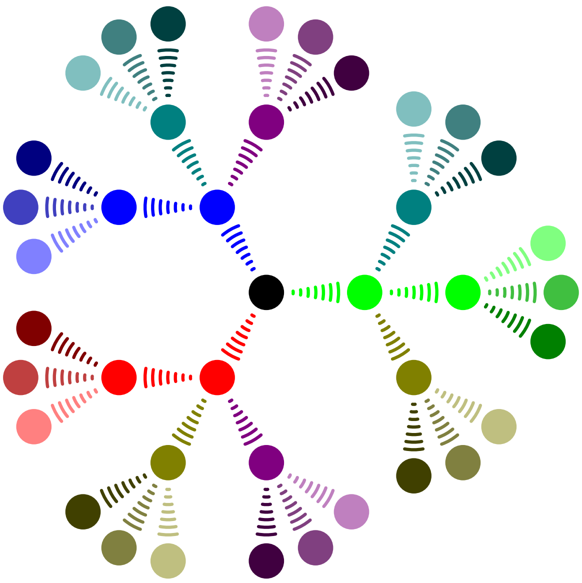
Ti*k*Z and PGF

Manual for version 1.18



```
\tikzstyle{level 1}=[sibling angle=120]
\tikzstyle{level 2}=[sibling angle=60]
\tikzstyle{level 3}=[sibling angle=30]
\tikzstyle{every node}=[fill]
\tikzstyle{edge from parent}=[snake=expanding waves,segment length=1mm,segment angle=10,draw]

\tikz [grow cyclic,shape=circle,very thick,level distance=13mm,cap=round]
  \node {} child [color=\A] foreach \A in {red,green,blue}
    { node {} child [color=\A!50!\B] foreach \B in {red,green,blue}
      { node {} child [color=\A!50!\B!50!\C] foreach \C in {black,gray,white}
        { node {} }
      }
    };
```

Für meinen Vater, damit er noch viele schöne TEX-Graphiken erschaffen kann.

Till

The Ti*k*Z and PGF Packages

Manual for version 1.18

`http://sourceforge.net/projects/pgf`

Till Tantau*

Institut für Theoretische Informatik
Universität zu Lübeck

June 12, 2007

Contents

*Editor of this documentation. Parts of this documentation have been written by other authors as indicated in these parts or chapters and in Section 1.5.

1 Introduction

The PGF package, where "PGF" is supposed to mean "portable graphics format" (or "pretty, good, functional" if you prefer...), is a package for creating graphics in an "inline" manner. It defines a number of TEX commands that draw graphics. For example, the code `\tikz \draw (0pt,0pt) -- (20pt,6pt);` yields the line and the code `\tikz \fill[orange] (1ex,1ex) circle (1ex);` yields .

In a sense, when you use PGF you "program" your graphics, just as you "program" your document when you use TEX. You get all the advantages of the "TEX-approach to typesetting" for your graphics: quick creation of simple graphics, precise positioning, the use of macros, often superior typography. You also inherit all the disadvantages: steep learning curve, no WYSIWYG, small changes require a long recompilation time, and the code does not really "show" how things will look like.

1.1 Structure of the System

The PGF system consists of different layers:

System layer: This layer provides a complete abstraction of what is going on "in the driver." The driver is a program like `dvips` or `dvipdfm` that takes a `.dvi` file as input and generates a `.ps` or a `.pdf` file. (The `pdftex` program also counts as a driver, even though it does not take a `.dvi` file as input. Never mind.) Each driver has its own syntax for the generation of graphics, causing headaches to everyone who wants to create graphics in a portable way. PGF's system layer "abstracts away" these differences. For example, the system command `\pgfsys@lineto{10pt}{10pt}` extends the current path to the coordinate (10pt, 10pt) of the current `{pgfpicture}`. Depending on whether `dvips`, `dvipdfm`, or `pdftex` is used to process the document, the system command will be converted to different `\special` commands. The system layer is as "minimalistic" as possible since each additional command makes it more work to port PGF to a new driver.

As a user, you will not use the system layer directly.

Basic layer: The basic layer provides a set of basic commands that allow you to produce complex graphics in a much easier manner than by using the system layer directly. For example, the system layer provides no commands for creating circles since circles can be composed from the more basic Bézier curves (well, almost). However, as a user you will want to have a simple command to create circles (at least I do) instead of having to write down half a page of Bézier curve support coordinates. Thus, the basic layer provides a command `\pgfpathcircle` that generates the necessary curve coordinates for you.

The basic layer is consists of a *core*, which consists of several interdependent packages that can only be loaded *en bloc,* and additional packages that extend the core by more special-purpose commands like node management or a plotting interface. For instance, the BEAMER package uses the core, but not all of the additional packages of the basic layer.

Frontend layer: A frontend (of which there can be several) is a set of commands or a special syntax that makes using the basic layer easier. A problem with directly using the basic layer is that code written for this layer is often too "verbose." For example, to draw a simple triangle, you may need as many as five commands when using the basic layer: One for beginning a path at the first corner of the triangle, one for extending the path to the second corner, one for going to the third, one for closing the path, and one for actually painting the triangle (as opposed to filling it). With the `tikz` frontend all this boils down to a single simple METAFONT-like command:

```
\draw (0,0) -- (1,0) -- (1,1) -- cycle;
```

There are different frontends:

- The Ti*k*Z frontend is the "natural" frontend for PGF. It gives you access to all features of PGF, but it is intended to be easy to use. The syntax is a mixture of METAFONT and PSTRICKS and some ideas of myself. This frontend is *neither* a complete METAFONT compatibility layer nor a PSTRICKS compatibility layer and it is not intended to become either.
- The `pgfpict2e` frontend reimplements the standard LATEX `{picture}` environment and commands like `\line` or `\vector` using the PGF basic layer. This layer is not really "necessary" since the `pict2e.sty` package does at least as good a job at reimplementing the `{picture}` environment. Rather, the idea behind this package is to have a simple demonstration of how a frontend can be implemented.

It would be possible to implement a pgftricks frontend that maps PSTRICKS commands to PGF commands. However, I have not done this and even if fully implemented, many things that work in PSTRICKS will not work, namely whenever some PSTRICKS command relies too heavily on PostScript trickery. Nevertheless, such a package might be useful in some situations.

As a user of PGF you will use the commands of a frontend plus perhaps some commands of the basic layer. For this reason, this manual explains the frontends first, then the basic layer, and finally the system layer.

1.2 Comparison with Other Graphics Packages

PGF is not the only graphics package for TeX. In the following, I try to give a reasonably fair comparison of the PGF-system and other packages.

1. The standard LaTeX {picture} environment allows you to create simple graphics, but little more. This is certainly not due to a lack of knowledge or imagination on the part of LaTeX's designer(s). Rather, this is the price paid for the {picture} environment's portability: It works together with all backend drivers.

2. The pstricks package is certainly powerful enough to create any conceivable kind of graphic, but it is not portable at all. Most importantly, it does not work with pdftex nor with any other driver that produces anything but PostScript code.

   Compared to PGF, pstricks has a broader support base. There are many nice extra packages for special purpose sitations that have been contributed by users over the last decade.

   The Ti*k*Z syntax is more consistent than the pstricks syntax as Ti*k*Z was developed "in a more centralized manner" and also "with the shortcomings on pstricks in mind."

   Note that a number of neat tricks that pstricks can do are impossible in PGF. In particular, pstricks has access to the powerful PostScript programming language, which allows trickery such as inline function plotting.

3. The xypic package is an older package for creating graphics. However, it is more difficult to use and to learn because the syntax and the documentation are a bit cryptic.

4. The dratex package is a small graphic package for creating a graphics. Compared to the other package, including PGF, it is very small, which may or may not be an advantage.

5. The metapost program is a very powerful alternative to PGF. However, it is an external program, which entails a bunch of problems. The time needed both to create a small graphic and also to compile it is much greater than in PGF. The main problem with metapost, however, is the inclusion of labels. This is *much* easier to achieve using PGF.

6. The xfig program is an important alternative to Ti*k*Z for users who do not wish to "program" their graphics as is necessary with Ti*k*Z and the other packages above. Their is a conversion program that will convert xfig graphics to both Ti*k*Z and for PGF, but it is still under construction.

1.3 Utilities: Page Management

The PGF package include a special subpackage called pgfpages, which is used to assemble several pages into a single page. This package is not really about creating graphics, but it is part of PGF nevertheless, mostly because its implementation uses PGF heavily.

The subpackage pgfpages provides commands for assembling several "virtual pages" into a single "physical page." The idea is that whenever TeX has a page ready for "shipout," pgfpages interrupts this shipout and instead stores the page to be shipped out in a special box. When enough "virtual pages" have been accumulated in this way, they are scaled down and arranged on a "physical page," which then *really* shipped out. This mechanism allows you to create "two page on one page" versions of a document directly inside LaTeX without the use of any external programs.

However, pgfpages can do quite a lot more than that. You can use it to put logos and watermark on pages, print up to 16 pages on one page, add borders to pages, and more.

1.4 How to Read This Manual

This manual describes both the design of the PGF system and its usage. The organization is very roughly according to “user-friendliness.” The commands and subpackages that are easiest and most frequently used are described first, more low-level and esoteric features are discussed later.

If you have not yet installed PGF, please read the installation first. Second, it might be a good idea to read the tutorial. Finally, you might wish to skim through the description of Ti*k*Z. Typically, you will not need to read the sections on the basic layer. You will only need to read the part on the system layer if you intend to write your own frontend or if you wish to port PGF to a new driver.

The “public” commands and environments provided by the `pgf` package are described throughout the text. In each such description, the described command, environment or option is printed in red. Text shown in green is optional and can be left out.

1.5 Authors and Acknowledgements

The bulk of the PGF system and its documentation was written by Till Tantau. The PGF mathematical engine was written and documented by Mark Wibrow. Additionally, numerous people have contributed to the PGF system by writing emails, spotting bugs, or sending libraries. Many thanks to all these people, who are too numerous to name them all!

1.6 Getting Help

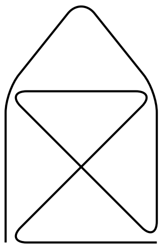
When you need help with PGF and Ti*k*Z, please do the following:

1. Read the manual, at least the part that has to do with your problem.
2. If that does not solve the problem, try having a look at the sourceforge development page for PGF and Ti*k*Z (see the title of this document). Perhaps someone has already reported a similar problem and someone has found a solution.
3. On the website you will find numerous forums for getting help. There, you can write to help forums, file bug reports, join mailing lists, and so on.
4. Before you file a bug report, especially a bug report concerning the installation, make sure that this is really a bug. In particular, have a look at the `.log` file that results when you TeX your files. This `.log` file should show that all the right files are loaded from the right directories. Nearly all installation problems can be resolved by looking at the `.log` file.
5. *As a last resort* you can try to email me (Till Tantau) or, if the problem concerns the mathematical engine, Mark Wibrow. I do not mind getting emails, I simply get way too many of them. Because of this, I cannot guarantee that your emails will be answered timely or even at all. Your chances that your problem will be fixed are somewhat higher if you mail to the PGF mailing list (naturally, I read this list and answer questions when I have the time).
6. Please, do not phone me in my office (unless, of course, you attend one of my lectures).

Part I
Tutorials and Guidelines

by Till Tantau

To help you get started with TikZ, instead of a long installation and configuration section, this manual starts with tutorials. They explain all the basic and some of the more advanced features of the system, without going into all the details. This part also contains some guidelines on how you should proceed when creating graphics using TikZ.



```
\tikz \draw[thick,rounded corners=8pt]
  (0,0) -- (0,2) -- (1,3.25) -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
```

2 Tutorial: A Picture for Karl's Students

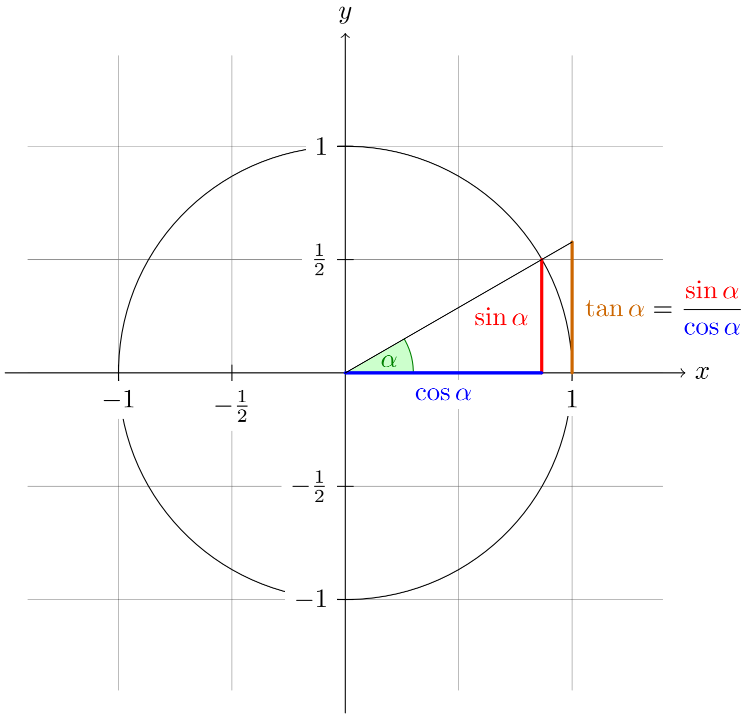
This tutorial is intended for new users of PGF and Ti*k*Z. It does not give an exhaustive account of all the features of Ti*k*Z or PGF, just of those that you are likely to use right away.

Karl is a math and chemistry high-school teacher. He used to create the graphics in his worksheets and exams using LaTeX's `{picture}` environment. While the results were acceptable, creating the graphics often turned out to be a lengthy process. Also, there tended to be problems with lines having slightly wrong angles and circles also seemed to be hard to get right. Naturally, his students could not care less whether the lines had the exact right angles and they find Karl's exams too difficult no matter how nicely they were drawn. But Karl was never entirely satisfied with the result.

Karl's son, who was even less satisfied with the results (he did not have to take the exams, after all), told Karl that he might wish to try out a new package for creating graphics. A bit confusingly, this package seems to have two names: First, Karl had to download and install a package called PGF. Then it turns out that inside this package there is another package called Ti*k*Z, which is supposed to stand for "Ti*k*Z ist *kein* Zeichenprogramm." Karl finds this all a bit strange and Ti*k*Z seems to indicate that the package does not do what he needs. However, having used GNU software for quite some time and "GNU not being Unix," there seems to be hope yet. His son assures him that Ti*k*Z's name is intended to warn people that Ti*k*Z is not a program that you can use to draw graphics with your mouse or tablet. Rather, it is more like a "graphics language."

2.1 Problem Statement

Karl wants to put a graphic on the next worksheet for his students. He is currently teaching his students about sine and cosine. What he would like to have is something that looks like this (ideally):



The angle α is 30° in the example ($\pi/6$ in radians). The sine of α, which is the height of the red line, is

$$\sin\alpha = 1/2.$$

By the Theorem of Pythagoras we have $\cos^2\alpha + \sin^2\alpha = 1$. Thus the length of the blue line, which is the cosine of α, must be

$$\cos\alpha = \sqrt{1 - 1/4} = \tfrac{1}{2}\sqrt{3}.$$

This shows that $\tan\alpha$, which is the height of the orange line, is

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha} = 1/\sqrt{3}.$$

2.2 Setting up the Environment

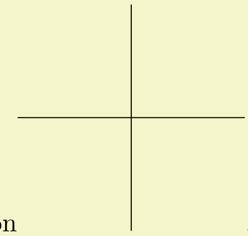
In Ti*k*Z, to draw a picture, at the start of the picture you need to tell TeX or LaTeX that you want to start a picture. In LaTeX this is done using the environment `{tikzpicture}`, in plain TeX you just use `\tikzpicture` to start the picture and `\endtikzpicture` to end it.

2.2.1 Setting up the Environment in LaTeX

Karl, being a LaTeX user, thus sets up his file as follows:

```
\documentclass{article} % say
\usepackage{tikz}
\begin{document}
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
\end{document}
```

When executed, that is, run via `pdflatex` or via `latex` followed by `dvips`, the resulting will contain something that looks like this:

We are working on  .

```
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
```

Admittedly, not quite the whole picture, yet, but we do have the axes established. Well, not quite, but we have the lines that make up the axes drawn. Karl suddenly has a sinking feeling that the picture is still some way off.

Let's have a more detailed look at the code. First, the package `tikz` is loaded. This package is a so-called "frontend" to the basic PGF system. The basic layer, which is also described in this manual, is somewhat more, well, basic and thus harder to use. The frontend makes things easier by providing a simpler syntax.

Inside the environment there are two `\draw` commands. They mean: "The path, which is specified following the command up to the semicolon, should be drawn." The first path is specified as `(-1.5,0) -- (0,1.5)`, which means "a straight line from the point at position $(-1.5, 0)$ to the point at position $(0, 1.5)$." Here, the positions are specified within a special coordinate system in which, initially, one unit is 1cm.

Karl is quite pleased to note that the environment automatically reserves enough space to encompass the picture.

2.2.2 Setting up the Environment in Plain TEX

Karl's wife Gerda, who also happens to be a math teacher, is not a LATEX user, but uses plain TEX since she prefers to do things "the old way." She can also use Ti*k*Z. Instead of `\usepackage{tikz}` she has to write `\input tikz.tex` and instead of `\begin{tikzpicture}` she writes `\tikzpicture` and instead of `\end{tikzpicture}` she writes `\endtikzpicture`.

Thus, she would use:

```
%% Plain TeX file
\input tikz.tex
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
We are working on
\tikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\endtikzpicture.
\bye
```

Gerda can typeset this file using either `pdftex` or `tex` together with `dvips`. Ti*k*Z will automatically discern which driver she is using. If she wishes to use `dvipdfm` together with `tex`, she either needs to modify the file `pgf.cfg` or can write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` somewhere *before* she inputs `tikz.tex` or `pgf.tex`.

2.2.3 Setting up the Environment in ConTEXt

Karl's uncle Hans uses ConTEXt. Like Gerda, Hans can also use Ti*k*Z. Instead of `\usepackage{tikz}` he says `\usemodule[tikz]`. Instead of `\begin{tikzpicture}` he writes `\starttikzpicture` and instead of `\end{tikzpicture}` he writes `\stoptikzpicture`.

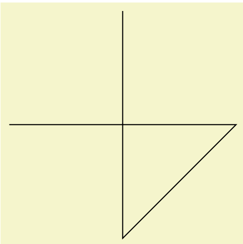
His version of the example looks like this:

```
%% ConTeXt file
\usemodule[tikz]

\starttext
  We are working on
  \starttikzpicture
    \draw (-1.5,0) -- (1.5,0);
    \draw (0,-1.5) -- (0,1.5);
  \stoptikzpicture.
\stoptext
```

Hans will now typeset this file in the usual way using `texexec`.

2.3 Straight Path Construction

The basic building block of all pictures in Ti*k*Z is the path. A *path* is a series of straight lines and curves that are connected (that is not the whole picture, but let us ignore the complications for the moment). You start a path by specifying the coordinates of the start position as a point in round brackets, as in `(0,0)`. This is followed by a series of "path extension operations." The simplest is `--`, which we used already. It must be followed by another coordinate and it extends the path in a straight line to this new position. For example, if we were to turn the two paths of the axes into one path, the following would result:



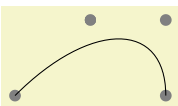
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

Karl is a bit confused by the fact that there is no `{tikzpicture}` environment, here. Instead, the little command `\tikz` is used. This command either takes one argument (starting with an opening brace as in `\tikz{\draw (0,0) -- (1.5,0)}`, which yields ———) or collects everything up to the next semicolon and puts it inside a `{tikzpicture}` environment. As a rule of thumb, all Ti*k*Z graphic drawing commands must occur as an argument of `\tikz` or inside a `{tikzpicture}` environment. Fortunately, the command `\draw` will only be defined inside this environment, so there is little chance that you will accidentally do something wrong here.

2.4 Curved Path Construction

The next thing Karl wants to do is to draw the circle. For this, straight lines obviously will not do. Instead, we need some way to draw curves. For this, Ti*k*Z provides a special syntax. One or two "control points" are needed. The math behind them is not quite trivial, but here is the basic idea: Suppose you are at point x and the first control point is y. Then the curve will start "going in the direction of y at x," that is, the tangent of the curve at x will point toward y. Next, suppose the curve should end at z and the second support point is w. Then the curve will, indeed, end at z and the tangent of the curve at point z will go through w.

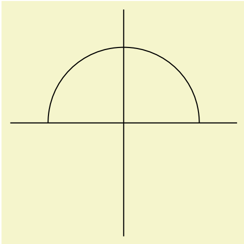
Here is an example (the control points have been added for clarity):



```
\begin{tikzpicture}
  \filldraw [gray] (0,0) circle (2pt)
                   (1,1) circle (2pt)
                   (2,1) circle (2pt)
                   (2,0) circle (2pt);
  \draw (0,0) .. controls (1,1) and (2,1) .. (2,0);
\end{tikzpicture}
```

The general syntax for extending a path in a "curved" way is `.. controls` ⟨*first control point*⟩ `and` ⟨*second control point*⟩ `..` ⟨*end point*⟩. You can leave out the `and` ⟨*second control point*⟩, which causes the first one to be used twice.

So, Karl can now add the first half circle to the picture:

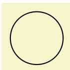


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (-1,0) .. controls (-1,0.555) and (-0.555,1) .. (0,1)
               .. controls (0.555,1) and (1,0.555) .. (1,0);
\end{tikzpicture}
```

Karl is happy with the result, but finds specifying circles in this way to be extremely awkward. Fortunately, there is a much simpler way.

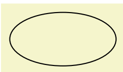
2.5 Circle Path Construction

In order to draw a circle, the path construction operation `circle` can be used. This operation is followed by a radius in round brackets as in the following example: (Note that the previous position is used as the *center* of the circle.)



```
\tikz \draw (0,0) circle (10pt);
```

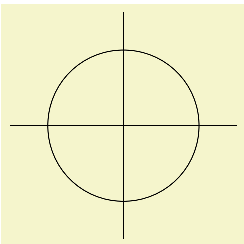
You can also append an ellipse to the path using the `ellipse` operation. Instead of a single radius you can specify two of them, one for the x-direction and one for the y-direction, separated by `and`:



```
\tikz \draw (0,0) ellipse (20pt and 10pt);
```

To draw an ellipse whose axes are not horizontal and vertical, but point in an arbitrary direction (a "turned ellipse" like ) you can use transformations, which are explained later. The code for the little ellipse is `\tikz \draw[rotate=30] (0,0) ellipse (6pt and 3pt);`, by the way.

So, returning to Karl's problem, he can write `\draw (0,0) circle (1cm);` to draw the circle:

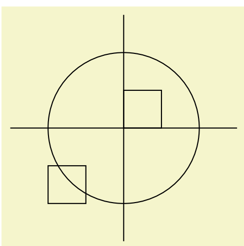


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
\end{tikzpicture}
```

At this point, Karl is a bit alarmed that the circle is so small when he wants the final picture to be much bigger. He is pleased to learn that Ti*k*Z has powerful transformation options and scaling everything by a factor of three is very easy. But let us leave the size as it is for the moment to save some space.

2.6 Rectangle Path Construction

The next things we would like to have is the grid in the background. There are several ways to produce it. For example, one might draw lots of rectangles. Since rectangles are so common, there is a special syntax for them: To add a rectangle to the current path, use the `rectangle` path construction operation. This operation should be followed by another coordinate and will append a rectangle to the path such that the previous coordinate and the next coordinates are corners of the rectangle. So, let us add two rectangles to the picture:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (0,0) rectangle (0.5,0.5);
  \draw (-0.5,-0.5) rectangle (-1,-1);
\end{tikzpicture}
```

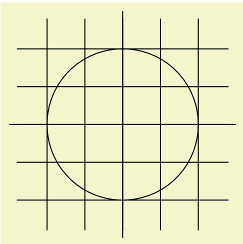
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed."

So, Karl is about to resort to simply drawing four vertical and four horizontal lines using the nice `\draw` command, when he learns that there is a `grid` path construction operation.

2.7 Grid Path Construction

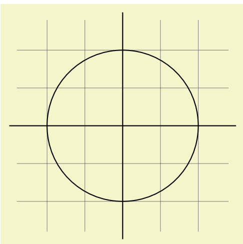
The `grid` path operation adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the `grid` operation. For example, the code `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` produces . Note how the optional argument for `\draw` can be used to specify a grid width (there are also `xstep` and `ystep` to define the steppings independently). As Karl will learn soon, there are *lots* of things that can be influenced using such options.

For Karl, the following code could be used:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw[step=.5cm] (-1.4,-1.4) grid (1.4,1.4);
\end{tikzpicture}
```

Having another look at the desired picture, Karl notices that it would be nice for the grid to be more subdued. (His son told him that grids tend to be distracting if they are not subdued.) To subdue the grid, Karl adds two more options to the `\draw` command that draws the grid. First, he uses the color `gray` for the grid lines. Second, he reduces the line width to `very thin`. Finally, he swaps the ordering of the commands so that the grid is drawn first and everything else on top.



```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
\end{tikzpicture}
```

2.8 Adding a Touch of Style

Instead of the options `gray,very thin` Karl could also have said `style=help lines`. *Styles* are predefined sets of options that can be used to organize how a graphic is drawn. By saying `style=help lines` you say "use the style that I (or someone else) has set for drawing help lines." If Karl decides, at some later point, that grids should be drawn, say, using the color `blue!50` instead of `gray`, he could say the following:

```
\tikzstyle help lines=[color=blue!50,very thin]
```

Alternatively, he could have said the following:

```
\tikzstyle help lines+=[color=blue!50]
```

This would have added the `color=blue!50` option. The `help lines` style would now contain *two* color options, but the second would override the first.

Using styles makes your graphics code more flexible. You can change the way things look easily in a consistent manner.

To build a hierarchy of styles you can have one style use another. So in order to define a style `Karl's grid` that is based on the `grid` style Karl could say

```
\tikzstyle Karl's grid=[style=help lines,color=blue!50]
...
\draw[style=Karl's grid] (0,0) grid (5,5);
```

You can also leave out the `style=`. Thus, whenever TikZ encounters an options that it does not know about, it will check whether this option happens to be the name of a style. If so, the style is used. Thus, Karl could also have written:

```
\tikzstyle Karl's grid=[help lines,color=blue!50]
...
\draw[Karl's grid] (0,0) grid (5,5);
```

For some styles, like the `very thin` style, it is pretty clear what the style does and there is no need to say `style=very thin`. For other styles, like `help lines`, it seems more natural to me to say `style=help lines`. But, mainly, this is a matter of taste.

2.9 Drawing Options

Karl wonders what other options there are that influence how a path is drawn. He saw already that the `color=`⟨*color*⟩ option can be used to set the line's color. The option `draw=`⟨*color*⟩ does nearly the same, only it sets the color for the lines only and a different color can be used for filling (Karl will need this when he fills the arc for the angle).

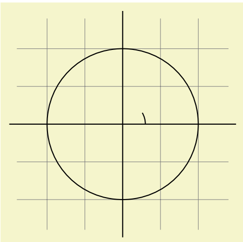
He saw that the style `very thin` yields very thin lines. Karl is not really surprised by this and neither is he surprised to learn that `thin` yields thin lines, `thick` yields thick lines, `very thick` yields very thick lines, `ultra thick` yields really, really thick lines and `ultra thin` yields lines that are so thin that low-resolution printers and displays will have trouble showing them. He wonders what gives lines of "normal" thickness. It turns out that `thin` is the correct choice. This seems strange to Karl, but his son explains him that LaTeX has two commands called `\thinlines` and `\thicklines` and that `\thinlines` gives the line width of "normal" lines, more precisely, of the thickness that, say, the stem of a letter like "T" or "i" has. Nevertheless, Karl would like to know whether there is anything "in the middle" between `thin` and `thick`. There is: `semithick`.

Another useful thing one can do with lines is to dash or dot them. For this, the two styles `dashed` and `dotted` can be used, yielding - - and ...... Both options also exist in a loose and a dense version, called `loosely dashed`, `densely dashed`, `loosely dotted`, and `densely dotted`. If he really, really needs to, Karl can also define much more complex dashing patterns with the `dash pattern` option, but his son insists that dashing is to be used with utmost care and mostly distracts. Karl's son claims that complicated dashing patterns are evil. Karl's students do not care about dashing patterns.

2.10 Arc Path Construction

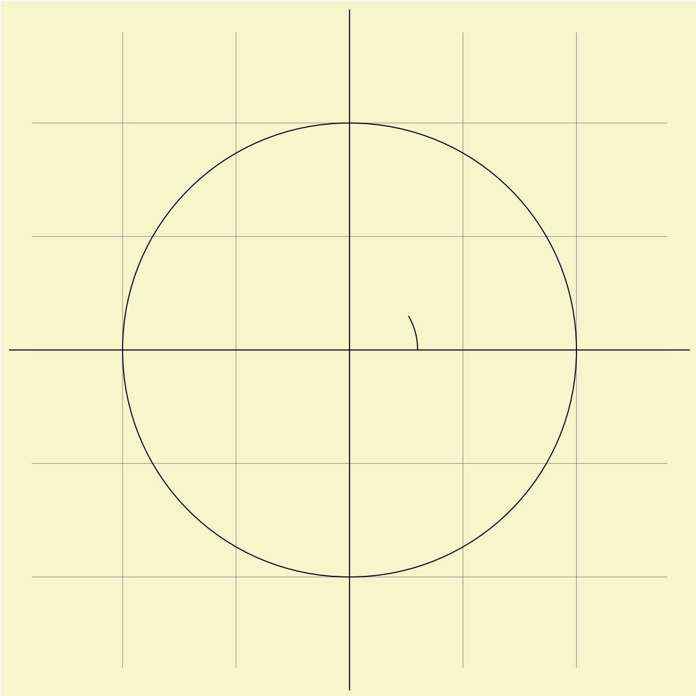
Our next obstacle is to draw the arc for the angle. For this, the `arc` path construction operation is useful, which draws part of a circle or ellipse. This `arc` operation must be followed by a triple in rounded brackets, where the components of the triple are separated by colons. The first two components are angles, the last one is a radius. An example would be `(10:80:10pt)`, which means "an arc from 10 degrees to 80 degrees on a circle of radius 10pt." Karl obviously needs an arc from 0° to 30°. The radius should be something relatively small, perhaps around one third of the circle's radius. This gives: `(0:30:3mm)`.

When one uses the arc path construction operation, the specified arc will be added with its starting point at the current position. So, we first have to "get there."



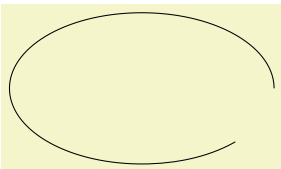
```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

Karl thinks this is really a bit small and he cannot continue unless he learns how to do scaling. For this, he can add the `[scale=3]` option. He could add this option to each `\draw` command, but that would be awkward. Instead, he adds it to the whole environment, which causes this option to apply to everything within.



```
\begin{tikzpicture}[scale=3]
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

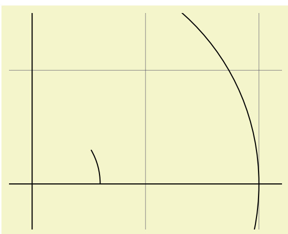
As for circles, you can specify "two" radii in order to get an elliptical arc.



```
\tikz \draw (0,0) arc (0:315:1.75cm and 1cm);
```

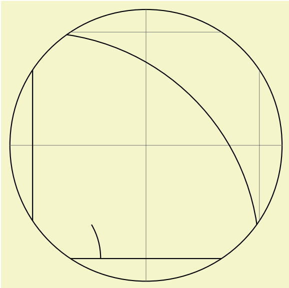
2.11 Clipping a Path

In order to save space in this manual, it would be nice to clip Karl's graphics a bit so that we can focus on the "interesting" parts. Clipping is pretty easy in Ti*k*Z. You can use the `\clip` command clip all subsequent drawing. It works like `\draw`, only it does not draw anything, but uses the given path to clip everything subsequently.



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

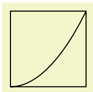
You can also do both at the same time: Draw *and* clip a path. For this, use the `\draw` command and add the `clip` option. (This is not the whole picture: You can also use the `\clip` command and add the `draw` option. Well, that is also not the whole picture: In reality, `\draw` is just a shorthand for `\path[draw]` and `\clip` is a shorthand for `\path[clip]` and you could also say `\path[draw,clip]`.) Here is an example:



```
\begin{tikzpicture}[scale=3]
  \clip[draw] (0.5,0.5) circle (.6cm);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \draw (3mm,0mm) arc (0:30:3mm);
\end{tikzpicture}
```

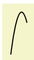
2.12 Parabola and Sine Path Construction

Although Karl does not need them for his picture, he is pleased to learn that there are `parabola` and `sin` and `cos` path operations for adding parabolas and sine and cosine curves to the current path. For the `parabola` operation, the current point will lie on the parabola as well as the point given after the parabola operation. Consider the following example:



```
\tikz \draw (0,0) rectangle (1,1)  (0,0) parabola (1,1);
```

It is also possible to place the bend somewhere else:

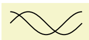


```
\tikz \draw[x=1pt,y=1pt] (0,0) parabola bend (4,16) (6,12);
```

The operations `sin` and `cos` add a sine or cosine curve in the interval $[0, \pi/2]$ such that the previous current point is at the start of the curve and the curve ends at the given end point. Here are two examples:

A sine curve.

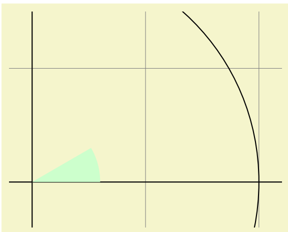
```
A sine \tikz \draw[x=1ex,y=1ex] (0,0) sin (1.57,1); curve.
```



```
\tikz \draw[x=1.57ex,y=1ex] (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
                            (0,1) cos (1,0) sin (2,-1) cos (3,0) sin (4,1);
```

2.13 Filling and Drawing

Returning to the picture, Karl now wants the angle to be "filled" with a very light green. For this he uses `\fill` instead of `\draw`. Here is what Karl does:



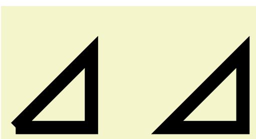
```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \fill[green!20!white] (0,0) -- (3mm,0mm) arc (0:30:3mm) -- (0,0);
\end{tikzpicture}
```

The color `green!20!white` means 20% green and 80% white mixed together. Such color expression are possible since PGF uses Uwe Kern's `xcolor` package, see the documentation of that package for details on color expressions.

What would have happened, if Karl had not "closed" the path using `--(0,0)` at the end? In this case, the path is closed automatically, so this could have been omitted. Indeed, it would even have been better to write the following, instead:

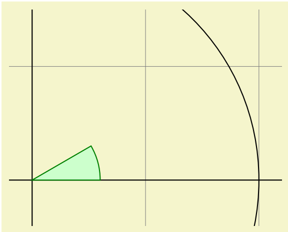
```
\fill[green!20!white] (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
```

The `--cycle` causes the current path to be closed (actually the current part of the current path) by smoothly joining the first and last point. To appreciate the difference, consider the following example:



```
\begin{tikzpicture}[line width=5pt]
  \draw (0,0) -- (1,0) -- (1,1) -- (0,0);
  \draw (2,0) -- (3,0) -- (3,1) -- cycle;
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

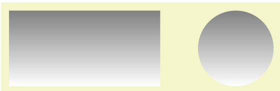
You can also fill and draw a path at the same time using the `\filldraw` command. This will first draw the path, then fill it. This may not seem too useful, but you can specify different colors to be used for filling and for stroking. These are specified as optional arguments like this:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20!white, draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
\end{tikzpicture}
```

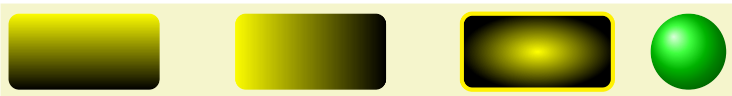
2.14 Shading

Karl briefly considers the possibility of making the angle "more fancy" by *shading* it. Instead of filling the with a uniform color, a smooth transition between different colors is used. For this, `\shade` and `\shadedraw`, for shading and drawing at the same time, can be used:



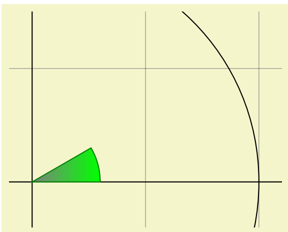
```
  \tikz \shade (0,0) rectangle (2,1)  (3,0.5) circle (.5cm);
```

The default shading is a smooth transition from gray to white. To specify different colors, you can use options:



```
\begin{tikzpicture}[rounded corners,ultra thick]
  \shade[top color=yellow,bottom color=black] (0,0) rectangle +(2,1);
  \shade[left color=yellow,right color=black] (3,0) rectangle +(2,1);
  \shadedraw[inner color=yellow,outer color=black,draw=yellow] (6,0) rectangle +(2,1);
  \shade[ball color=green] (9,.5) circle (.5cm);
\end{tikzpicture}
```

For Karl, the following might be appropriate:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \shadedraw[left color=gray,right color=green, draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
\end{tikzpicture}
```

However, he wisely decides that shadings usually only distract without adding anything to the picture.

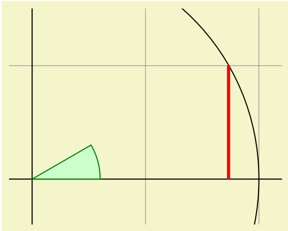
2.15 Specifying Coordinates

Karl now wants to add the sine and cosine lines. He knows already that he can use the `color=` option to set the lines's colors. So, what is the best way to specify the coordinates?

There are different ways of specifying coordinates. The easiest way is to say something like `(10pt,2cm)`. This means 10pt in x-direction and 2cm in y-directions. Alternatively, you can also leave out the units as in `(1,2)`, which means "one times the current x-vector plus twice the current y-vector." These vectors default to 1cm in the x-direction and 1cm in the y-direction, respectively.

In order to specify points in polar coordinates, use the notation `(30:1cm)`, which means 1cm in direction 30 degree. This is obviously quite useful to "get to the point $(\cos 30°, \sin 30°)$ on the circle."

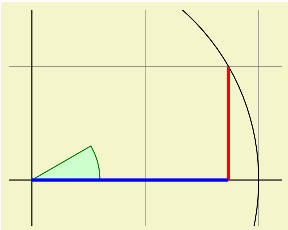
You can add a single `+` sign in front of a coordinate or two of them as in `+(1cm,0cm)` or `++(0cm,2cm)`. Such coordinates are interpreted differently: The first form means "1cm upwards from the previous specified position" and the second means "2cm to the right of the previous specified position, making this the new specified position." For example, we can draw the sine line as follows:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[red,very thick] (30:1cm) -- +(0,-0.5);
\end{tikzpicture}
```

Karl used the fact $\sin 30^\circ = 1/2$. However, he very much doubts that his students know this, so it would be nice to have a way of specifying "the point straight down from `(30:1cm)` that lies on the x-axis." This is, indeed, possible using a special syntax: Karl can write `(30:1cm |- 0,0)`. In general, the meaning of `(`$\langle p\rangle$` |- `$\langle q\rangle$`)` is "the intersection of a vertical line through p and a horizontal line through q."

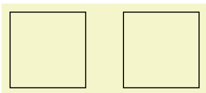
Next, let us draw the cosine line. One way would be to say `(30:1cm |- 0,0) -- (0,0)`. Another way is the following: we "continue" from where the sine ends:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[red,very thick]  (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick] (30:1cm) ++(0,-0.5) -- (0,0);
\end{tikzpicture}
```

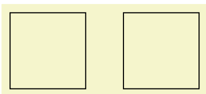
Note the there is no `--` between `(30:1cm)` and `+(0,-0.5)`. In detail, this path is interpreted as follows: "First, the `(30:1cm)` tells me to move by pen to $(\cos 30^\circ, 1/2)$. Next, there comes another coordinate specification, so I move my pen there without drawing anything. This new point is half a unit down from the last position, thus it is at $(\cos 30^\circ, 0)$. Finally, I move the pen to the origin, but this time drawing something (because of the `--`)."

To appreciate the difference between `+` and `++` consider the following example:



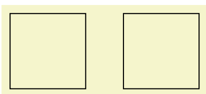
```
\begin{tikzpicture}
  \def\rectanglepath{-- ++(1cm,0cm)  -- ++(0cm,1cm)  -- ++(-1cm,0cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

By comparison, when using a single `+`, the coordinates are different:



```
\begin{tikzpicture}
  \def\rectanglepath{-- +(1cm,0cm)  -- +(1cm,1cm)  -- +(0cm,1cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

Naturally, all of this could have been written more clearly and more economically like this (either with a single of a double `+`):

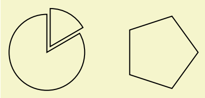


```
\tikz \draw (0,0) rectangle +(1,1)  (1.5,0) rectangle +(1,1);
```

Karl is left with the line for $\tan\alpha$, which seems difficult to specify using transformations and polar coordinates. For this he needs another way of specifying coordinates: Karl can specify intersections of lines as coordinates. The line for $\tan\alpha$ starts at $(1,0)$ and goes upward to a point that is at the intersection of a line going "up" and a line going from the origin through `(30:1cm)`. The syntax for this point is the following:

```
\draw[very thick,orange] (1,0) -- (intersection of 1,0--1,1 and 0,0--30:1cm);
```

In the following, two final examples of how to use relative positioning are presented. Note that the transformation options, which are explained later, are often more useful for shifting than relative positioning.

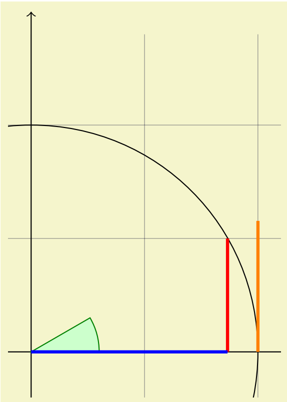


```
\begin{tikzpicture}[scale=0.5]
  \draw (0,0) -- (90:1cm) arc (90:360:1cm) arc (0:30:1cm) -- cycle;
  \draw (60:5pt) -- +(30:1cm) arc (30:90:1cm) -- cycle;

  \draw (3,0)  +(0:1cm) -- +(72:1cm) -- +(144:1cm) -- +(216:1cm) --
               +(288:1cm) -- cycle;
\end{tikzpicture}
```

2.16 Adding Arrow Tips

Karl now wants to add the little arrow tips at the end of the axes. He has noticed that in many plots, even in scientific journals, these arrow tips seem to missing, presumably because the generating programs cannot produce them. Karl thinks arrow tips belong at the end of axes. His son agrees. His students do not care about arrow tips.

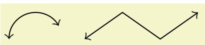
It turns out that adding arrow tips is pretty easy: Karl adds the option `->` to the drawing commands for the axes:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[red,very thick]    (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick]   (30:1cm) ++(0,-0.5) -- (0,0);
  \draw[orange,very thick] (1,0) -- (intersection of 1,0--1,1 and 0,0--30:1cm);
\end{tikzpicture}
```

If Karl had used the option `<-` instead of `->`, arrow tips would have been put at the beginning of the path. The option `<->` puts arrow tips at both ends of the path.

There are certain restrictions to the kind of paths to which arrow tips can be added. As a rule of thumb, you can add arrow tips only to a single open "line." For example, you should not try to add tips to, say, a rectangle or a circle. (You can try, but no guarantees as to what will happen now or in future versions.) However, you can add arrow tips to curved paths and to paths that have several segments, as in the following examples:



```
\begin{tikzpicture}
  \draw [<->] (0,0) arc (180:30:10pt);
  \draw [<->] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl has a more detailed look at the arrow that Ti*k*Z puts at the end. It looks like this when he zooms it: $\rightarrow$. The shape seems vaguely familiar and, indeed, this is exactly the end of TeX's standard arrow used in something like $f\colon A \to B$.

Karl likes the arrow, especially since it is not "as thick" as the arrows offered by many other packages. However, he expects that, sometimes, he might need to use some other kinds of arrow. To do so, Karl can say `>=`⟨*right arrow tip kind*⟩, where ⟨*right arrow tip kind*⟩ is a special arrow tip specification. For example, if Karl says `>=stealth`, then he tells Ti*k*Z that he would like "stealth-fighter-like" arrow tips:

```
\begin{tikzpicture}[>=stealth]
  \draw [->] (0,0) arc (180:30:10pt);
  \draw [<<-,very thick] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl wonders whether such a military name for the arrow type is really necessary. He is not really mollified when his son tells him that Microsoft's PowerPoint uses the same name. He decides to have his students discuss this at some point.

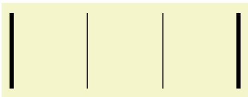
In addition to `stealth` there are several other predefined arrow tip kinds Karl can choose from, see Section 18. Furthermore, he can define arrows types himself, if he needs new ones.

2.17 Scoping

Karl saw already that there are numerous graphic options that affect how paths are rendered. Often, he would like to apply certain options to a whole set of graphic commands. For example, Karl might wish to draw three paths using a `thick` pen, but would like everything else to be drawn "normally."

If Karl wishes to set a certain graphic option for the whole picture, he can simply pass this option to the `\tikz` command or to the `{tikzpicture}` environment (Gerda would pass the options to `\tikzpicture` and Hans passes them to `\starttikzpicture`). However, if Karl wants to apply graphic options to a local group, he put these commands inside a `{scope}` environment (Gerda uses `\scope` and `\endscope`, Hans uses `\startscope` and `\stopscope`). This environment takes graphic options as an optional argument and these options apply to everything inside the scope, but not to anything outside.

Here is an example:



```
\begin{tikzpicture}[ultra thick]
  \draw (0,0) -- (0,1);
  \begin{scope}[thin]
    \draw (1,0) -- (1,1);
    \draw (2,0) -- (2,1);
  \end{scope}
  \draw (3,0) -- (3,1);
\end{tikzpicture}
```

Scoping has another interesting effect: Any changes to the clipping area are local to the scope. Thus, if you say `\clip` somewhere inside a scope, the effect of the `\clip` command ends at the end of the scope. This is useful since there is no other way of "enlarging" the clipping area.

Karl has also already seen that giving options to commands like `\draw` apply only to that command. In turns out that the situation is slightly more complex. First, options to a command like `\draw` are not really options to the command, but they are "path options" and can be given anywhere on the path. So, instead of `\draw[thin] (0,0) -- (1,0);` one can also write `\draw (0,0) [thin] -- (1,0);` or `\draw (0,0) -- (1,0) [thin];`; all of these have the same effect. This might seem strange since in the last case, it would appear that the `thin` should take effect only "after" the line from $(0,0)$ to $(1,0)$ has been draw. However, most graphic options only apply to the whole path. Indeed, if you say both `thin` and `thick` on the same path, the last option given will "win."

When reading the above, Karl notices that only "most" graphic options apply to the whole path. Indeed, all transformation options do *not* apply to the whole path, but only to "everything following them on the path." We will have a more detailed look at this in a moment. Nevertheless, all options given during a path construction apply only to this path.

2.18 Transformations

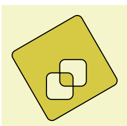
When you specify a coordinate like `(1cm,1cm)`, where is that coordinate placed on the page? To determine the position, Ti*k*Z, TeX, and PDF or PostScript all apply certain transformations to the given coordinate in order to determine the finally position on the page.

Ti*k*Z provides numerous options that allow you to transform coordinates in PGF's private coordinate system. For example, the `xshift` option allows you to shift all subsequent points by a certain amount:


```
\tikz \draw (0,0) -- (0,0.5) [xshift=2pt] (0,0) -- (0,0.5);
```

It is important to note that you can change transformation "in the middle of a path," a feature that is not supported by PDF or PostScript. The reason is that PGF keeps track of its own transformation matrix.

Here is a more complicated example:



```
\begin{tikzpicture}[even odd rule,rounded corners=2pt,x=10pt,y=10pt]
  \filldraw[fill=examplefill] (0,0)   rectangle (1,1)
    [xshift=5pt,yshift=5pt]   (0,0)   rectangle (1,1)
                  [rotate=30]  (-1,-1) rectangle (2,2);
\end{tikzpicture}
```

The most useful transformations are `xshift` and `yshift` for shifting, `shift` for shifting to a given point as in `shift={(1,0)}` or `shift={+(0,0)}` (the braces are necessary so that TeX does not mistake the comma for separating options), `rotate` for rotating by a certain angle (there is also a `rotate around` for rotating around a given point), `scale` for scaling by a certain factor, `xscale` and `yscale` for scaling only in the x- or y-direction (`xscale=-1` is a flip), and `xslant` and `yslant` for slanting. If these transformation and those that I have not mentioned are not sufficient, the `cm` option allows you to apply an arbitrary transformation matrix. Karl's students, by the way, do not know what a transformation matrix is.

2.19 Repeating Things: For-Loops

Karl's next aim is to add little ticks on the axes at positions -1, $-1/2$, $1/2$, and 1. For this, it would be nice to use some kind of "loop," especially since he wishes to do the same thing at each of these positions. There are different packages for doing this. LaTeX has its own internal command for this, `pstricks` comes along with the powerful `\mulitdo` command. All of these can be used together with PGF and Ti*k*Z, so if you are familiar with them, feel free to use them. PGF introduces yet another command, called `\foreach`, which I introduced since I could never remember the syntax of the other packages. `\foreach` is defined in the package `pgffor` and can be used independently of PGF. Ti*k*Z includes it automatically.

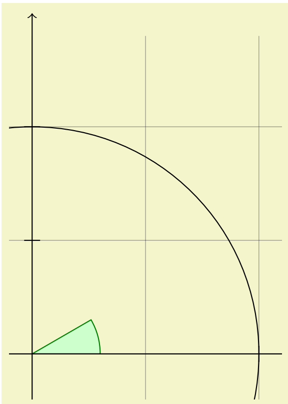
In its basic form, the `\foreach` command is easy to use:

$x = 1$, $x = 2$, $x = 3$,

```
\foreach \x in {1,2,3} {$x =\x$, }
```

The general syntax is `\foreach` ⟨*variable*⟩ `in {`⟨*list of values*⟩`}` ⟨*commands*⟩. Inside the ⟨*commands*⟩, the ⟨*variable*⟩ will be assigned to the different values. If the ⟨*commands*⟩ do not start with a brace, everything up to the next semicolon is used as ⟨*commands*⟩.

For Karl and the ticks on the axes, he could use the following code:

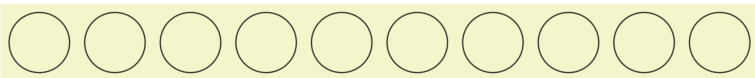


```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);

  \foreach \x in {-1cm,-0.5cm,1cm}
    \draw (\x,-1pt) -- (\x,1pt);
  \foreach \y in {-1cm,-0.5cm,0.5cm,1cm}
    \draw (-1pt,\y) -- (1pt,\y);
\end{tikzpicture}
```

As a matter of fact, there are many different ways of creating the ticks. For example, Karl could have put the `\draw ...;` inside curly braces. He could also have used, say,

```
\foreach \x in {-1,-0.5,1}
  \draw[xshift=\x cm] (0pt,-1pt) -- (0pt,1pt);
```

Karl is curious what would happen in a more complicated situation where there are, say, 20 ticks. It seems bothersome to explicitly mention all these numbers in the set for `\foreach`. Indeed, it is possible to use `...` inside the `\foreach` statement to iterate over a large number of values (which must, however, be dimensionless real numbers) as in the following example:

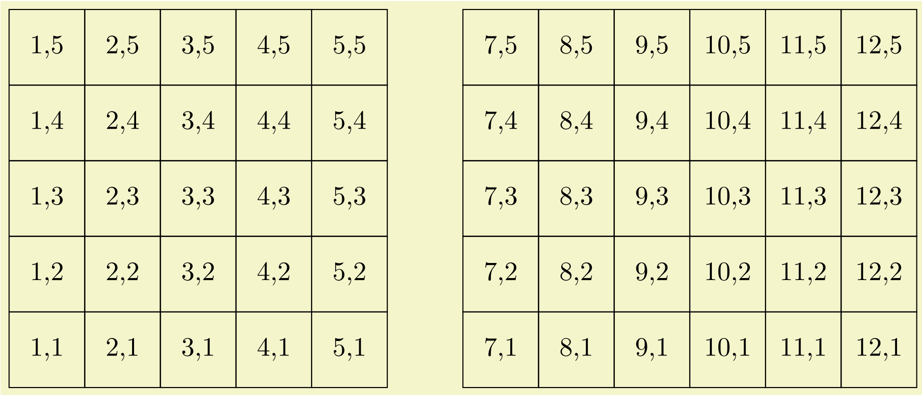


```
\tikz \foreach \x in {1,...,10}
        \draw (\x,0) circle (0.4cm);
```

If you provide *two* numbers before the `...`, the `\foreach` statement will use their difference for the stepping:


```
\tikz \foreach \x in {-1,-0.5,...,1}
        \draw (\x cm,-1pt) -- (\x cm,1pt);
```

We can also nest loops to create interesting effects:



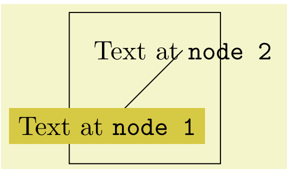

```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
    \foreach \y in {1,...,5}
    {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
    }
\end{tikzpicture}
```

The `\foreach` statement can do even trickier stuff, but the above gives the idea.

2.20 Adding Text

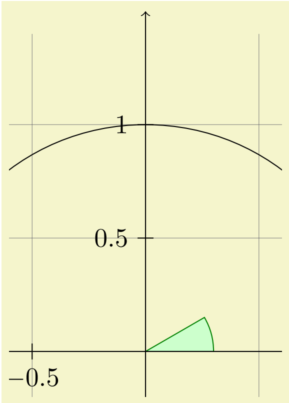
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

Ti*k*Z offers an easy-to-use and powerful system for adding text and, more generally, complex shapes to a picture at specific positions. The basic idea is the following: When Ti*k*Z is constructing a path and encounters the keyword `node` in the middle of a path, it reads a *node specification*. The keyword `node` is typically followed by some options and then some text between curly braces. This text is put inside a normal TEX box (if the node specification directly follows a coordinate, which is usually the case, Ti*k*Z is able to perform some magic so that it is even possible to use verbatim text inside the boxes) and then placed at the current position, that is, at the last specified position (possibly shifted a bit, according to the given options). However, all nodes are drawn only after the path has been completely drawn/filled/shaded/clipped/whatever.




```
\begin{tikzpicture}
  \draw (0,0) rectangle (2,2);
  \draw (0.5,0.5) node [fill=examplefill]
                       {Text at \verb!node 1!}
     -- (1.5,1.5) node {Text at \verb!node 2!};
\end{tikzpicture}
```

Obviously, Karl would not only like to place nodes *on* the last specified position, but also to the left or the right of these positions. For this, every node object that you put in your picture is equipped with several *anchors*. For example, the `north` anchor is in the middle at the upper end of the shape, the `south` anchor is at the bottom and the `north east` anchor is in the upper right corner. When you given the option `anchor=north`, the text will be placed such that this northern anchor will lie on the current position and the text is, thus, below the current position. Karl uses this to draw the ticks as follows:

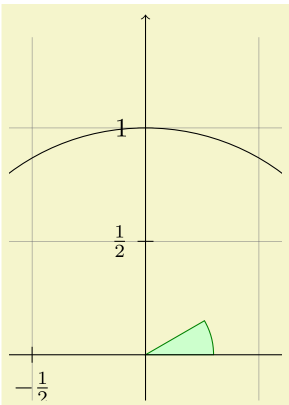



```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,style=help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);   \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);

  \foreach \x in {-1,-0.5,1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\x$};
  \foreach \y in {-1,-0.5,0.5,1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\y$};
\end{tikzpicture}
```

This is quite nice, already. Using these anchors, Karl can now add most of the other text elements. However, Karl thinks that, though "correct," it is quite counter-intuitive that in order to place something *below* a given point, he has to use the *north* anchor. For this reason, there is an option called `below`, which does the same as `anchor=north`. Similarly, `above right` does the same as `anchor=south east`. In addition, `below` takes an optional dimension argument. If given, the shape will additionally be shifted downwards by the given amount. So, `below=1pt` can be used to put a text label below some point and, additionally shift it 1pt downwards.

Karl is not quite satisfied with the ticks. He would like to have 1/2 or $\frac{1}{2}$ shown instead of 0.5, partly to show off the nice capabilities of TEX and Ti*k*Z, partly because for positions like 1/3 or π it is certainly very much preferable to have the "mathematical" tick there instead of just the "numeric" tick. His students, on the other hand, prefer 0.5 over 1/2 since they are not too fond of fractions in general.

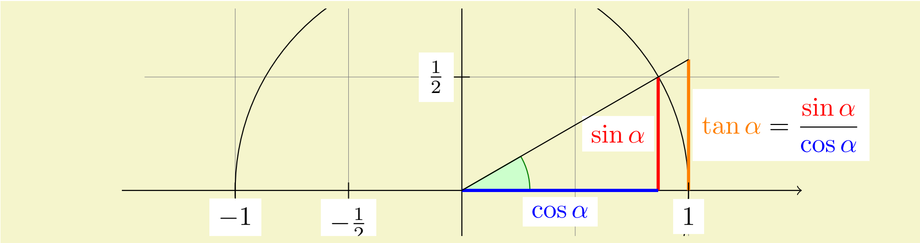
Karl now faces a problem: For the `\foreach` statement, the position `\x` should still be given as `0.5` since Ti*k*Z will not know where `\frac{1}{2}` is supposed to be. On the other hand, the typeset text should really be `\frac{1}{2}`. To solve this problem, `\foreach` offers a special syntax: Instead of having one variable `\x`, Karl can specify two (or even more) variables separated by a slash as in `\x / \xtext`. Then, the elements in the set over which `\foreach` iterates must also be of the form ⟨*first*⟩`/`⟨*second*⟩. In each iteration, `\x` will be set to ⟨*first*⟩ and `\xtext` will be set to ⟨*second*⟩. If no ⟨*second*⟩ is given, the ⟨*first*⟩ will be used again. So, here is the new code for the ticks:




```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,style=help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black]
    (0,0) -- (3mm,0mm) arc (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0); \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle (1cm);

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\ytext$};
\end{tikzpicture}
```

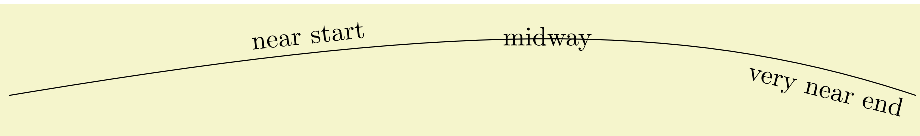
Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy solution: Karl can add the `[fill=white]` option to fill out the background of the text shape with a white color.

The next thing Karl wants to do is to add the labels like $\sin\alpha$. For this, he would like to place a label "in the middle of line." To do so, instead of specifying the label `node {$\sin\alpha$}` directly after one of the endpoints of the line (which would place the label at that endpoint), Karl can give the label directly after the `--`, before the coordinate. By default, this places the label in the middle of the line, but the `pos=` options can be used to modify this. Also, options like `near start` and `near end` can be used to modify this position:




```
\begin{tikzpicture}[scale=3]
  \clip (-2,-0.2) rectangle (2,0.8);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm) arc
  (0:30:3mm) -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0) coordinate (x axis);
  \draw[->] (0,-1.5) -- (0,1.5) coordinate (y axis);
  \draw (0,0) circle (1cm);

  \draw[very thick,red]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);
  \draw[very thick,blue]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);
  \draw[very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {$\displaystyle \tan \alpha \color{black}=
      \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$}
    (intersection of 0,0--30:1cm and 1,0--1,1) coordinate (t);

  \draw (0,0) -- (t);

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north,fill=white] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east,fill=white] {$\ytext$};
\end{tikzpicture}
```

You can also position labels on curves and, by adding the `sloped` option, have them rotated such that they match the line's slope. Here is an example:




```
\begin{tikzpicture}
  \draw (0,0) .. controls (6,1) and (9,1) ..
    node[near start,sloped,above] {near start}
    node {midway}
    node[very near end,sloped,below] {very near end} (12,0);
\end{tikzpicture}
```

It remains to draw the explanatory text at the right of the picture. The main difficulty here lies in limiting the width of the text "label," which is quite long, so that line breaking is used. Fortunately, Karl can use the option `text width=6cm` to get the desired effect. So, here is the full code:

```
\begin{tikzpicture}[scale=3,cap=round]
  % Local definitions
  \def\costhirty{0.8660256}

  % Colors
  \colorlet{anglecolor}{green!50!black}
  \colorlet{sincolor}{red}
  \colorlet{tancolor}{orange!80!black}
  \colorlet{coscolor}{blue}

  % Styles
  \tikzstyle{axes}=[]
  \tikzstyle{important line}=[very thick]
  \tikzstyle{information text}=[rounded corners,fill=red!10,inner sep=1ex]

  % The graphic
  \draw[style=help lines,step=0.5cm] (-1.4,-1.4) grid (1.4,1.4);

  \draw (0,0) circle (1cm);

  \begin{scope}[style=axes]
    \draw[->] (-1.5,0) -- (1.5,0) node[right] {$x$} coordinate(x axis);
    \draw[->] (0,-1.5) -- (0,1.5) node[above] {$y$} coordinate(y axis);

    \foreach \x/\xtext in {-1, -.5/-\frac{1}{2}, 1}
      \draw[xshift=\x cm] (0pt,1pt) -- (0pt,-1pt) node[below,fill=white] {$\xtext$};

    \foreach \y/\ytext in {-1, -.5/-\frac{1}{2}, .5/\frac{1}{2}, 1}
      \draw[yshift=\y cm] (1pt,0pt) -- (-1pt,0pt) node[left,fill=white] {$\ytext$};
  \end{scope}

  \filldraw[fill=green!20,draw=anglecolor] (0,0) -- (3mm,0pt) arc(0:30:3mm);
  \draw (15:2mm) node[anglecolor] {$\alpha$};

  \draw[style=important line,sincolor]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);

  \draw[style=important line,coscolor]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);

  \draw[style=important line,tancolor] (1,0) -- node[right=1pt,fill=white] {
    $\displaystyle \tan \alpha \color{black}=
    \frac{{\color{sincolor}\sin \alpha}}{\color{coscolor}\cos \alpha}$}
    (intersection of 0,0--30:1cm and 1,0--1,1) coordinate (t);

  \draw (0,0) -- (t);

  \draw[xshift=1.85cm]
    node[right,text width=6cm,style=information text]
    {
      The {\color{anglecolor} angle $\alpha$} is $30^\circ$ in the
      example ($\pi/6$ in radians). The {\color{sincolor}sine of
        $\alpha$}, which is the height of the red line, is
      \[
      {\color{sincolor} \sin \alpha} = 1/2.
      \]
      By the Theorem of Pythagoras ...
    };
\end{tikzpicture}
```

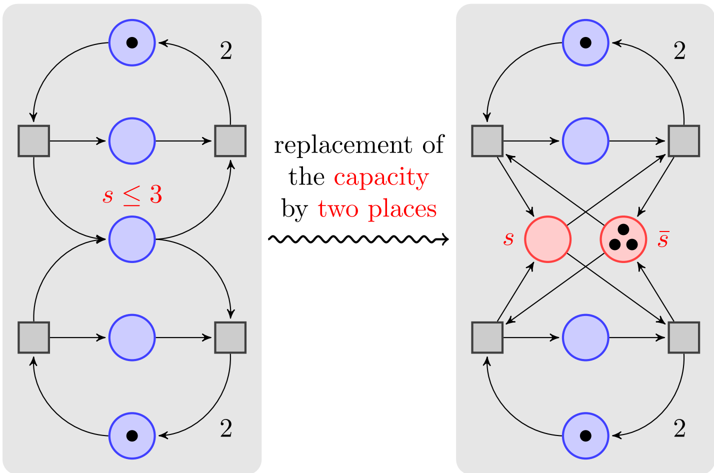
3 Tutorial: A Petri-Net for Hagen

In this second tutorial we explore the node mechanism of Ti*k*Z and PGF.

Hagen must give a talk tomorrow about his favorite formalism for distributed systems: Petri nets! Hagen used to give his talks using a blackboard and everyone seemed to be perfectly concent with this. Unfortunately, his audience has been spoiled recently with fancy projector-based presentations and there seems to be a certain amount of peer pressure that this Petri nets should also be drawn using a graphic program. One of the professors at his institutes recommends Ti*k*Z for this and Hagen decides to give it a try.

3.1 Problem Statement

For his talk, Hagen wishes to create a graphic that demonstrates how a net with place capacities can be simulated by a net without capacities. The graphic should look like this, ideally:



3.2 Setting up the Environment

For the picture Hagen will need to load the Ti*k*Z package as did Karl in the previous tutorial. However, Hagen will also need to load some additional *library packages* that Karl did not need. These library packages contain additional definitions like extra arrow tips that are typically not needed in a picture and that need to be loaded explicitly.

Hagen will need to load three libraries: The arrow tip library for the special arrow tip used in the graphic, the snake library with the "snaking line" in the middle, and the background library for the two rectangular areas that are behind the two main parts of the picture.

3.2.1 Setting up the Environment in LaTeX

When using LaTeX use:

```
\documentclass{article} % say

\usepackage{tikz}
\usetikzlibrary{arrows,snakes,backgrounds}

\begin{document}
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```

3.2.2 Setting up the Environment in Plain TeX

When using plain TeX use:

```
%% Plain TeX file
\input tikz.tex
\usetikzlibrary{arrows,snakes,backgrounds}
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
\tikzpicture
  \draw (0,0) -- (1,1);
\endtikzpicture
\bye
```

3.2.3 Setting up the Environment in ConTEXt

When using ConTEX use:

```
%% ConTeXt file
\usemodule[tikz]
\usetikzlibrary[arrows,snakes,backgrounds]

\starttext
  \starttikzpicture
    \draw (0,0) -- (1,1);
  \stoptikzpicture
\startext
```

3.3 Introduction to Nodes

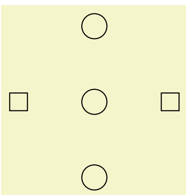
In principle, we already know how to create the graphics that Hagen desires (except perhaps for the snaked line, we will come to that): We start with big light gray rectangle and then add lots of circles and small rectangle, plus some arrows.

However, this approach has numerous disadvantages: First, it is hard to change anything at a later stage. For example, if we decide to add more places to the Petri nets (the circles are called places in Petri net theory), all of the coordinates change and we need to recalculate everything. Second, it is hard to read the code for the Petri net as it just a long and complicated list of coordinates and drawing commands – the underlying structure of the Petri net is lost.

Fortunately, Ti*k*Z offers a powerful mechanism for avoiding the above problems: nodes. We already came across nodes in the previous tutorial, where we used them to add labels to Karl's graphic. In the present tutorial we will see that nodes are much more powerful.

A node is a small part of a picture. When a node is created, you provide a position where the node should be drawn and a *shape*. A node of shape `circle` will be drawn as a circle, a node of shape `rectangle` as a rectangle, and so on. A node may also contain same text, which is why Karl used nodes to show text. Finally, a node can get a *name* for later reference.

In Hagen's picture we will use nodes for the places and for the transitions of the Petri net (the places are the circles, the transitions are the rectangles). Let us start with the upper half of the left Petir net. In this upper half we have three places and two transitions. Instead of drawing three circles and two rectangles, we use three nodes of shape `circle` and two nodes of shape `rectangle`.



```
\begin{tikzpicture}
  \path ( 0,2) node [shape=circle,draw] {}
        ( 0,1) node [shape=circle,draw] {}
        ( 0,0) node [shape=circle,draw] {}
        ( 1,1) node [shape=rectangle,draw] {}
        (-1,1) node [shape=rectangle,draw] {};
\end{tikzpicture}
```

Hagen notes that this does not quite look like the final picture, but it seems like a good first step.

Let us have a more detailed look at the code. The whole picture consists of a single path. Ignoring the `node` operations there is not much going on in this path: It is just a sequence of coordinates with nothing "happening" between them. Indeed, even if something were to happen like a line-to or a curve-to, the `\path` command would not "do" anything with the resulting path. So, all the magic must be in the `node` commands.

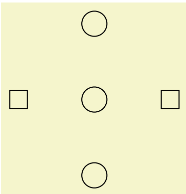
In the previous tutorial we learned that a `node` will add a piece of text at the last coordinate. Thus, each of the five nodes is added at a different position. In the above code, this text is empty (because of the empty `{}`). So, why do we see anything at all? The answer is the `draw` option for the `node` operation: It causes the "shape around the text" to be drawn.

So, the code `(0,2) node [shape=circle,draw] {}` means the following: "In the main path, add a move-to to the coordinate `(0,2)`. Then, temporarily suspend the construction of the main path while the node is build. This node will be a `circle` around an empty text. This circle is to be `drawn`, but not filled or otherwise used. Once this whole node is constructed, it is saved until after the main path is finished. Then, it is drawn." Then following `(0,1) node [shape=circle,draw] {}` then has the following effect: "Continue the main path with a move-to to `(0,1)`. Then construct a node at this position also. This node is also shown after the main path is finished." And so on.

3.4 Placing Nodes Using the At Syntax

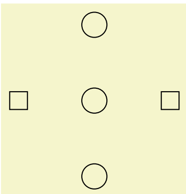
Hagen now understands how the `node` operation adds nodes to the path, but it seems a bit silly to create a path using the `\path` operation, consisting of numerous superfluous move-to operations, only to place nodes. He is pleased to learn that there are ways to add nodes in a more sensible manner.

First, the `node` operation allows one to add `at (`⟨*coordinate*⟩`)` in order to directly specify where the node should be placed, sidestepping the rule that nodes are placed on the last coordinate. Hagen can then write the following:



```
\begin{tikzpicture}
  \path node at ( 0,2) [shape=circle,draw] {}
        node at ( 0,1) [shape=circle,draw] {}
        node at ( 0,0) [shape=circle,draw] {}
        node at ( 1,1) [shape=rectangle,draw] {}
        node at (-1,1) [shape=rectangle,draw] {};
\end{tikzpicture}
```

Now Hagen is still left with a single empty path, but at least the path no longer contains strange move-tos. It turns out that this can be improved further: The `\node` command is an abbreviation for `\path node`, which allows Hagen to write:

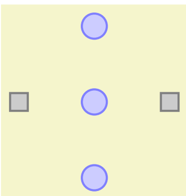


```
\begin{tikzpicture}
  \node at ( 0,2) [circle,draw] {};
  \node at ( 0,1) [circle,draw] {};
  \node at ( 0,0) [circle,draw] {};
  \node at ( 1,1) [rectangle,draw] {};
  \node at (-1,1) [rectangle,draw] {};
\end{tikzpicture}
```

Hagen likes this syntax much better than the previous one. Note that Hagen has also omitted the `shape=` since, like `color=`, Ti*k*Z allows you to omit the `shape=` if there is no confusion.

3.5 Using Styles

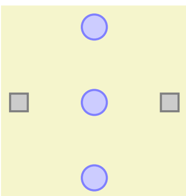
Feeling adventurous, Hagen tries to make the nodes look nicer. In the final picture, the circles and rectangle should be filled with different colors, resulting in the following code:



```
\begin{tikzpicture}[thick]
  \node at ( 0,2) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 0,1) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 0,0) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 1,1) [rectangle,draw=black!50,fill=black!20] {};
  \node at (-1,1) [rectangle,draw=black!50,fill=black!20] {};
\end{tikzpicture}
```

While this looks nicer in the picture, the code starts to get a bit ugly. Ideally, we would like our code to transport the message "there are three places and two transitions" and not so much which filling colors should be used.

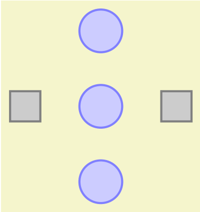
To solve this problem, Hagen uses styles. He defines a style for places and another style for transitions:



```
\tikzstyle{place}=[circle,draw=blue!50,fill=blue!20,thick]
\tikzstyle{transition}=[rectangle,draw=black!50,fill=black!20,thick]
\begin{tikzpicture}
  \node at ( 0,2) [place] {};
  \node at ( 0,1) [place] {};
  \node at ( 0,0) [place] {};
  \node at ( 1,1) [transition] {};
  \node at (-1,1) [transition] {};
\end{tikzpicture}
```

3.6 Node Size

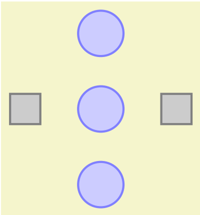
Before Hagen starts naming and connecting the nodes, let us first make sure that the nodes get their final appearance. They are still too small. Indeed, Hagen wonders why they have any size at all, after all, the text is empty. The reason is than Ti*k*Z automatically adds some space around the text. The amount is set using the option `inner sep`. So, to increase the size of the nodes, Hagen could write:



```
\tikzstyle{place}=[circle,draw=blue!50,fill=blue!20,thick]
\tikzstyle{transition}=[rectangle,draw=black!50,fill=black!20,thick]
\begin{tikzpicture}[inner sep=2mm]
  \node at ( 0,2) [place] {};
  \node at ( 0,1) [place] {};
  \node at ( 0,0) [place] {};
  \node at ( 1,1) [transition] {};
  \node at (-1,1) [transition] {};
\end{tikzpicture}
```

However, this is not really the best way to achieve the desired effect. It is much better to use the `minimum size` option instead. This option allows Hagen to specify a minimum size that the node should have. If the nodes actually needs to be bigger because of a longer text, it will be larger, but if the text is empty, then the node will have `minimum size`. This option is also useful to ensure that several nodes containing different amounts of text have the same size. The options `minimum height` and `minimum width` allow you to specify the minimum height and width independently.

So, what Hagen needs to do is to provide `minimum size` for the nodes. To be on the safe side, he also sets `inner sep=0pt`. This ensures that the nodes will really have size `minimum size` and not, for very small minimum sizes, the minimal size necessary to encompass the automatically added space.



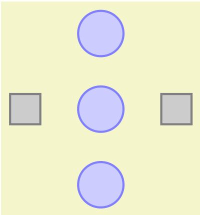
```
\tikzstyle{place}=[circle,draw=blue!50,fill=blue!20,thick,
                   inner sep=0pt,minimum size=6mm]
\tikzstyle{transition}=[rectangle,draw=black!50,fill=black!20,thick,
                        inner sep=0pt,minimum size=4mm]
\begin{tikzpicture}
  \node at ( 0,2) [place] {};
  \node at ( 0,1) [place] {};
  \node at ( 0,0) [place] {};
  \node at ( 1,1) [transition] {};
  \node at (-1,1) [transition] {};
\end{tikzpicture}
```

3.7 Naming Nodes

Hagen's next aim is to connect the nodes using arrows. This seems like a tricky business since the arrows should not start in the middle of the nodes, but somewhere on the border and Hagen would very much like to avoid computing these positions by hand.

Fortunately, PGF will perform all the necessary calculations for him. However, he first has to assign names to the nodes so that he can reference them later on.

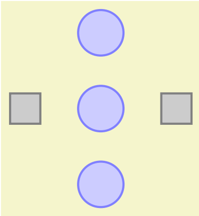
There are two ways to name a node. The first is the use the `name=` option. The second method is to write the desired name in parentheses after the `node` operation. Hagen thinks that this second method seems strange, but he will soon change his opinion.



```
% ... setup styles
\begin{tikzpicture}
  \node (waiting 1)  at ( 0,2)     [place] {};
  \node (critical 1) at ( 0,1)     [place] {};
  \node (semaphore)  at ( 0,0)     [place] {};
  \node (leave critical) at ( 1,1) [transition] {};
  \node (enter critical) at (-1,1) [transition] {};
\end{tikzpicture}
```

Hagen is pleased to note that the names help in understanding the code. Names for nodes can be pretty arbitrary, but they should not contain commas, periods, parentheses, colons, and some other special characters. However, they can contain underscores and hyphens.

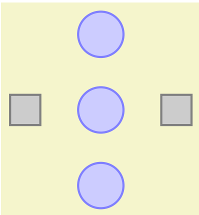
The syntax for the `node` operation is quite liberal with respect to the order in which node names, the `at` specifier, and the options must come. Indeed, you can even have multiple option blocks between the `node` and the text in curly braces, they accumulate. You can rearrange them arbitrarily and perhaps the following might be preferable:



```
\begin{tikzpicture}
  \node[place]      (waiting 1)      at ( 0,2) {};
  \node[place]      (critical 1)     at ( 0,1) {};
  \node[place]      (semaphore)      at ( 0,0) {};
  \node[transition] (leave critical) at ( 1,1) {};
  \node[transition] (enter critical) at (-1,1) {};
\end{tikzpicture}
```

3.8 Placing Nodes Using Relative Placement

Although Hagen still wishes to connect the nodes, he first wishes to address another problem again: The placement of the nodes. Although he likes the `at` syntax, in this particular case he would prefer placing the nodes "relative to each other." So, Hagen would like to say that the `critical 1` node should be below the `waiting 1` node, wherever the `waiting 1` node might be. There are different ways of achieving this, but the nicest one in Hagen's case is the `below of` option:



```
\begin{tikzpicture}
  \node[place]      (waiting)                            {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
\end{tikzpicture}
```

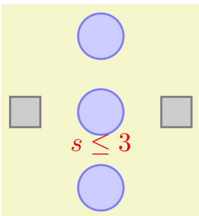
The `below of` and similar options setup the position of the node in such a manner that it is placed at the distance `node distance` in the specified direction of the given direction. The `node distance` is the distance between the centers of the nodes, not between the borders.

Even though the above code has the same effect the earlier code, Hagen can pass it to his colleagues how will be able to just read and understand it, perhaps without even having to see the picture.

3.9 Adding Labels Next to Nodes

Before we have a look at how Hagen can connect the nodes, let us add the capacity "$s \le 3$" to the bottom node. For this, two approaches are possible:

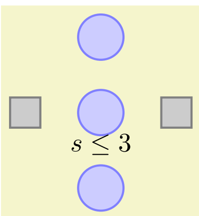
1. Hagen can just add a new node above the `north` anchor of the `semaphore` node.




```
\begin{tikzpicture}
  \node[place]      (waiting)                            {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};

  \node [red,above] at (semaphore.north) {$s\le 3$};
\end{tikzpicture}
```

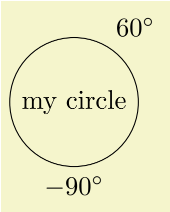
This is a general approach that will "always work."

2. Hagen can use the special `label` option. This option is given to a `node` and it causes *another* node to be added next to the node where the option is given. Here is the idea: When we construct the `semaphore` node, we wish to indicate that we want another node with the capacity above it. For this, we use the option `label=above:$s\le 3$`. This option is interpreted as follows: We want a node above the `semaphore` node and this node should read "$s \le 3$." Instead of `above` we could also use things like `below left` before the colon or a number like `60`.



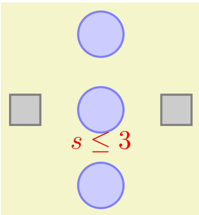

```
\begin{tikzpicture}
  \node[place]      (waiting)                            {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical,
                                      label=above:$s\le3$] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
\end{tikzpicture}
```

It is also possible to give multiple `label` options, this causes multiple labels to be drawn.




```
\tikz
  \node [circle,draw,label=60:$60^\circ$,label=below:$-90^\circ$] {my circle};
```

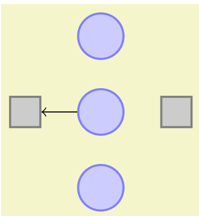
Hagen is not fully satisfied with the `label` option since the label is not red. To achieve this, has has two options: First, he can redefine the `every label` style. Second, he can add options to the label's node. These options are given following the `label=`, so he would write `label=[red]above:$s\le3$`. However, this does not quite work since TeX thinks that the `]` closes the whole option list of the `semaphore` node. So, Hagen has to add braces and writes `label={[red]above:$s\le3$}`. Since this looks a bit ugly, Hagen decides to redefine the `every label` style.




```
\begin{tikzpicture}
  \tikzstyle{every label}=[red]
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical,
                                      label=above:$s\le3$] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
\end{tikzpicture}
```

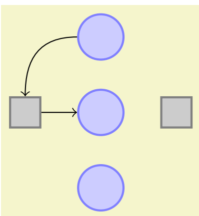
3.10 Connecting Nodes

It is now high time to connect the nodes. Let us start with something simple, namely with the straight line from `enter critical` to `critical`. We want this line to start at the right side of `enter critical` and to end at the left side of `critical`. For this, we can use the *anchors* of the nodes. Every node defines a whole bunch of anchors that lie on its border or inside it. For example, the `center` anchor is at the center of the node, the `west` anchor is on the left of the node, and so on. To access the coordinate of a node, we use a coordinate that contains the node's name followed by a dot, followed by the anchor's name:



```
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
  \draw [->] (critical.west) -- (enter critical.east);
\end{tikzpicture}
```

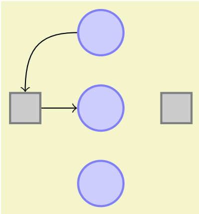
Next, let us tackle the curve from `waiting` to `enter critical`. This can be specified using curves and controls:



```
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
  \draw [->] (enter critical.east) -- (critical.west);
  \draw [->] (waiting.west) .. controls +(left:5mm) and +(up:5mm)
                            .. (enter critical.north);
\end{tikzpicture}
```

Hagen sees how he can now add all his edges, but the whole process seems a but awkward and not very flexible. Again, the code seems to obscure the structure of the graphic rather than showing it.

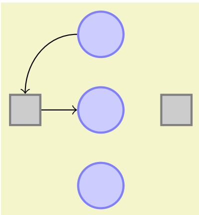
So, let us start improving the code for the edges. First, Hagen can leave out the anchors:



```
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
  \draw [->] (enter critical) -- (critical);
  \draw [->] (waiting) .. controls +(left:8mm) and +(up:8mm)
                       .. (enter critical);
\end{tikzpicture}
```

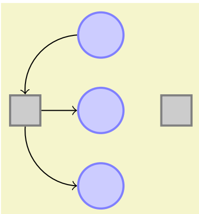
Hagen is a bit surprised that this works. After all, how did Ti*k*Z know that the line from `enter critical` to `critical` should actually start on the borders? Whenever Ti*k*Z encounters a whole node name as a "coordinate," it tries to "be smart" about the anchor that it should choose for this node. Depending on what happens next, Ti*k*Z will choose an anchor that lies on the border of the node on a line to the next coordinate or control point. The exact rules are a bit complex, but the chosen point will usually be correct – and when it is not, Hagen can still specify the desired anchor by hand.

Hagen would now like to simplify the curve operation somehow. It turns out that this can be accomplished using a special path operation: the `to` operation. This operation takes many options (you can even define new ones yourself). One pair of option is useful for Hagen: The pair `in` and `out`. These options take angles at which a curve should leave or reach the start or target coordinates. Without these options, a straight line is drawn:



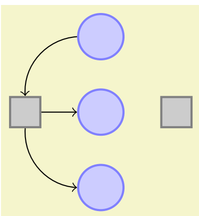
```
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
  \draw [->] (enter critical) to                  (critical);
  \draw [->] (waiting)        to [out=180,in=90] (enter critical);
\end{tikzpicture}
```

There is another option for the `to` operation, that is even better suited to Hagen's problem: The `bend right` option. This option also takes an angle, but this angle only specifies the angle by which the curve is bend to the right:



```
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {};
  \draw [->] (enter critical) to                  (critical);
  \draw [->] (waiting)        to [bend right=45] (enter critical);
  \draw [->] (enter critical) to [bend right=45] (semaphore);
\end{tikzpicture}
```

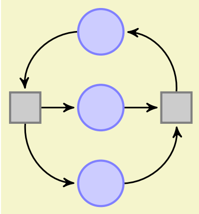
It is now time for Hagen to learn about yet another way of specifying edges: Using the `edge` path operation. This operation is very similar to the `to` operation, but there is one important difference: Like a node the edge generated by the `edge` operation is not part of the main path, but is added only later. This may not seem very important, but is has some nice consequences. For example, every edge can have its own arrow tips and its own color and so one and, still, all the edges can be given on the same path. This allows Hagen to write the following:



```
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};
  \node[transition] (leave critical) [right of=critical] {};
  \node[transition] (enter critical) [left of=critical]  {}
    edge [->]               (critical)
    edge [<-,bend left=45]  (waiting)
    edge [->,bend right=45] (semaphore);
\end{tikzpicture}
```

Each `edge` caused a new path to be constructed, consisting of a `to` between the node `enter critical` and the node following the `edge` command.

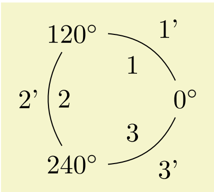
The finishing touch is to introduce two styles `pre` and `post` and to use the `bend angle=45` option to set the bend angle once and for all:



```
% Styles place and transition as before
\tikzstyle{pre}=[<-,shorten <=1pt,>=stealth',semithick]
\tikzstyle{post}=[->,shorten >=1pt,>=stealth',semithick]
\begin{tikzpicture}[bend angle=45]
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};

  \node[transition] (leave critical) [right of=critical] {}
    edge [pre]              (critical)
    edge [post,bend right]  (waiting)
    edge [pre, bend left]   (semaphore);
  \node[transition] (enter critical) [left of=critical]  {}
    edge [post]             (critical)
    edge [pre, bend left]   (waiting)
    edge [post,bend right]  (semaphore);
\end{tikzpicture}
```

3.11 Adding Labels Next to Lines

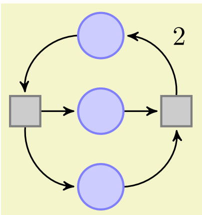
The next thing that Hagen needs to add is the “2” at the arcs. For this Hagen can use Ti*k*Z's automatic node placement: By adding the option `auto`, Ti*k*Z will position nodes on curves and lines in such a way that they are not on the curve but next to it. Adding `swap` will mirror the label with respect to the line. Here is a general example:




```
\begin{tikzpicture}[auto,bend right]
  \node (a) at (0:1) {$0^\circ$};
  \node (b) at (120:1) {$120^\circ$};
  \node (c) at (240:1) {$240^\circ$};

  \draw (a) to node {1} node [swap] {1'} (b)
        (b) to node {2} node [swap] {2'} (c)
        (c) to node {3} node [swap] {3'} (a);
\end{tikzpicture}
```

What is happening here? The nodes are given somehow inside the `to` operation! When this is done, the node is placed on the middle of the curve or line created by the `to` operation. The `auto` option then causes the node to be moved in such a way that it does not lie on the curve, but next to it. In the example we provide even two nodes on each `to` operation.

For Hagen that `auto` option is not really necessary since the two “2” labels could also easily be placed “by hand.” However, in a complicated plot with numerous edges automatic placement can be a blessing.



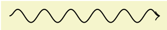

```
% Styles as before
\begin{tikzpicture}[bend angle=45]
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};

  \node[transition] (leave critical) [right of=critical] {}
    edge [pre]                               (critical)
    edge [post,bend right] node[auto,swap] {2} (waiting)
    edge [pre, bend left]                    (semaphore);
  \node[transition] (enter critical) [left of=critical]  {}
    edge [post]                              (critical)
    edge [pre, bend left]                    (waiting)
    edge [post,bend right]                   (semaphore);
\end{tikzpicture}
```

3.12 Adding the Snaked Line and Multi-Line Text

With the node mechanism Hagen can now easily create the two Petri nets. What he is unsure of is how he can create the snaked line between the nets.

For this he can use a *snake*. Snakes a called thus since the most basic form of a snake looks exactly like a snake. However, and repeating pattern can be used as a snake like bumps or a saw or even much more complicated stuff.

To draw the snake, Hagen only needs to set the `snake=snake` option on the path. This causes all straight lines of the path to be replaced by snakes. It is also possible to use snakes only in certain parts of a path, but Hagen will not need this.

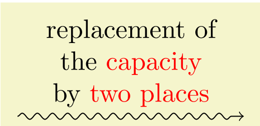


```
\begin{tikzpicture}
  \draw [->,snake=snake] (0,0) -- (2,0);
\end{tikzpicture}
```

Well, that does not look quite right, yet. The problem is that the snake happens to end exactly at the position where the arrow begins. Fortunately, there is an option that helps here. Also, the snake should be a bit smaller, which can be influenced by even more options.


```
\begin{tikzpicture}
  \draw [->,snake=snake,
         segment amplitude=.4mm,
         segment length=2mm,
         line after snake=1mm] (0,0) -- (3,0);
\end{tikzpicture}
```

Now Hagen needs to add the text above the snake. This text is a bit challenging since it is a multi-line text. To typeset such text, Hagen needs to specify a width for the text and he needs to specify that the text should be centered.

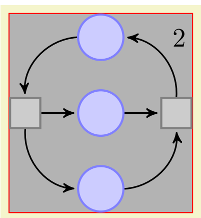



```
\begin{tikzpicture}
  \draw [->,snake=snake,
         segment amplitude=.4mm,
         segment length=2mm,
         line after snake=1mm] (0,0) -- (3,0)
    node [above,text width=3cm,text centered,midway]
    {
      replacement of the \textcolor{red}{capacity} by
      \textcolor{red}{two places}
    };
\end{tikzpicture}
```

3.13 Using Layers: The Background Rectangles

Hagen still needs to add the background rectangles. These are a bit tricky: Hagen would like to draw the rectangles *after* the Petri nets are finished. The reason is that only then can he conveniently refer to the coordinates that make up the corners of the rectangle. If Hagen draws the rectangle first, then he needs to know the exact size of the Petri net – which he does not.

The solution is to use *layers*. When the background library is loaded, Hagen can put parts of his picture inside a `{pgfonlayer}` environment. Then this part of the picture becomes part of the layer that is given as an argument to this environment. When the `{tikzpicture}` environment ends, the layers are put on top of each other, starting with the background layer. This causes everything drawn on the background layer to be behind the main text.




```
% Styles as before
\begin{tikzpicture}[bend angle=45]
  \node[place]      (waiting)                            {};
  \node[place]      (critical)       [below of=waiting]  {};
  \node[place]      (semaphore)      [below of=critical] {};

  \node[transition] (leave critical) [right of=critical] {}
    edge [pre]                                  (critical)
    edge [post,bend right] node[auto,swap] {2} (waiting)
    edge [pre, bend left]                       (semaphore);
  \node[transition] (enter critical) [left of=critical]  {}
    edge [post]                                 (critical)
    edge [pre, bend left]                       (waiting)
    edge [post,bend right]                      (semaphore);

  \begin{pgfonlayer}{background}
    \filldraw [fill=black!30,draw=red]
              (semaphore.south -| enter critical.west)
      rectangle (waiting.north   -| leave critical.east);
  \end{pgfonlayer}
\end{tikzpicture}
```

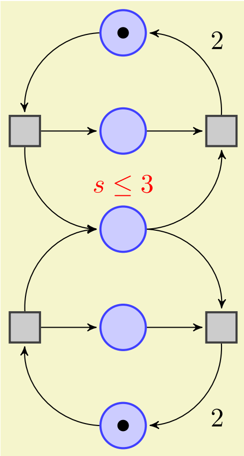
3.14 The Complete Code

Hagen has now finally put everything together. Only then does he learn that there is already a library for drawing Petri nets! It turns out that this library mainly provides the same definitions as Hagen did. For example, it defines a `place` style in a similar way as Hagen did. Adjusting the code so that it uses the library shortens Hagen code a bit, as shown in the following.

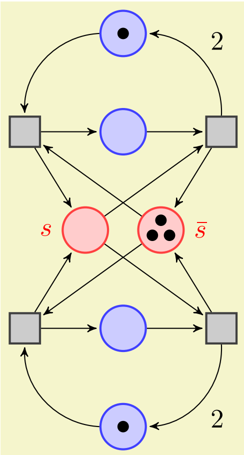
First, Hagen needs less style definitions, but he still needs to specify the colors of places and transitions.

```
\tikzstyle{every place}=     [minimum size=6mm,thick,draw=blue!75,fill=blue!20]
\tikzstyle{every transition}=[thick,draw=black!75,fill=black!20]

\tikzstyle{red place}=       [place,draw=red!75,fill=red!20]

\tikzstyle{every label}=     [red]
\begin{tikzpicture}[node distance=1.3cm,>=stealth',bend angle=45,auto]
```

Now comes the code for the nets:




```
\node [place,tokens=1] (w1)                                    {};
\node [place]          (c1) [below of=w1]                      {};
\node [place]          (s)  [below of=c1,label=above:$s\le 3$] {};
\node [place]          (c2) [below of=s]                       {};
\node [place,tokens=1] (w2) [below of=c2]                      {};

\node [transition] (e1) [left of=c1] {}
  edge [pre,bend left]                  (w1)
  edge [post,bend right]                (s)
  edge [post]                           (c1);
\node [transition] (e2) [left of=c2] {}
  edge [pre,bend right]                 (w2)
  edge [post,bend left]                 (s)
  edge [post]                           (c2);
\node [transition] (l1) [right of=c1] {}
  edge [pre]                            (c1)
  edge [pre,bend left]                  (s)
  edge [post,bend right] node[swap] {2} (w1);
\node [transition] (l2) [right of=c2] {}
  edge [pre]                            (c2)
  edge [pre,bend right]                 (s)
  edge [post,bend left]  node {2}       (w2);
```




```
\begin{scope}[xshift=6cm]
  \node [place,tokens=1]     (w1')                              {};
  \node [place]              (c1') [below of=w1']               {};
  \node [red place]          (s1') [below of=c1',xshift=-5mm]
          [label=left:$s$]                                      {};
  \node [red place,tokens=3] (s2') [below of=c1',xshift=5mm]
          [label=right:$\bar s$]                                {};
  \node [place]              (c2') [below of=s1',xshift=5mm]    {};
  \node [place,tokens=1]     (w2') [below of=c2']               {};

  \node [transition] (e1') [left of=c1'] {}
    edge [pre,bend left]                  (w1')
    edge [post]                           (s1')
    edge [pre]                            (s2')
    edge [post]                           (c1');
  \node [transition] (e2') [left of=c2'] {}
    edge [pre,bend right]                 (w2')
    edge [post]                           (s1')
    edge [pre]                            (s2')
    edge [post]                           (c2');
  \node [transition] (l1') [right of=c1'] {}
    edge [pre]                            (c1')
    edge [pre]                            (s1')
    edge [post]                           (s2')
    edge [post,bend right] node[swap] {2} (w1');
  \node [transition] (l2') [right of=c2'] {}
    edge [pre]                            (c2')
    edge [pre]                            (s1')
    edge [post]                           (s2')
    edge [post,bend left]  node {2}       (w2');
\end{scope}
```

The code for the background and the snake is the following:

```
  \draw [-to,thick,snake=snake,segment amplitude=.4mm,segment length=2mm,line after snake=1mm]
    ([xshift=5mm]s -| l1) -- ([xshift=-5mm]s1' -| e1')
    node [above=1mm,midway,text width=3cm,text centered]
      {replacement of the \textcolor{red}{capacity} by \textcolor{red}{two places}};

  \begin{pgfonlayer}{background}
    \filldraw [line width=4mm,join=round,black!10]
      (w1.north  -| l1.east)  rectangle (w2.south  -| e1.west)
      (w1'.north -| l1'.east) rectangle (w2'.south -| e1'.west);
  \end{pgfonlayer}
\end{tikzpicture}
```

4 Guidelines on Graphics

The present section is not about PGF or TikZ, but about general guidelines and principles concerning the creation of graphics for scientific presentations, papers, and books.

The guidelines in this section come from different sources. Many of them are just what I would like to claim is "common sense," some reflect my personal experience (though, hopefully, not my personal preferences), some come from books (the bibliography is still missing, sorry) on graphic design and typography. The most influential source are the brilliant books by Edward Tufte. While I do not agree with everything written in these books, many of Tufte's arguments are so convincing that I decided to repeat them in the following guidelines.

The first thing you should ask yourself when someone presents a bunch of guidelines is: Should I really follow these guidelines? This is an important questions, because there are good reasons not to follow general guidelines.

- The person who setup the guidelines may have had other objectives than you do. For example, a guideline might say "use the color red for emphasis." While this guideline makes perfect sense for, say, a presentation using a projector, red "color" has the *opposite* effect of "emphasis" when printed using a black-and-white printer.

  Guidelines were almost always setup to address a specific situation. If you are not in this situation, following a guideline can do more harm than good.

- The basic rule of typography is: "Every rule can be broken, as long as you are *aware* that you are breaking a rule." This rule also applies to graphics. Phrased differently, the basic rule states: "The only mistakes in typography are things done is ignorance."

  When you are aware of a rule and when you decide that breaking the rule has a desirable effect, break the rule.

So, before you apply a guideline or choose not to apply it, ask yourself these questions:

1. Does this guideline really address my situation?
2. If you do the opposite a guideline says you should do, will the advantages outweigh the disadvantages this guideline was supposed to prevent?

4.1 Planning the Time Needed for the Creation of Graphics

When you create a paper with numerous graphics, the time needed to create these graphics becomes an important factor. How much time should you calculate for the creation of graphics?

As a general rule, assume that a graphic will need as much time to create as would a text of the same length. For example, when I write a paper, I need about one hour per page for the first draft. Later, I need between two and four hours per page for revisions. Thus, I expect to need about half an hour for the creation of *a first draft* of a half page graphic. Later on, I expect another one to two hours before the final graphic is finished.

In many publications, even in good journals, the authors and editors have obviously invested a lot of time on the text, but seem to have spend about five minutes to create all of the graphics. Graphics often seem to have been added as an "afterthought" or look like a screen shot of whatever the authors's statistical software shows them. As will be argued later on, the graphics that programs like GNUPLOT produce by default are of poor quality.

Creating informative graphics that help the reader and that fit together with the main text is a difficult, lengthy process.

- Treat graphics as first-class citizens of your papers. They deserve as much time and energy as the text does.
- Arguably, the creation of graphics deserves *even more* time than the writing of the main text since more attention will be paid to the graphics and they will be looked at first.
- Plan as much time for the creation and revision of a graphic as you would plan for text of the same size.
- Difficult graphics with a high information density may require even more time.

- Very simple graphics will require less time, but most likely you do not want to have "very simple graphics" in your paper, anyway; just as you would not like to have a "very simple text" of the same size.

4.2 Workflow for Creating a Graphic

When you write a (scientific) paper, you will most likely follow the following pattern: You have some results/ideas that you would like to report about. The creation of the paper will typically start with compiling a rough outline. Then, the different sections are filled with text to create a first draft. This draft is then revised repeatedly until, often after substantial revision, a final paper results. In a good journal paper there is typically not be a single sentence that has survived unmodified from the first draft.

Creating a graphics follows the same pattern:

- Decide on what the graphic should communicate. Make this a conscious decision, that is, determine "What is the graphic supposed to tell the reader?"
- Create an "outline," that is, the rough overall "shape" of the graphic, containing the most crucial elements. Often, it is useful to do this using pencil and paper.
- Fill out the finer details of the graphic to create a first draft.
- Revise the graphic repeatedly along with the rest of the paper.

4.3 Linking Graphics With the Main Text

Graphics can be placed at different places in a text. Either, they can be inlined, meaning they are somewhere "in the middle of the text" or they can be placed in standalone "figures." Since printers (the people) like to have their pages "filled," (both for aesthetic and economic reasons) standalone figures may traditionally be placed on pages in the document far removed from the main text that refers to them. LaTeX and TeX tend to encourage this "drifting away" of graphics for technical reasons.

When a graphic is inlined, it will more or less automatically be linked with the main text in the sense that the labels of the graphic will be implicitly explained by the surrounding text. Also, the main text will typically make it clear what the graphic is about and what is shown.

Quite differently, a standalone figure will often be viewed at a time when the main text that this graphic belongs to either has not yet been read or has been read some time ago. For this reason, you should follow the following guidelines when creating standalone figures:

- Standalone figures should have a caption than should make them "understandable by themselves."

  For example, suppose a graphic shows an example of the different stages of a quicksort algorithm. Then the figure's caption should, at the very least, inform the reader that "The figure shows the different stages of the quicksort algorithm introduced on page xyz." and not just "Quicksort algorithm."

- A good caption adds as much context information as possible. For example, you could say: "The figure shows the different stages of the quicksort algorithm introduced on page xyz. In the first line, the pivot element 5 is chosen. This causes..." While this information can also be given in the main text, putting it in the caption will ensure that the context is kept. Do not feel afraid of a 5-line caption. (Your editor may hate you for this. Consider hating them back.)

- Reference the graphic in your main text as in "For an example of quicksort 'in action,' see Figure 2.1 on page xyz."

- Most books on style and typography recommend that you do not use abbreviations as in "Fig. 2.1" but write "Figure 2.1."

  The main argument against abbreviations is that "a period is too valuable to waste it on an abbreviation." The idea is that a period will make the reader assume that the sentence ends after "Fig" and it takes a "conscious backtracking" to realize that the sentence did not end after all.

  The argument in favor of abbreviations is that they save space.

  Personally, I am not really convinced by either argument. On the one hand, I have not yet seen any hard evidence that abbreviations slow readers down. On the other hand, abbreviating all "Figure" by "Fig." is most unlikely to save even a single line in most documents.

  I avoid abbreviations.

4.4 Consistency Between Graphics and Text

Perhaps the most common "mistake" people do when creating graphics (remember that a "mistake" in design is always just "ignorance") is to have a mismatch between the way their graphics look and the way their text looks.

It is quite common that authors use several different programs for creating the graphics of a paper. An author might produce some plots using GNUPLOT, a diagram using XFIG, and include an `.eps` graphic a coauthor contributed using some unknown program. All these graphics will, most likely, use different line widths, different fonts, and have different sizes. In addition, authors often use options like `[height=5cm]` when including graphics to scale them to some "nice size."

If the same approach were taken to writing the main text, every section would be written in a different font at a different size. In some sections all theorems would be underlined, in another they would be printed all in uppercase letters, and in another in red. In addition, the margins would be different on each page.

Readers and editors would not tolerate a text if it were written in this fashion, but with graphics they often have to.

To create consistency between graphics and text, stick to the following guidelines:

- Do not scale graphics.

  This means that when generating graphics using an external program, create them "at the right size."

- Use the same font(s) both in graphics and the body text.

- Use the same line width in text and graphics.

  The "line width" for normal text is the width of the stem of letters like T. For TEX, this is usually 0.4 pt. However, some journals will not accept graphics with a normal line width below 0.5 pt.

- When using colors, use a consistent color coding in the text and in graphics. For example, if red is supposed to alert the reader to something in the main text, use red also in graphics for important parts of the graphic. If blue is used for structural elements like headlines and section titles, use blue also for structural elements of your graphic.

  However, graphics may also use a logical intrinsic color coding. For example, no matter what colors you normally use, readers will generally assume, say, that the color green as "positive, go, ok" and red as "alert, warning, action."

Creating consistency when using different graphic programs is almost impossible. For this reason, you should consider sticking to a single graphic program.

4.5 Labels in Graphics

Almost all graphics will contain labels, that is, pieces of text that explain parts of the graphics. When placing labels, stick to the following guidelines:

- Follow the rule of consistency when placing labels. You should do so in two ways: First, be consistent with the main text, that is, use the same font as the main text also for labels. Second, be consistent between labels, that is, if you format some labels in some particular way, format all labels in this way.

- In addition to using the same fonts in text and graphics, you should also use the same notation. For example, if you write $1/2$ in your main text, also use "$1/2$" as labels in graphics, not "0.5". A π is a "π" and not "3.141". Finally, $\mathrm{e}^{-\mathrm{i}\pi}$ is "$\mathrm{e}^{-\mathrm{i}\pi}$", not "$-1$", let alone "-1".

- Labels should be legible. They should not only have a reasonably large size, they also should not be obscured by lines or other text. This also applies to of lines and text *behind* the labels.

- Labels should be "in place." Whenever there is enough space, labels should be placed next to the thing they label. Only if necessary, add a (subdued) line from the label to the labeled object. Try to avoid labels that only reference explanations in external legends. Reader have to jump back and forth between the explanation and the object that is described.

- Consider subduing "unimportant" labels using, for example, a gray color. This will keep the focus on the actual graphic.

4.6 Plots and Charts

One of the most frequent kind of graphics, especially in scientific papers, are *plots*. They come in a large variety, including simple line plots, parametric plots, three dimensional plots, pie charts, and many more.

Unfortunately, plots are notoriously hard to get right. Partly, the default settings of programs like GNUPLOT or Excel are to blame for this since these programs make it very convenient to create bad plots.

The first question you should ask yourself when creating a plot is the following:

- Are there enough data points to merit a plot?

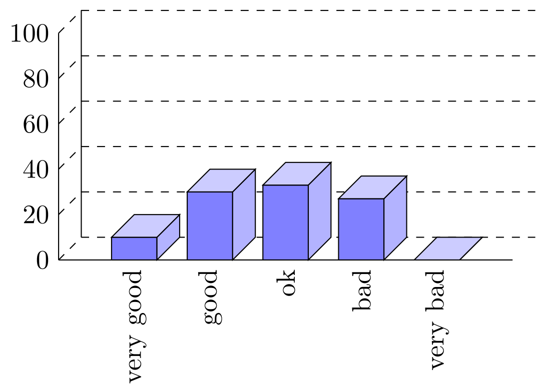
If the answer is "not really," use a table.

A typical situation where a plot is unnecessary is when people present a few numbers in a bar diagram. Here is a real-life example: At the end of a seminar a lecturer asked the participants for feedback. Of the 50 participants, 30 returned the feedback form. According to the feedback, three participants considered the seminar "very good," nine considered it "good," ten "ok," eight "bad," and no one thought that the seminar was "very bad."

A simple way of summing up this information is the following table:

Rating given	*Participants (out of 50) who gave this rating*	*Percentage*
"very good"	3	6%
"good"	9	18%
"ok"	10	20%
"bad"	8	16%
"very bad"	0	0%
none	20	40%

What the lecturer did was to visualize the data using a 3D bar diagram. It looked like this:



Both the table and the "plot" have about the same size. If your first thought is "the graphic looks nicer than the table," try to answer the following questions based on the information in the table or in the graphic:

1. How many participants where there?
2. How many participants returned the feedback form?
3. What percentage of the participants returned the feedback form?
4. How many participants checked "very good"?
5. What percentage out of all participants checked "very good"?
6. Did more than a quarter of the participants check "bad" or "very bad"?
7. What percentage of the participants that returned the form checked "very good"?

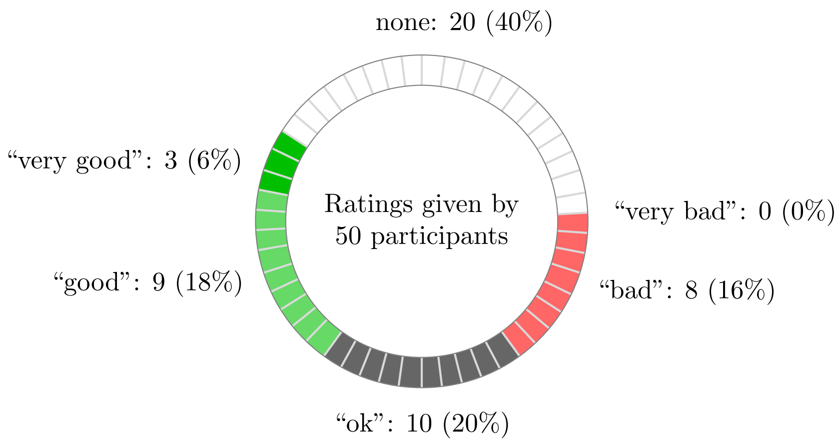
Sadly, the graphic does not allow us to answer *a single one of these questions*. The table answers all of them directly, except for the last one. In essence, the information density of the graphic is very nearly zero. The table has a much higher information density; despite the fact that it uses quite a lot of white space to present a few numbers.

Here is the list of things that went wrong with the 3D-bar diagram:

- The whole graphic is dominated by irritating background lines.
- It is not clear what the numbers at the left mean; presumably percentages, but it might also be the absolute number of participants.
- The labels at the bottom are rotated, making them hard to read.

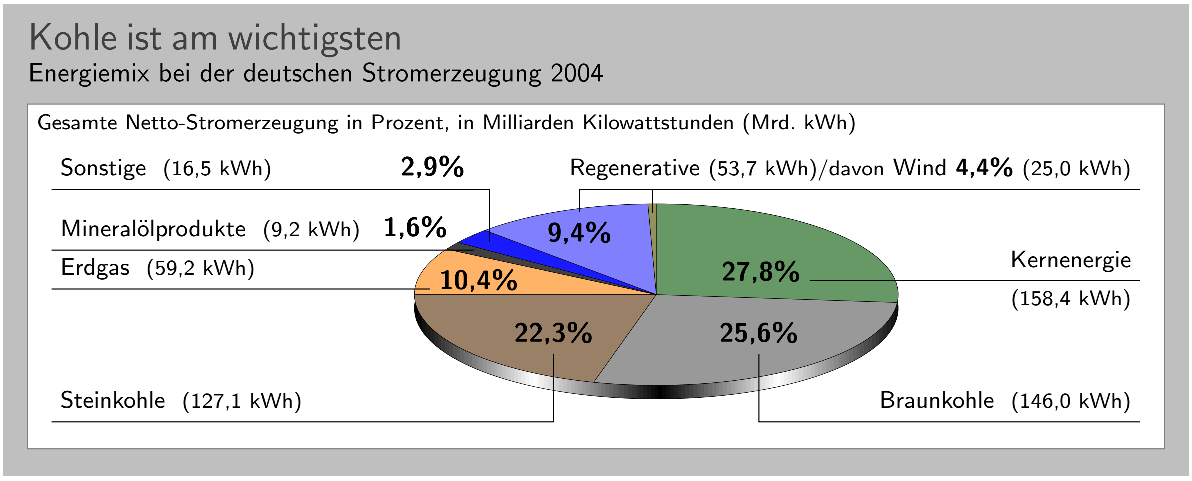
  (In the real presentation that I saw, the text was rendered at a very low resolution with about 10 by 6 pixels per letter with wrong kerning, making the rotated text almost impossible to read.)
- The third dimension adds complexity to the graphic without adding information.
- The three dimensional setup makes it much harder to gauge the height of the bars correctly. Consider the "bad" bar. It the number this bar stands for more than 20 or less? While the front of the bar is below the 20 line, the back of the bar (which counts) is above.
- It is impossible to tell which numbers are represented by the bars. Thus, the bars needlessly hide the information these bars are all about.
- What do the bar heights add up to? Is it 100% or 60%?
- Does the bar for "very bad" represent 0 or 1?
- Why are the bars blue?

You might argue that in the example the exact numbers are not important for the graphic. The important things is the "message," which is that there are more "very good" and "good" ratings than "bad" and "very bad." However, to convey this message either use a sentence that says so or use a graphic that conveys this message more clearly:



The above graphic has about the same information density as the table (about the same size and the same numbers are shown). In addition, one can directly "see" that there are more good or very good ratings than bad ones. One can also "see" that the number of people who gave no rating at all is not negligible, which is quite common for feedback forms.

Charts are not always a good idea. Let us look at an example that I redrew from a pie chart in *Die Zeit*, June 4th, 2005:



This graphic has been redrawn in Ti*k*Z, but the original looks very similar.
At first sight, the graphic looks "nice and informative," but there are a lot of things that went wrong:

- The chart is three dimensional. However, the shadings add nothing "information-wise," at best, they distract.
- In a 3D-pie-chart the relative sizes are very strongly distorted. For example, the area taken up by the gray color of "Braunkohle" is larger than the area taken up by the green color of "Kernenergie" *despite the fact that the percentage of Braunkohle is less than the percentage of Kernenergie.*
- The 3D-distortion gets worse for small areas. The area of "Regenerative" somewhat larger than the area of "Erdgas." The area of "Wind" is slightly smaller than the area of "Mineralölprodukte" *although the percentage of Wind is nearly three times larger than the percentage of Mineralölprodukte.*

  In the last case, the different sizes are only partly due to distortion. The designer(s) of the original graphic have also made the "Wind" slice too small, even taking distortion into account. (Just compare the size of "Wind" to "Regenerative" in general.)
- According to its caption, this chart is supposed to inform us that coal was the most important energy source in Germany in 2004. Ignoring the strong distortions caused by the superfluous and misleading 3D-setup, it takes quite a while for this message to get across.

  Coal as an energy source is split up into two slices: one for "Steinkohle" and one for "Braunkohle" (two different kinds of coal). When you add them up, you see that the whole lower half of the pie chart is taken up by coal.

  The two areas for the different kinds of coal are not visually linked at all. Rather, two different colors are used, the labels are on different sides of the graphic. By comparison, "Regenerative" and "Wind" are very closely linked.
- The color coding of the graphic follows no logical pattern at all. Why is nuclear energy green? Regenerative energy is light blue, "other sources" are blue. It seems more like a joke that the area for "Braunkohle" (which literally translates to "brown coal") is stone gray, while the area for "Steinkohle" (which literally translates to "stone coal") is brown.
- The area with the lightest color is used for "Erdgas." This area stands out most because of the brighter color. However, for this chart "Erdgas" is not really important at all.

Edward Tufte calls graphics like the above "chart junk."
Here are a few recommendations that may help you avoid producing chart junk:

- Do not use 3D pie charts. They are *evil.*
- Consider using a table instead of a pie chart.
- Due not apply colors randomly; use them to direct the readers's focus and to group things.
- Do not use background patterns, like a crosshatch or diagonal lines, instead of colors. They distract. Background patterns in information graphics are *evil.*

4.7 Attention and Distraction

Pick up your favorite fiction novel and have a look at a typical page. You will notice that the page is very uniform. Nothing is there to distract the reader while reading; no large headlines, no bold text, no large white areas. Indeed, even when the author does wish to emphasize something, this is done using italic letters. Such letters blend nicely with the main text—at a distance you will not be able to tell whether a page contains italic letters, but you would notice a single bold word immediately. The reason novels are typeset this way is the following paradigm: Avoid distractions.

Good typography (like good organization) is something you do *not* notice. The job of typography is to make reading the text, that is, "absorbing" its information content, as effortless as possible. For a novel, readers absorb the content by reading the text line-by-line, as if they were listening to someone telling the story. In this situation anything on the page that distracts the eye from going quickly and evenly from line to line will make the text harder to read.

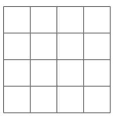
Now, pick up your favorite weekly magazine or newspaper and have a look at a typical page. You will notice that there is quite a lot "going on" on the page. Fonts are used at different sizes and in different arrangements, the text is organized in narrow columns, typically interleaved with pictures. The reason magazines are typeset in this way is another paradigm: Steer attention.

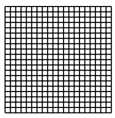
Readers will not read a magazine like a novel. Instead of reading a magazine line-by-line, we use headlines and short abstracts to check whether we want to read a certain article or not. The job of typography is to steer our attention to these abstracts and headlines, first. Once we have decided that we want to read an article, however, we no longer tolerate distractions, which is why the main text of articles is typeset exactly the same way as a novel.

The two principles "avoid distractions" and "steer attention" also apply to graphics. When you design a graphic, you should eliminate everything that will "distract the eye." At the same time, you should try to actively help the reader "through the graphic" by using fonts/colors/line widths to highlight different parts.

Here is a non-exhaustive list of things that can distract readers:

- Strong contrasts will always be registered first by the eye. For example, consider the following two grids:

  Even though the left grid comes first in our normal reading order, the right one is much more likely to be seen first: The white-to-black contrast is higher than the gray-to-white contrast. In addition, there are more "places" adding to the overall contrast in the right grid.

  Things like grids and, more generally, help lines usually should not grab the attention of the readers and, hence, should be typeset with a low contrast to the background. Also, a loosely-spaced grid is less distracting than a very closely-spaced grid.

- Dashed lines create many points at which there is black-to-white contrast. Dashed or dotted lines can be very distracting and, hence, should be avoided in general.

  Do not use different dashing patterns to differentiate curves in plots. You loose data points this way and the eye is not particularly good at "grouping things according to a dashing pattern." The eye is *much* better at grouping things according to colors.

- Background patterns filling an area using diagonal lines or horizontal and vertical lines or just dots are almost always distracting and, usually, serve no real purpose.

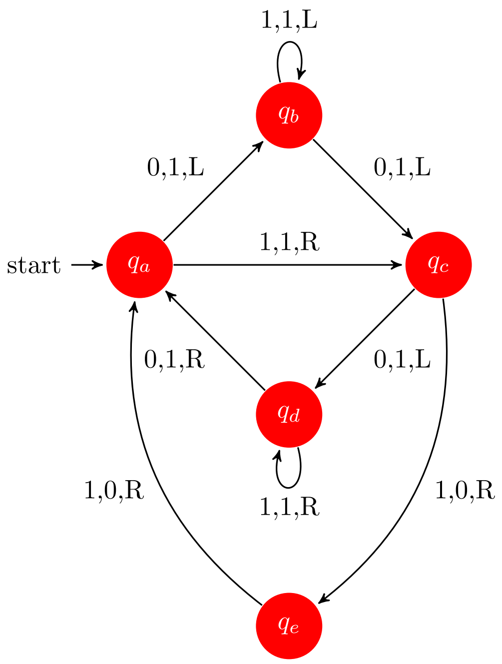
- Background images and shadings distract and only seldom add anything of importance to a graphic.

- Cute little cliparts can easily draw attention away from the data.

Part II
Installation and Configuration

by Till Tantau

This part explains how the system is installed. Typically, someone has already done so for your system, so this part can be skipped; but if this is not the case and you are the poor fellow who has to do the installation, read the present part.



The current candidate for the busy beaver for five states. It is presumed that this Turing machine writes a maximum number of 1's before halting among all Turing machines with five states and the tape alphabet $\{0, 1\}$. Proving this conjecture is an open research problem.

```
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto,node distance=2.8cm,semithick]
  \tikzstyle{every state}=[fill=red,draw=none,text=white]

  \node[initial,state] (A)                    {$q_a$};
  \node[state]         (B) [above right of=A] {$q_b$};
  \node[state]         (D) [below right of=A] {$q_d$};
  \node[state]         (C) [below right of=B] {$q_c$};
  \node[state]         (E) [below of=D]       {$q_e$};

  \path (A) edge              node {0,1,L} (B)
            edge              node {1,1,R} (C)
        (B) edge [loop above] node {1,1,L} (B)
            edge              node {0,1,L} (C)
        (C) edge              node {0,1,L} (D)
            edge [bend left]  node {1,0,R} (E)
        (D) edge [loop below] node {1,1,R} (D)
            edge              node {0,1,R} (A)
        (E) edge [bend left]  node {1,0,R} (A);

   \node [right=1cm,text width=8cm,font=\footnotesize] at (C)
   {
     The current candidate for the busy beaver for five states. It is
     presumed that this Turing machine writes a maximum number of
     $1$'s before halting among all Turing machines with five states
     and the tape alphabet $\{0, 1\}$. Proving this conjecture is an
     open research problem.
   };
\end{tikzpicture}
```

5 Installation

There are different ways of installing PGF, depending on your system and needs, and you may need to install other packages as well as, see below. Before installing, you may wish to review the licenses under which the package is distributed, see Section 6.

Typically, the package will already be installed on your system. Naturally, in this case you do not need to worry about the installation process at all and you can skip the rest of this section.

5.1 Package and Driver Versions

This documentation is part of version 1.18 of the PGF package. In order to run PGF, you need a reasonably recent TEX installation. When using LATEX, you need the following packages installed (newer versions should also work):

- `xcolor` version 2.00.
- `xkeyval` version 1.8, if you wish to use Ti*k*Z.

With plain TEX, `xcolor` is not needed, but you obviously do not get its (full) functionality.

Currently, PGF supports the following backend drivers:

- `pdftex` version 0.14 or higher. Earlier versions do not work.
- `dvips` version 5.94a or higher. Earlier versions may also work.

  For inter-picture connections, you need process pictures using `pdftex` version 1.40 or higher running in DVI mode.
- `dvipdfm` version 0.13.2c or higher. Earlier versions may also work.

  For inter-picture connections, you need process pictures using `pdftex` version 1.40 or higher running in DVI mode.
- `tex4ht` version 2003-05-05 or higher. Earlier versions may also work.
- `vtex` version 8.46a or higher. Earlier versions may also work.
- `textures` version 2.1 or higher. Earlier versions may also work.

Currently, PGF supports the following formats:

- `latex` with complete functionality.
- `plain` with complete functionality, except for graphics inclusion, which works only for pdfTEX.
- `context` with complete functionality, except for graphics inclusion, which works only for pdfTEX.

For more details, see Section 7.

5.2 Installing Prebundled Packages

I do not create or manage prebundled packages of PGF, but, fortunately, nice other people do. I cannot give detailed instructions on how to install these packages, since I do not manage them, but I *can* tell you were to find them. If you have a problem with installing, you might wish to have a look at the Debian page or the MikTEX page first.

5.2.1 Debian

The command "`aptitude install pgf`" should do the trick. Sit back and relax. In detail, the following packages are installed:

```
http://packages.debian.org/pgf
http://packages.debian.org/latex-xcolor
```

5.2.2 MiKTeX

For MiKTEX, use the update wizard to install the (latest versions of the) packages called `pgf`, `xcolor`, and `xkeyval`.

5.3 Installation in a texmf Tree

For a permanent installation, you place the files of the the PGF package in an appropriate `texmf` tree.

When you ask TEX to use a certain class or package, it usually looks for the necessary files in so-called `texmf` trees. These trees are simply huge directories that contain these files. By default, TEX looks for files in three different `texmf` trees:

- The root `texmf` tree, which is usually located at `/usr/share/texmf/` or `c:\texmf\` or somewhere similar.
- The local `texmf` tree, which is usually located at `/usr/local/share/texmf/` or `c:\localtexmf\` or somewhere similar.
- Your personal `texmf` tree, which is usually located in your home directory at `~/texmf/` or `~/Library/texmf/`.

You should install the packages either in the local tree or in your personal tree, depending on whether you have write access to the local tree. Installation in the root tree can cause problems, since an update of the whole TEX installation will replace this whole tree.

5.3.1 Installation that Keeps Everything Together

Once you have located the right texmf tree, you must decide whether you want to install PGF in such a way that "all its files are kept in one place" or whether you want to be "TDS-compliant," where TDS means "TEX directory structure."

If you want to keep "everything in one place," inside the `texmf` tree that you have chosen create a sub-sub-directory called `texmf/tex/generic/pgf` or `texmf/tex/generic/pgf-1.18`, if you prefer. Then place all files of the `pgf` package in this directory. Finally, rebuild TEX's filename database. This is done by running the command `texhash` or `mktexlsr` (they are the same). In MikTEX, there is a menu option to do this.

5.3.2 Installation that is TDS-Compliant

While the above installation process is the most "natural" one and although I would like to recommend it since it makes updating and managing the PGF package easy, it is not TDS-compliant. If you want to be TDS-compliant, proceed as follows: (If you do not know what TDS-compliant means, you probably do not want to be TDS-compliant.)

The `.tar` file of the `pgf` package contains the following files and directories at its root: `README`, `doc`, `generic`, `plain`, and `latex`. You should "merge" each of the four directories with the following directories `texmf/doc`, `texmf/tex/generic`, `texmf/tex/plain`, and `texmf/tex/latex`. For example, in the `.tar` file the `doc` directory contains just the directory `pgf`, and this directory has to be moved to `texmf/doc/pgf`. The root `README` file can be ignored since it is reproduced in `doc/pgf/README`.

You may also consider keeping everything in one place and using symbolic links to point from the TDS-compliant directories to the central installation.

For a more detailed explanation of the standard installation process of packages, you might wish to consult `http://www.ctan.org/installationadvice/`. However, note that the PGF package does not come with a `.ins` file (simply skip that part).

5.4 Updating the Installation

To update your installation from a previous version, all you need to do is to replace everything in the directory `texmf/tex/generic/pgf` with the files of the new version (or in all the directories where `pgf` was installed, if you chose a TDS-compliant installation). The easiest way to do this is to first delete the old version and then proceed as described above. Sometimes, there are changes in the syntax of certain command from version to version. If things no longer work that used to work, you may wish to have a look at the release notes and at the change log.

6 Licenses and Copyright

6.1 Which License Applies?

Different parts of the PGF package are distributed under different licenses:

1. The *code* of the package is dual-license. This means that you can decide which license you wish to use when using the PGF package. The two options are:
   (a) You can use the GNU Public License, version 2.
   (b) You can use the LaTeX Project Public License, version 1.3c.
2. The *documentation* of the package is also dual-license. Again, you can choose between two options:
   (a) You can use the GNU Free Documentation License, version 1.2.
   (b) You can use the LaTeX Project Public License, version 1.3c.

The "documentation of the package" refers to all files in the subdirectory `doc` of the `pgf` package. A detailed listing can be found in the file `doc/generic/pgf/licenses/manifest-documentation.txt`. All files in other directories are part of the "code of the package." A detailed listing can be found in the file `doc/generic/pgf/licenses/manifest-code.txt`.

In the resest of this section, the licenses are presented. The following text is copyrighted, see the plain text versions of these licenses in the directory `doc/generic/pgf/licenses` for details.

The example picture used in this manual, the Brave GNU World logo, is taken from the Brave GNU World homepage, where it is copyrighted as follows: "Copyright (C) 1999, 2000, 2001, 2002, 2003, 2004 Georg C. F. Greve. Permission is granted to make and distribute verbatim copies of this transcript as long as the copyright and this permission notice appear."

6.2 The GNU Public License, Version 2

6.2.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

6.2.2 Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsubsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

6.2.3 No Warranty

10. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

11. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

6.3 The LaTeX Project Public License, Version 1.3c 2006-05-20

6.3.1 Preamble

The LaTeX Project Public License (LPPL) is the primary license under which the the LaTeX kernel and the base LaTeX packages are distributed.

You may use this license for any work of which you hold the copyright and which you wish to distribute. This license may be particularly suitable if your work is TeX-related (such as a LaTeX package), but it is written in such a way that you can use it even if your work is unrelated to TeX.

The section 'WHETER AND HOW TO DISTRIBUTE WORKS UNDER THIS LICENSE', below, gives instructions, examples, and recommendations for authors who are considering distributing their works under this license.

This license gives conditions under which a work may be distributed and modified, as well as conditions under which modified versions of that work may be distributed.

We, the LaTeX3 Project, believe that the conditions below give you the freedom to make and distribute modified versions of your work that conform with whatever technical specifications you wish while maintaining the availability, integrity, and reliability of that work. If you do not see how to achieve your goal while meeting these conditions, then read the document '`cfgguide.tex`' and '`modguide.tex`' in the base LaTeX distribution for suggestions.

6.3.2 Definitions

In this license document the following terms are used:

Work Any work being distributed under this License.

Derived Work Any work that under any applicable law is derived from the Work.

Modification Any procedure that produces a Derived Work under any applicable law – for example, the production of a file containing an original file associated with the Work or a significant portion of such a file, either verbatim or with modifications and/or translated into another language.

Modify To apply any procedure that produces a Derived Work under any applicable law.

Distribution Making copies of the Work available from one person to another, in whole or in part. Distribution includes (but is not limited to) making any electronic components of the Work accessible by file transfer protocols such as FTP or HTTP or by shared file systems such as Sun's Network File System (NFS).

Compiled Work A version of the Work that has been processed into a form where it is directly usable on a computer system. This processing may include using installation facilities provided by the Work, transformations of the Work, copying of components of the Work, or other activities. Note that modification of any installation facilities provided by the Work constitutes modification of the Work.

Current Maintainer A person or persons nominated as such within the Work. If there is no such explicit nomination then it is the 'Copyright Holder' under any applicable law.

Base Interpreter A program or process that is normally needed for running or interpreting a part or the whole of the Work.

A Base Interpreter may depend on external components but these are not considered part of the Base Interpreter provided that each external component clearly identifies itself whenever it is used interactively. Unless explicitly specified when applying the license to the Work, the only applicable Base Interpreter is a 'LaTeX-Format' or in the case of files belonging to the 'LaTeX-format' a program implementing the 'TeX language'.

6.3.3 Conditions on Distribution and Modification

1. Activities other than distribution and/or modification of the Work are not covered by this license; they are outside its scope. In particular, the act of running the Work is not restricted and no requirements are made concerning any offers of support for the Work.

2. You may distribute a complete, unmodified copy of the Work as you received it. Distribution of only part of the Work is considered modification of the Work, and no right to distribute such a Derived Work may be assumed under the terms of this clause.

3. You may distribute a Compiled Work that has been generated from a complete, unmodified copy of the Work as distributed under Clause 2 above, as long as that Compiled Work is distributed in such a way that the recipients may install the Compiled Work on their system exactly as it would have been installed if they generated a Compiled Work directly from the Work.

4. If you are the Current Maintainer of the Work, you may, without restriction, modify the Work, thus creating a Derived Work. You may also distribute the Derived Work without restriction, including Compiled Works generated from the Derived Work. Derived Works distributed in this manner by the Current Maintainer are considered to be updated versions of the Work.

5. If you are not the Current Maintainer of the Work, you may modify your copy of the Work, thus creating a Derived Work based on the Work, and compile this Derived Work, thus creating a Compiled Work based on the Derived Work.

6. If you are not the Current Maintainer of the Work, you may distribute a Derived Work provided the following conditions are met for every component of the Work unless that component clearly states in the copyright notice that it is exempt from that condition. Only the Current Maintainer is allowed to add such statements of exemption to a component of the Work.

   (a) If a component of this Derived Work can be a direct replacement for a component of the Work when that component is used with the Base Interpreter, then, wherever this component of the Work identifies itself to the user when used interactively with that Base Interpreter, the replacement component of this Derived Work clearly and unambiguously identifies itself as a modified version of this component to the user when used interactively with that Base Interpreter.

(b) Every component of the Derived Work contains prominent notices detailing the nature of the changes to that component, or a prominent reference to another file that is distributed as part of the Derived Work and that contains a complete and accurate log of the changes.

(c) No information in the Derived Work implies that any persons, including (but not limited to) the authors of the original version of the Work, provide any support, including (but not limited to) the reporting and handling of errors, to recipients of the Derived Work unless those persons have stated explicitly that they do provide such support for the Derived Work.

(d) You distribute at least one of the following with the Derived Work:

i. A complete, unmodified copy of the Work; if your distribution of a modified component is made by offering access to copy the modified component from a designated place, then offering equivalent access to copy the Work from the same or some similar place meets this condition, even though third parties are not compelled to copy the Work along with the modified component;

ii. Information that is sufficient to obtain a complete, unmodified copy of the Work.

7. If you are not the Current Maintainer of the Work, you may distribute a Compiled Work generated from a Derived Work, as long as the Derived Work is distributed to all recipients of the Compiled Work, and as long as the conditions of Clause 6, above, are met with regard to the Derived Work.

8. The conditions above are not intended to prohibit, and hence do not apply to, the modification, by any method, of any component so that it becomes identical to an updated version of that component of the Work as it is distributed by the Current Maintainer under Clause 4, above.

9. Distribution of the Work or any Derived Work in an alternative format, where the Work or that Derived Work (in whole or in part) is then produced by applying some process to that format, does not relax or nullify any sections of this license as they pertain to the results of applying that process.

10. (a) A Derived Work may be distributed under a different license provided that license itself honors the conditions listed in Clause 6 above, in regard to the Work, though it does not have to honor the rest of the conditions in this license.

(b) If a Derived Work is distributed under a different license, that Derived Work must provide sufficient documentation as part of itself to allow each recipient of that Derived Work to honor the restrictions in Clause 6 above, concerning changes from the Work.

11. This license places no restrictions on works that are unrelated to the Work, nor does this license place any restrictions on aggregating such works with the Work by any means.

12. Nothing in this license is intended to, or may be used to, prevent complete compliance by all parties with all applicable laws.

6.3.4 No Warranty

There is no warranty for the Work. Except when otherwise stated in writing, the Copyright Holder provides the Work 'as is', without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Work is with you. Should the Work prove defective, you assume the cost of all necessary servicing, repair, or correction.

In no event unless required by applicable law or agreed to in writing will The Copyright Holder, or any author named in the components of the Work, or any other party who may distribute and/or modify the Work as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of any use of the Work or out of inability to use the Work (including, but not limited to, loss of data, data being rendered inaccurate, or losses sustained by anyone as a result of any failure of the Work to operate with any other programs), even if the Copyright Holder or said author or said other party has been advised of the possibility of such damages.

6.3.5 Maintenance of The Work

The Work has the status 'author-maintained' if the Copyright Holder explicitly and prominently states near the primary copyright notice in the Work that the Work can only be maintained by the Copyright Holder or simply that it is 'author-maintained'.

The Work has the status 'maintained' if there is a Current Maintainer who has indicated in the Work that they are willing to receive error reports for the Work (for example, by supplying a valid e-mail address). It is not required for the Current Maintainer to acknowledge or act upon these error reports.

The Work changes from status 'maintained' to 'unmaintained' if there is no Current Maintainer, or the person stated to be Current Maintainer of the work cannot be reached through the indicated means of communication for a period of six months, and there are no other significant signs of active maintenance.

You can become the Current Maintainer of the Work by agreement with any existing Current Maintainer to take over this role.

If the Work is unmaintained, you can become the Current Maintainer of the Work through the following steps:

1. Make a reasonable attempt to trace the Current Maintainer (and the Copyright Holder, if the two differ) through the means of an Internet or similar search.
2. If this search is successful, then enquire whether the Work is still maintained.
   (a) If it is being maintained, then ask the Current Maintainer to update their communication data within one month.
   (b) If the search is unsuccessful or no action to resume active maintenance is taken by the Current Maintainer, then announce within the pertinent community your intention to take over maintenance. (If the Work is a LaTeX work, this could be done, for example, by posting to `comp.text.tex`.)
3. (a) If the Current Maintainer is reachable and agrees to pass maintenance of the Work to you, then this takes effect immediately upon announcement.
   (b) If the Current Maintainer is not reachable and the Copyright Holder agrees that maintenance of the Work be passed to you, then this takes effect immediately upon announcement.
4. If you make an 'intention announcement' as described in 2b above and after three months your intention is challenged neither by the Current Maintainer nor by the Copyright Holder nor by other people, then you may arrange for the Work to be changed so as to name you as the (new) Current Maintainer.
5. If the previously unreachable Current Maintainer becomes reachable once more within three months of a change completed under the terms of 3b or 4, then that Current Maintainer must become or remain the Current Maintainer upon request provided they then update their communication data within one month.

A change in the Current Maintainer does not, of itself, alter the fact that the Work is distributed under the LPPL license.

If you become the Current Maintainer of the Work, you should immediately provide, within the Work, a prominent and unambiguous statement of your status as Current Maintainer. You should also announce your new status to the same pertinent community as in 2b above.

6.3.6 Whether and How to Distribute Works under This License

This section contains important instructions, examples, and recommendations for authors who are considering distributing their works under this license. These authors are addressed as 'you' in this section.

6.3.7 Choosing This License or Another License

If for any part of your work you want or need to use *distribution* conditions that differ significantly from those in this license, then do not refer to this license anywhere in your work but, instead, distribute your work under a different license. You may use the text of this license as a model for your own license, but your license should not refer to the LPPL or otherwise give the impression that your work is distributed under the LPPL.

The document '`modguide.tex`' in the base LaTeX distribution explains the motivation behind the conditions of this license. It explains, for example, why distributing LaTeX under the GNU General Public

License (GPL) was considered inappropriate. Even if your work is unrelated to LaTeX, the discussion in '`modguide.tex`' may still be relevant, and authors intending to distribute their works under any license are encouraged to read it.

6.3.8 A Recommendation on Modification Without Distribution

It is wise never to modify a component of the Work, even for your own personal use, without also meeting the above conditions for distributing the modified component. While you might intend that such modifications will never be distributed, often this will happen by accident – you may forget that you have modified that component; or it may not occur to you when allowing others to access the modified version that you are thus distributing it and violating the conditions of this license in ways that could have legal implications and, worse, cause problems for the community. It is therefore usually in your best interest to keep your copy of the Work identical with the public one. Many works provide ways to control the behavior of that work without altering any of its licensed components.

6.3.9 How to Use This License

To use this license, place in each of the components of your work both an explicit copyright notice including your name and the year the work was authored and/or last substantially modified. Include also a statement that the distribution and/or modification of that component is constrained by the conditions in this license.

Here is an example of such a notice and statement:

```
%% pig.dtx
%% Copyright 2005 M. Y. Name
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3 or later is part of all distributions of LaTeX
% version 2005/12/01 or later.
%
% This work has the LPPL maintenance status 'maintained'.
%
% The Current Maintainer of this work is M. Y. Name.
%
% This work consists of the files pig.dtx and pig.ins
% and the derived file pig.sty.
```

Given such a notice and statement in a file, the conditions given in this license document would apply, with the 'Work' referring to the three files '`pig.dtx`', '`pig.ins`', and '`pig.sty`' (the last being generated from '`pig.dtx`' using '`pig.ins`'), the 'Base Interpreter' referring to any 'LaTeX-Format', and both 'Copyright Holder' and 'Current Maintainer' referring to the person 'M. Y. Name'.

If you do not want the Maintenance section of LPPL to apply to your Work, change 'maintained' above into 'author-maintained'. However, we recommend that you use 'maintained' as the Maintenance section was added in order to ensure that your Work remains useful to the community even when you can no longer maintain and support it yourself.

6.3.10 Derived Works That Are Not Replacements

Several clauses of the LPPL specify means to provide reliability and stability for the user community. They therefore concern themselves with the case that a Derived Work is intended to be used as a (compatible or incompatible) replacement of the original Work. If this is not the case (e.g., if a few lines of code are reused for a completely different task), then clauses 6b and 6d shall not apply.

6.3.11 Important Recommendations

Defining What Constitutes the Work The LPPL requires that distributions of the Work contain all the files of the Work. It is therefore important that you provide a way for the licensee to determine which files constitute the Work. This could, for example, be achieved by explicitly listing all the files of the Work near the copyright notice of each file or by using a line such as:

```
% This work consists of all files listed in manifest.txt.
```

in that place. In the absence of an unequivocal list it might be impossible for the licensee to determine what is considered by you to comprise the Work and, in such a case, the licensee would be entitled to make reasonable conjectures as to which files comprise the Work.

6.4 GNU Free Documentation License, Version 1.2, November 2002

6.4.1 Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

6.4.2 Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

6.4.3 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

6.4.4 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

6.4.5 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified

Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6.4.6 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6.4.7 Collection of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

6.4.8 Aggregating with independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

6.4.9 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

6.4.10 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6.4.11 Future Revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

6.4.12 Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

7 Input and Output Formats

TEX was designed to be a flexible system. This is true both for the *input* for TEX as well as for the *output*. The present section explains which input formats there are and how they are supported by PGF. It also explains which different output formats can be produced.

7.1 Supported Input Formats

TEX does not prescribe exactly how your input should be formatted. While it is *customary* that, say, an opening brace starts a scope in TEX, this is by no means necessary. Likewise, it is *customary* that environments start with `\begin`, but TEX could not really care less about the exact command name.

Even though TEX can be reconfigured, users can not. For this reason, certain *input formats* specify a set of commands and conventions how input for TEX should be formatted. There are currently three "major" formats: Donald Knuth's original `plain` TEX format, Leslie Lamport's popular LATEX format, and Hans Hangen's ConTEXt format.

7.1.1 Using the LATEX Format

Using PGF and Ti*k*Z with the LATEX format is easy: You say `\usepackage{pgf}` or `\usepackage{tikz}`. Usually, that is all you need to do, all configuration will be done automatically and (hopefully) correctly.

The style files used for the LATEX format reside in the subdirectory `latex/pgf/` of the PGF-system. Mainly, what these files do is to include files in the directory `generic/pgf`. For example, here is the content of the file `latex/pgf/frontends/tikz.sty`:

```
% Copyright 2006 by Till Tantau
%
% This file may be distributed and/or modified
%
% 1. under the LaTeX Project Public License and/or
% 2. under the GNU Public License.
%
% See the file doc/generic/pgf/licenses/LICENSE for more details.


\RequirePackage{pgf,calc,pgffor,xkeyval}

\input{tikz.code.tex}

\endinput
```

The files in the `generic/pgf` directory do the actual work.

7.1.2 Using the Plain TEX Format

When using the plain TEX format, you say `\input{pgf.tex}` or `\input{tikz.tex}`. Then, instead of `\begin{pgfpicture}` and `\end{pgfpicture}` you use `\pgfpicture` and `\endpgfpicture`.

Unlike for the LATEX format, PGF is not as good at discerning the appropriate configuration for the plain TEX format. In particular, it can only automatically determine the correct output format if you use `pdftex` or `tex` plus `dvips`. For all other output formats you need to set the macro `\pgfsysdriver` to the correct value. See the description of using output formats later on.

PGF was originally written for use with LATEX and this shows in a number of places. Nevertheless, the plain TEX support is reasonably good.

Like the LATEX style files, the plain TEX files like `tikz.tex` also just include the correct `tikz.code.tex` file.

7.1.3 Using the ConTEXt Format

When using the ConTEXt format, you say `\usemodule[pgf]` or `\usemodule[tikz]`. As for the plain TEX format you also have to replace the start- and end-of-environment tags as follows: Instead of `\begin{pgfpicture}` and `\end{pgfpicture}` you use `\startpgfpicture` and `\stoppgfpicture`; similarly, instead of `\begin{tikzpicture}` and `\end{tikzpicture}` you use must now use `\starttikzpicture` and `\stoptikzpicture`; and so on for other environments.

The ConTEXt support is very similar to the plain TEX support, so the same restrictions apply: You may have to set the output format directly and graphics inclusion may be a problem.

In addition to pgf and tikz there also exist modules like pgfcore, pgfbaseimage, pgflibrarysnakes and so on. To use them, you may need to include the module pgfmod first (the modules pgf and tikz both include pgfmod for you, so typically you can skip this). This special module is necessary since ConTEXt satanically restricts the length of module names to 6 characters and PGF's long names are mapped to cryptic 6-letter-names for you by the module pgfmod.

7.2 Supported Output Formats

An output format is a format in which TEX outputs the text it has typeset. Producing the output is (conceptually) a two-stage process:

1. TEX typesets your text and graphics. The result of this typesetting is mainly a long list of letter–coordinate pairs, plus (possibly) some "special" commands. This long list of pairs is written to something called a .dvi-file.
2. Some other program reads this .dvi-file and translates the letter–coordinate pairs into, say, PostScript commands for placing the given letter at the given coordinate.

The classical example of this process is the combination of latex and dvips. The latex program (which is just the tex program called with the LATEX-macros preinstalled) produces a .dvi-file as its output. The dvips program takes this output and produces a .ps-file (a PostScript) file. Possibly, this file is further converted using, say, ps2pdf, whose name is supposed to mean "PostScript to PDF." Another example of programs using this process is the combination of tex and dvipdfm. The dvipdfm program takes a .dvi-file as input and translates the letter–coordinate pairs therein into PDF-commands, resulting in a .pdf file directly. Finally, the tex4ht is also a program that takes a .dvi-file and produces an output, this time it is a .html file. The programs pdftex and pdflatex are special: They directly produce a .pdf-file without the intermediate .dvi-stage. However, from the programmer's point of view they behave exactly as if there where an intermediate stage.

Normally, TEX only produces letter–coordinate pairs as its "output." This obviously makes is difficult to draw, say, a curve. For this, "special" commands can be used. Unfortunately, these special commands are not the same for the different programs that process the .dvi-file. Indeed, every program that takes a .dvi-file as input has a totally different syntax for the special commands.

One of the main jobs of PGF is to "abstract way" the difference in the syntax of the different programs. However, this means that support for each program has to be "programmed," which is a time-consuming and complicated process.

7.2.1 Selecting the Backend Driver

When TEX typesets your document, it does not know which program you are going to use to transform the .dvi-file. If your .dvi-file does not contain any special commands, this would be fine; but these days almost all .dvi-files contain lots of special commands. It is thus necessary to tell TEX which program you are going to use later on.

Unfortunately, there is no "standard" way of telling this to TEX. For the LATEX format a sophisticated mechanism exists inside the graphics package and PGF plugs into this mechanism. For other formats and when this plugging does not work as expected, it is necessary to tell PGF directly which program you are going to use. This is done by redefining the macro \pgfsysdriver to an appropriate value *before* you load pgf. If you are going to use the dvips program, you set this macro to the value pgfsys-dvips.def; if you use pdftex or pdflatex, you set it to pgfsys-pdftex.def; and so on. In the following, details of the support of the different programs are discussed.

7.2.2 Producing PDF Output

PGF supports three programs that produce PDF output (PDF means "portable document format" and was invented by the Adobe company): dvipdfm, pdftex, and vtex. The pdflatex program is the same as the pdftex program: it uses a different input format, but the output is exactly the same.

File pgfsys-pdftex.def

This is the driver file for use with pdfTEX, that is, with the pdftex or pdflatex command. It includes pgfsys-common-pdf.def.

This driver has the "complete" functionality. This means, everything PGF "can do at all" is implemented in this driver.

File pgfsys-dvipdfm.def

This is a driver file for use with (la)tex followed by dvipdfm. It includes pgfsys-common-pdf.def.

This driver supports most of PGF's features, but there are some restrictions:

1. In LaTeX mode it uses graphicx for the graphics inclusion and does not support masking.
2. In plain TeX mode it does not support image inclusion.
3. For remembering of pictures (inter-picture connections) you need to use a recent version of pdftex running in DVI-mode.

File pgfsys-vtex.def

This is the driver file for use with the commercial VTEX program. Even though it produces PDF output, it includes pgfsys-common-postscript.def. Note that the VTEX program can produce *both* Postscript and PDF output, depending on the command line parameters. However, whether you produce Postscript or PDF output does not change anything with respect to the driver.

This driver supports most of PGF's features, except for the following restrictions:

1. In LaTeX mode it uses graphicx for the graphics inclusion and does not support masking.
2. In plain TeX mode it does not support image inclusion.
3. Shading is fully implemented, but yields the same quality as the implementation for dvips.
4. Opacity is not supported.
5. Remembering of pictures (inter-picture connections) is not supported.

It is also possible to produce a .pdf-file by first producing a PostScript file (see below) and then using a PostScript-to-PDF conversion program like ps2pdf or the Acrobat Distiller.

7.2.3 Producing PostScript Output

File pgfsys-dvips.def

This is a driver file for use with (la)tex followed by dvips. It includes pgfsys-common-postscript.def.

This driver also supports most of PGF's features, except for the following restrictions:

1. In LaTeX mode it uses graphicx for the graphics inclusion and does not support masking.
2. In plain TeX mode it does not support image inclusion.
3. Shading is fully implemented, but the results will not be as good as with a driver producing .pdf as output.
4. Opacity works only in conjunction with newer versions of GhostScript.
5. For remembering of pictures (inter-picture connections) you need to use a recent version of pdftex running in DVI-mode.

File pgfsys-textures.def

This is a driver file for use with the TEXTURES program. It includes pgfsys-common-postscript.def.

This driver has exactly the same restrictions as the driver for dvips.

You can also use the vtex program together with pgfsys-vtex.def to produce Postscript output.

7.2.4 Producing HTML / SVG Output

The tex4ht program converts .dvi-files to .html-files. While the HTML-format cannot be used to draw graphics, the SVG-format can. Using the following driver, you can ask PGF to produce an SVG-picture for each PGF graphic in your text.

File pgfsys-tex4ht.def

This is a driver file for use with the tex4ht program. It includes pgfsys-common-svg.def.

When using this driver you should be aware of the following restrictions:

1. In LaTeX mode it uses graphicx for the graphics inclusion.

2. In plain TEX mode it does not support image inclusion.
3. Remembering of pictures (inter-picture connections) is not supported.
4. Text inside `pgfpictures` is not supported very well. The reason is that the SVG specification currently does not support text very well and it is also not possible to correctly "escape back" to HTML. All these problems will hopefully disappear in the future, but currently only two kinds of text work reasonably well: First, plain text without math mode, special characters or anything else special. Second, *very* simple mathematical text that contains subscripts or superscripts. Even then, variables are not correctly set in italics and, in general, text simple does not look very nice.
5. If you use text that contains anything special, even something as simple as `$\alpha$`, this may corrupt the graphic since `text4ht` does not always produce valid XML code. So, once more, *stick to very simple node text inside graphics.* Sorry.
6. Unlike for other output formats, the bounding box of a picture "really crops" the picture.
7. Matrices do not work.

The driver basically works as follows: When a `{pgfpicture}` is started, appropriate `\special` commands are used to directed the output of `tex4ht` to a new file called `\jobname-xxx.svg`, where `xxx` is a number that is increased for each graphic. Then, till the end of the picture, each (system layer) graphic command creates a special that inserts appropriate SVG literal text into the output file. The exact details are a bit complicated since the imaging model and the processing model of PostScript/PDF and SVG are not quite the same; but they are "close enough" for PGF's purposes.

7.2.5 Producing Perfectly Portable DVI Output

File `pgfsys-dvi.def`

This is a driver file that can be used with any output driver, except for `tex4ht`.

The driver will produce perfectly portable `.dvi` files by composing all pictures entirely of black rectangles, the basic and only graphic shape supported by the TEX core. Even straight, but slanted lines are tricky to get right in this model (they need to be composed of lots of little squares).

Naturally, *very little* is possible with this driver. In fact, so little is possible that it is easier to list what is possible:

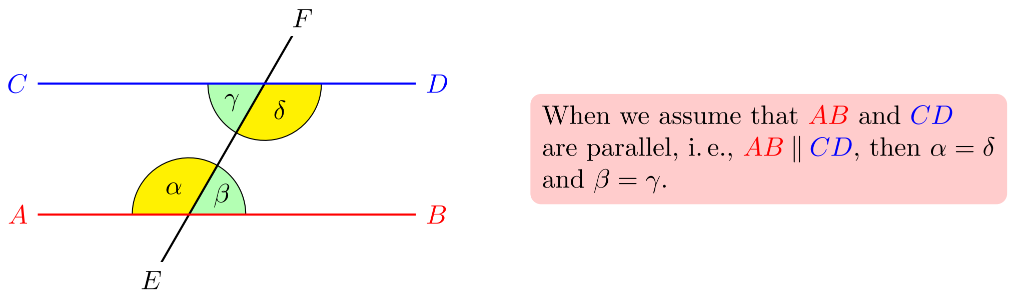
- Text boxes can be placed in the normal way.
- Lines and curves can be drawn (stroked). If they are not horizontal or vertical, they are composed of hundred of small rectangles.
- Lines of different width are supported.
- Transformations are supported.

Note that, say, even filling is not supported! (Let alone color or anything fancy.)

This driver has only one real application: It might be useful when you only need horizontal or vertical lines in a picture. Then, the results are quite satisfactory.

Part III
Ti*k*Z ist *kein* Zeichenprogramm

by Till Tantau




```
\begin{tikzpicture}
  \draw[fill=yellow] (0,0) -- (60:.75cm) arc (60:180:.75cm);
  \draw(120:0.4cm) node {$\alpha$};

  \draw[fill=green!30] (0,0) -- (right:.75cm) arc (0:60:.75cm);
  \draw(30:0.5cm) node {$\beta$};

  \begin{scope}[shift={(60:2cm)}]
    \draw[fill=green!30] (0,0) -- (180:.75cm) arc (180:240:.75cm);
    \draw (30:-0.5cm) node {$\gamma$};

    \draw[fill=yellow] (0,0) -- (240:.75cm) arc (240:360:.75cm);
    \draw (-60:0.4cm) node {$\delta$};
  \end{scope}

  \begin{scope}[thick]
    \draw  (60:-1cm) node[fill=white] {$E$} -- (60:3cm) node[fill=white] {$F$};
    \draw[red]                    (-2,0) node[left] {$A$} -- (3,0) node[right]{$B$};
    \draw[blue,shift={(60:2cm)}] (-3,0) node[left] {$C$} -- (2,0) node[right]{$D$};

    \draw[shift={(60:1cm)},xshift=4cm]
    node [right,text width=6cm,rounded corners,fill=red!20,inner sep=1ex]
    {
      When we assume that $\color{red}AB$ and $\color{blue}CD$ are
      parallel, i.\,e., ${\color{red}AB} \mathbin{\|} \color{blue}CD$,
      then $\alpha = \delta$ and $\beta = \gamma$.
    };
  \end{scope}
\end{tikzpicture}
```

8 Design Principles

This section describes the design principles behind the Ti*k*Z frontend, where Ti*k*Z means "Ti*k*Z ist *kein* Zeichenprogramm." To use Ti*k*Z, as a LaTeX user say `\usepackage{tikz}` somewhere in the preamble, as a plain TeX user say `\input tikz.tex`. Ti*k*Z's job is to make your life easier by providing an easy-to-learn and easy-to-use syntax for describing graphics.

The commands and syntax of Ti*k*Z were influenced by several sources. The basic command names and the notion of path operations is taken from METAFONT, the option mechanism comes from PSTRICKS, the notion of styles is reminiscent of SVG. To make it all work together, some compromises were necessary. I also added some ideas of my own, like meta-arrows and coordinate transformations.

The following basic design principles underlie Ti*k*Z:

1. Special syntax for specifying points.
2. Special syntax for path specifications.
3. Actions on paths.
4. Key-value syntax for graphic parameters.
5. Special syntax for nodes.
6. Special syntax for trees.
7. Grouping of graphic parameters.
8. Coordinate transformation system.

8.1 Special Syntax For Specifying Points

Ti*k*Z provides a special syntax for specifying points and coordinates. In the simplest case, you provide two TeX dimensions, separated by commas, in round brackets as in `(1cm,2pt)`.

You can also specify a point in polar coordinates by using a colon instead of a comma as in `(30:1cm)`, which means "1cm in a 30 degrees direction."

If you do not provide a unit, as in `(2,1)`, you specify a point in PGF's xy-coordinate system. By default, the unit x-vector goes 1cm to the right and the unit y-vector goes 1cm upward.

By specifying three numbers as in `(1,1,1)` you specify a point in PGF's xyz-coordinate system.

It is also possible to use an anchor of a previously defined shape as in `(first node.south)`.

You can add two plus signs before a coordinate as in `++(1cm,0pt)`. This means "1cm to the right of the last point used." This allows you to easily specify relative movements. For example, `(1,0) ++(1,0) ++(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(2,1)`.

Finally, instead of two plus signs, you can also add a single one. This also specifies a point in a relative manner, but it does not "change" the current point used in subsequent relative commands. For example, `(1,0) +(1,0) +(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(1,1)`.

8.2 Special Syntax For Path Specifications

When creating a picture using Ti*k*Z, your main job is the specification of *paths*. A path is a series of straight or curved lines, which need not be connected. Ti*k*Z makes it easy to specify paths, partly using the syntax of METAPOST. For example, to specify a triangular path you use

```
(5pt,0pt) -- (0pt,0pt) -- (0pt,5pt) -- cycle
```

and you get ◺ when you draw this path.

8.3 Actions on Paths

A path is just a series of straight and curved lines, but it is not yet specified what should happen with it. One can *draw* a path, *fill* a path, *shade* it, *clip* it, or do any combination of these. Drawing (also known as *stroking*) can be thought of as taking a pen of a certain thickness and moving it along the path, thereby drawing on the canvas. Filling means that the interior of the path is filled with a uniform color. Obviously, filling makes sense only for *closed* paths and a path is automatically closed prior to filling, if necessary.

Given a path as in `\path (0,0) rectangle (2ex,1ex);`, you can draw it by adding the `draw` option as in `\path[draw] (0,0) rectangle (2ex,1ex);`, which yields . The `\draw` command is just an abbreviation for `\path[draw]`. To fill a path, use the `fill` option or the `\fill` command, which is an abbreviation for `\path[fill]`. The `\filldraw` command is an abbreviation for `\path[fill,draw]`. Shading is caused by the `shade` option (there are `\shade` and `\shadedraw` abbreviations) and clipping by the `clip` option. There is is also a `\clip` command, which does the same as `\path[clip]`, but not commands like `\drawclip`. Use, say, `\draw[clip]` or `\path[draw,clip]` instead.

All of these commands can only be used inside `{tikzpicture}` environments.

Ti*k*Z allows you to use different colors for filling and stroking.

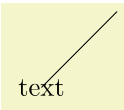
8.4 Key-Value Syntax for Graphic Parameters

Whenever Ti*k*Z draws or fills a path, a large number of graphic parameters influenced the rendering. Examples include the colors used, the dashing pattern, the clipping area, the line width, and many others. In Ti*k*Z, all these options are specified as lists of so called key-value pairs, as in `color=red`, that are passed as optional parameters to the path drawing and filling commands. This usage is similar to PSTRICKS. For example, the following will draw a thick, red triangle;


```
\tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (1,0) -- cycle;
```

8.5 Special Syntax for Specifying Nodes

Ti*k*Z introduces a special syntax for adding text or, more generally, nodes to a graphic. When you specify a path, add nodes as in the following example:




```
\tikz \draw (1,1) node {text} -- (2,2);
```

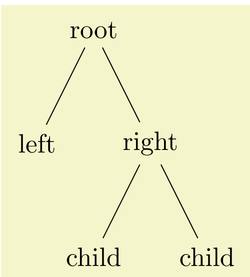
Nodes are inserted at the current position of the path, but only *after* the path has been rendered. When special options are given, as in `\draw (1,1) node[circle,draw] {text};`, the text is not just put at the current position. Rather, it is surrounded by a circle and this circle is "drawn."

You can add a name to a node for later reference either by using the option `name=`⟨*node name*⟩ or by stating the node name in parentheses outside the text as in `node[circle](name){text}`.

Predefined shapes include `rectangle`, `circle`, and `ellipse`, but it is possible (though a bit challenging) to define new shapes.

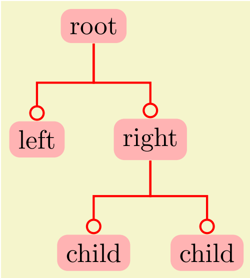
8.6 Special Syntax for Specifying Trees

In addition to the "node syntax," Ti*k*Z also introduces a special syntax for drawing trees. The syntax is intergrated with the special node syntax and only few new commands need to be remebered. In essence, a `node` can be followed by any number of children, each introduced by the keyword `child`. The children are nodes themselves, each of which may have children in turn.

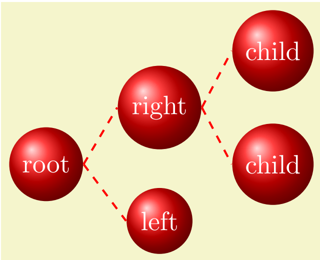



```
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

Since trees are made up from nodes, it is possible to use options to modify the way trees are drawn. Here are two examples of the above tree, redrawn with different options:



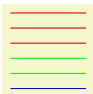

```
\begin{tikzpicture}[edge from parent fork down]
  \tikzstyle{every node}=[fill=red!30,rounded corners]
  \tikzstyle{edge from parent}=[red,-o,thick,draw]
  \node {root}
      child {node {left}}
      child {node {right}
        child {node {child}}
        child {node {child}}
      };
\end{tikzpicture}
```




```
\begin{tikzpicture}
    [parent anchor=east,child anchor=west,grow=east]
  \tikzstyle{every node}=[ball color=red,circle,text=white]
  \tikzstyle{edge from parent}=[draw,dashed,thick,red]
  \node {root}
      child {node {left}}
      child {node {right}
        child {node {child}}
        child {node {child}}
      };
\end{tikzpicture}
```

8.7 Grouping of Graphic Parameters

Graphic parameters should often apply to several path drawing or filling commands. For example, we may wish to draw numerous lines all with the same line width of 1pt. For this, we put these commands in a `{scope}` environment that takes the desired graphic options as an optional parameter. Naturally, the specified graphic parameters apply only to the drawing and filling commands inside the environment. Furthermore, nested `{scope}` environments or individual drawing commands can override the graphic parameters of outer `{scope}` environments. In the following example, three red lines, two green lines, and one blue line are drawn:



```
\begin{tikzpicture}
  \begin{scope}[color=red]
    \draw (0mm,10mm) -- (10mm,10mm);
    \draw (0mm, 8mm) -- (10mm, 8mm);
    \draw (0mm, 6mm) -- (10mm, 6mm);
  \end{scope}
  \begin{scope}[color=green]
    \draw             (0mm, 4mm) -- (10mm, 4mm);
    \draw             (0mm, 2mm) -- (10mm, 2mm);
    \draw[color=blue] (0mm, 0mm) -- (10mm, 0mm);
  \end{scope}
\end{tikzpicture}
```

The `{tikzpicture}` environment itself also behaves like a `{scope}` environment, that is, you can specify graphic parameters using an optional argument. These optional apply to all commands in the picture.

8.8 Coordinate Transformation System

TikZ relies entirely on PGF's *coordinate* transformation system to perform transformations. PGF also supports *canvas* transformations, a more low-level transformation system, but this system is not accessible from TikZ. There are two reasons for this: First, the canvas transformation must be used with great care and often results in "bad" graphics with changing line width and text in wrong sizes. Second, PGF looses track of where nodes and shapes are positioned when canvas transformations are used.

For more details on the difference between coordinate transformations and canvas transformations see Section 42.4.

9 Hierarchical Structures: Package, Environments, Scopes, and Styles

The present section explains how your files should be structured when you use TikZ. On the top level, you need to include the `tikz` package. In the main text, each graphic needs to be put in a `{tikzpicture}` environment. Inside these environments, you can use `{scope}` environments to create internal groups. Inside the scopes you use `\path` commands to actually draw something. On all levels (except for the package level), graphic options can be given that apply to everything within the environment.

9.1 Loading the Package and the Libraries

```
\usepackage{tikz} % LaTeX
\input tikz.tex   % plain TeX
\usemodule[tikz]  % ConTeXt
```

This package does not have any options.

This will automatically load the PGF package and some other stuff that TikZ needs (like the `xkeyval` package).

PGF needs to know what TeX driver you are intending to use. In most cases PGF is clever enough to determine the correct driver for you; this is true in particular if you LaTeX. Currently, the only situation where PGF cannot know the driver "by itself" is when you use plain TeX or ConTeXt together with `dvipdfm`. In this case, you have to write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` *before* you input `tikz.tex`.

`\usetikzlibrary{`⟨*list of libraries*⟩`}`

Once TikZ has been loaded, you can use this command to load further libraries. The list of libraries should contain the names of libraries separated by commas. Instead of curly braces, you can also use square brackets, which is something ConTeXt users will like. If you try to load a library a second time, nothing will happen.

Example: `\usetikzlibrary{arrows}`

The above command will load a whole bunch of extra arrow tip definitions.

What this command does is to load the file `pgflibrarytikz`⟨*library*⟩`.code.tex` for each ⟨*library*⟩ in the ⟨*list of libraries*⟩. Thus, to write your own library file, all you need to do is to place a file of the appropriate name somewhere where TeX can find it. LaTeX, plain TeX, and ConTeXt users can then use your library.

9.2 Creating a Picture

9.2.1 Creating a Picture Using an Environment

The "outermost" scope of TikZ is the `{tikzpicture}` environment. You may give drawing commands only inside this environment, giving them outside (as is possible in many other packages) will result in chaos.

In TikZ, the way graphics are rendered is strongly influenced by graphic options. For example, there is an option for setting the color used for drawing, another for setting the color used for filling, and also more obscure ones like the option for setting the prefix used in the filenames of temporary files written while plotting functions using an external program. The graphic options are nearly always specified in a so-called key-value style. (The "nearly always" refers to the name of nodes, which can also be specified differently.) All graphic options are local to the `{tikzpicture}` to which they apply.

`\begin{tikzpicture}[`⟨*options*⟩`]`
  ⟨*environment contents*⟩
`\end{tikzpicture}`

All TikZ commands should be given inside this environment, except for the `\tikzstyle` command. Unlike other packages, it is not possible to use, say, `\pgfpathmoveto` outside this environment and doing so will result in chaos. For TikZ, commands like `\path` are only defined inside this environment, so there is little chance that you will do something wrong here.

When this environment is encountered, the ⟨*options*⟩ are parsed. All options given here will apply to the whole picture.

Next, the contents of the environment is processed and the graphic commands therein are put into a box. Non-graphic text is suppressed as well as possible, but non-PGF commands inside a `{tikzpicture}` environment should not produce any "output" since this may totally scramble the positioning system of the backend drivers. The suppressing of normal text, by the way, is done by temporarily switching the font to `\nullfont`. You can, however, "escape back" to normal TeX typesetting. This happens, for example, when you specify a node.

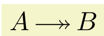
At the end of the environment, PGF tries to make a good guess at a good guess at the bounding box of the graphic and then resizes the box such that the box has this size. To "make its guess," everytime PGF encounters a coordinate, it updates the bound box's size such that it encompasses all these coordinates. This will usually give a good approximation at the bounding box, but will not always be accurate. First, the line thickness is not taken into account. Second, controls points of a curve often lie far "outside" the curve and make the bounding box too large. In this case, you should use the `[use as bounding box]` option.

The following option influences the baseline of the resulting picture:

- `baseline=`⟨*dimension or coordinate*⟩ Normally, the lower end of the picture is put on the baseline of the surrounding text. For example, when you give the code `\tikz\draw(0,0)circle(.5ex);`, PGF will find out that the lower end of the picture is at $-.5$ex and that the upper end is at .5ex. Then, the lower end will be put on the baseline, resulting in the following: ○.

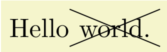
  Using this option, you can specify that the picture should be raised or lowered such that the height ⟨*dimension*⟩ is on the baseline. For example, `tikz[baseline=0pt]\draw(0,0)circle(.5ex);` yields ○ since, now, the baseline is on the height of the x-axis. If you omit the ⟨*dimensions*⟩, `0pt` is assumed as default.

  This options is often useful for "inlined" graphics as in

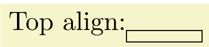
  


  ```
  $A \mathbin{\tikz[baseline] \draw[->>] (0pt,.5ex) -- (3ex,.5ex);} B$
  ```

  Instead of a ⟨*dimension*⟩ you can also provide a coordinate in parantheses. Then the effect is to put the baseline on the y-coordinate that the give ⟨*coordinate*⟩ has *at the end of the picture*. This means that, at the end of the picture, the ⟨*coordinate*⟩ is evaluated and then the baseline is set to the y-coordinate of the resulting point. This makes it easy to reference the y-coordinate of, say, the base line of nodes.


  ```
  Hello
  \tikz[baseline=(X.base)]
    \node [cross out,draw] (X) {world.};
  ```


  ```
  Top align:
  \tikz[baseline=(current bounding box.north)]
    \draw (0,0) rectangle (1cm,1ex);
  ```

- `execute at begin picture=`⟨*code*⟩ This option can be used to install some code that will be executed at the beginning of the picture. This option must be given in the argument of the `{tikzpicture}` environment itself since this option will not have an effect otherwise. After all, the picture has already "started" later on.

  This option is mainly used in styles like the `every picture` style to execute certain code at the start of a picture.

- `execute at end picture=`⟨*code*⟩ This option installs some code that will be executed at the end of the picture. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the `{tikzpicture}` environment.


  ```
  \begin{tikzpicture}[execute at end picture=%
    {
      \begin{pgfonlayer}{background}
        \path[fill=yellow,rounded corners]
          (current bounding box.south west) rectangle
          (current bounding box.north east);
      \end{pgfonlayer}
    }]
    \node at (0,0) {X};
    \node at (2,1) {Y};
  \end{tikzpicture}
  ```

All options "end" at the end of the picture. To set an option "globally" you can use the following style:

- `style=every picture` This style is installed at the beginning of each picture.

```
\tikzstyle{every picture}=[semithick]
```

In other TEX format, you should use instead the following commands:

`\tikzpicture[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\endtikzpicture`

This is the plain TEX version of the environment.

`\starttikzpicture[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\stoptikzpicture`

This is the ConTEXt version of the environment.

9.2.2 Creating a Picture Using a Command

The following two commands are used for "small" graphics.

`\tikz[`⟨*options*⟩`]{`⟨*commands*⟩`}`

This command places the ⟨*commands*⟩ inside a `{tikzpicture}` environment and adds a semicolon at the end. This is just a convenience.

The ⟨*commands*⟩ may not contain a paragraph (an empty line). This is a precaution to ensure that users really use this command only for small graphics.

Example: `\tikz{\draw (0,0) rectangle (2ex,1ex)}` yields ▭

`\tikz[`⟨*options*⟩`]`⟨*text*⟩`;`

If the ⟨*text*⟩ does not start with an opening brace, the end of the ⟨*text*⟩ is the next semicolon that is encountered.

Example: `\tikz \draw (0,0) rectangle (2ex,1ex);` yields ▭

9.2.3 Adding a Background

By default, pictures do not have any background, that is, they are "transparent" on all parts on which you do not draw anything. You may instead wish to have a colored background behind your picture or a black frame around it or lines above and below it or some other kind of decoration.

Since backgrounds are often not needed at all, the definition of styles for adding backgrounds has been put in the library package `pgflibrarytikzbackgrounds`. This package is documented in Section 20.

9.3 Using Scopes to Structure a Picture

Inside a `{tikzpicture}` environment you can create scopes using the `{scope}` environment. This environment is available only inside the `{tikzpicture}` environment, so once more, there is little chance of doing anything wrong.

`\begin{scope}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{scope}`

All ⟨*options*⟩ are local to the ⟨*environment contents*⟩. Furthermore, the clipping path is also local to the environment, that is, any clipping done inside the environment "ends" at its end.

```
\begin{tikzpicture}
  \begin{scope}[red]
    \draw (0mm,0mm) -- (10mm,0mm);
    \draw (0mm,1mm) -- (10mm,1mm);
  \end{scope}
  \draw (0mm,2mm) -- (10mm,2mm);
  \begin{scope}[green]
    \draw (0mm,3mm) -- (10mm,3mm);
    \draw (0mm,4mm) -- (10mm,4mm);
    \draw[blue] (0mm,5mm) -- (10mm,5mm);
  \end{scope}
\end{tikzpicture}
```

The following style influences scopes:

- `style=every scope` This style is installed at the beginning of every scope. I do not know really know what this might be good for, but who knows?

The following options are useful for scopes:

- `execute at begin scope=`⟨*code*⟩ This option install some code that will be executed at the beginning of the scope. This option must be given in the argument of the `{scope}` environment.

  The effect applies only to the current scope, not to subscopes.

- `execute at end scope=`⟨*code*⟩ This option installs some code that will be executed at the end of the current scope. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the `{scope}` environment.

  Again, the effect applies only to the current scope, not to subscopes.

`\scope[`⟨*options*⟩`]`
  ⟨*environment contents*⟩
`\endscope`

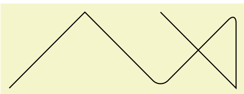
Plain TEX version of the environment.

`\startscope[`⟨*options*⟩`]`
  ⟨*environment contents*⟩
`\stopscope`

ConTEXt version of the environment.

9.4 Using Scopes Inside Paths

The `\path` command, which is described in much more detail in later sections, also takes graphic options. These options are local to the path. Furthermore, it is possible to create local scopes within a path simply by using curly braces as in



```
\tikz \draw (0,0) -- (1,1)
           {[rounded corners] -- (2,0) -- (3,1)}
           -- (3,0) -- (2,1);
```

Note that many options apply only to the path as a whole and cannot be scoped in this way. For example, it is not possible to scope the `color` of the path. See the explanations in the section on paths for more details.

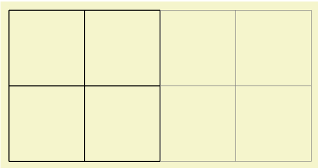
Finally, certain elements that you specify in the argument to the `\path` command also take local options. For example, a node specification takes options. In this case, the options apply only to the node, not to the surrounding path.

9.5 Using Styles to Manage How Pictures Look

There is a way of organizing sets of graphic options "orthogonally" to the normal scoping mechanism. For example, you might wish all your "help lines" to be drawn in a certain way like, say, gray and thin (do *not* dash them, that distracts). For this, you can use *styles*.

A style is simply a set of graphic options that is predefined at some point. Once a style has been defined, it can be used anywhere using the `style` option:

- style=⟨*style name*⟩ invokes all options that are currently set in the ⟨*style name*⟩. An example of a style is the predefined **help lines** style, which you should use for lines in the background like grid lines or construction lines. You can easily define new styles and modify existing ones.

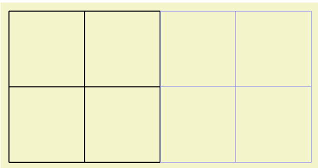


```
\begin{tikzpicture}
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

\tikzstyle⟨*style name*⟩+=[⟨*options*⟩]

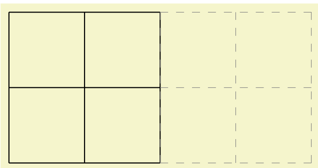
This command defines the style ⟨*style name*⟩. Whenever it is used using the style=⟨*style name*⟩ command, the ⟨*options*⟩ will be invoked. It is permissible that a style invokes another style using the style= command inside the ⟨*options*⟩, which allows you to build hierarchies of styles. Naturally, you should *not* create cyclic dependencies.

If the style already has a predefined meaning, it will unceremoniously be redefined without a warning.



```
\tikzstyle{help lines}=[blue!50,very thin]
\begin{tikzpicture}
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

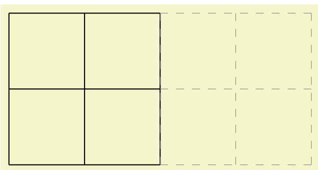
If the optional + is given, the options are *added* to the existing definition:



```
\tikzstyle{help lines}+=[dashed]% aaarghhh!!!
\begin{tikzpicture}
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

It is also possible to set a style using an option:

- set style={{⟨*style name*⟩}+=[⟨*options*⟩]} This option has the same effect as saying \tikzstyle before the argument of the option.



```
\begin{tikzpicture}[set style={{help lines}+=[dashed]}]
  \draw                   (0,0) grid +(2,2);
  \draw[style=help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

10 Specifying Coordinates

10.1 Overview

A *coordinate* is a position on the canvas on which your picture is drawn. Ti*k*Z uses a special syntax for specifying coordinates. Coordinates are always put in round brackets. The general syntax is `([`⟨*options*⟩`]`⟨*coordinate specification*⟩`)`.

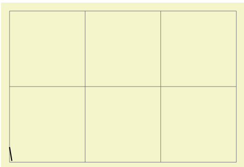
The ⟨*coordinate specification*⟩ specified coordinates using one of many different possible *coordinate systems*. Examples are the Cartesian coordinate system or polar coordinates or spherical coordinates. No matter which coordinate system is used, in the end, a specific point on the canvas is represented by the coordinate.

There are two ways of specifying which coordinate system should be used:

Explicitly You can specify the coordinate system explicitly. To do so, you give the name of the coordinate system at the beginning, followed by `cs:`, which stands for "coordinate system," followed by a specification of the coordinate using the key-value syntax. Thus, the general syntax for ⟨*coordinate specification*⟩ in the explicit case is `(`⟨*coordinate system*⟩ `cs:`⟨*list of key-value pairs specific to the coordinate system*⟩`)`.

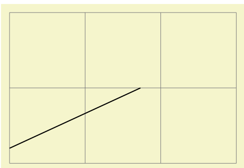
Implicitly The explicit specification is often too verbose when numerous coordinates should be given. Because of this, for the coordinate systems that you are likely to use often a special syntax is provided. Ti*k*Z will notice when you use a coordinate specified in a special syntax and will choose the correct coordinate system automatically.

Here is an example in which explicit the coordinate systems are specified explicitly:



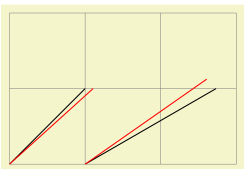
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw (canvas cs:x=0cm,y=2mm)
     -- (canvas polar cs:radius=2cm,angle=30);
\end{tikzpicture}
```

In the next example, the coordinate systems are implicit:



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw (0cm,2mm) -- (30:2cm);
\end{tikzpicture}
```

It is possible to give options that apply only to a single coordinate, although this makes sense for transformation options only. To give transformation options for a single coordinate, give these options at the beginning in brackets:



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (1,1);
  \draw[red] (0,0) -- ([xshift=3pt] 1,1);
  \draw      (1,0) -- +(30:2cm);
  \draw[red] (1,0) -- +([shift=(135:5pt)] 30:2cm);
\end{tikzpicture}
```

10.2 Coordinate Systems

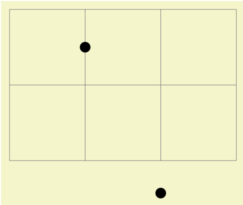
10.2.1 Canvas, XYZ, and Polar Coordinate Systems

Let us start with the basic coordinate systems.

Coordinate system `canvas`

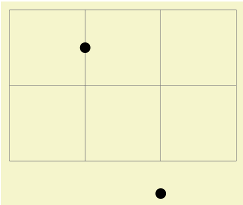
The simplest way of specifying a coordinate is to use the `canvas` coordinate system. You provide a dimension d_x using the `x=` option and another dimension d_y using the `y=` option. The position on the canvas is located at the position that is d_x to the right and d_y above the origin.

- x=⟨*dimension*⟩ Distance by which the coordinate is to the right of the origin. You can also write things like `1cm+2pt` since the `calc` package is used.
- y=⟨*dimension*⟩ Distance by which the coordinate is above the origin.



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);

  \fill (canvas cs:x=1cm,y=1.5cm)    circle (2pt);
  \fill (canvas cs:x=2cm,y=-5mm+2pt) circle (2pt);
\end{tikzpicture}
```

To specify a coordinate in the coordinate system implicitly, you use two dimensions that are seperated by a comma as in `(0cm,3pt)` or `(2cm,\textheight)`.



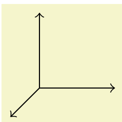
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);

  \fill (1cm,1.5cm)    circle (2pt);
  \fill (2cm,-5mm+2pt) circle (2pt);
\end{tikzpicture}
```

Coordinate system xyz

The `xyz` coordinate system allows you to specify a point as a multiple of three vectors called the x-, y-, and z-vectors. By default, the x-vector points 1cm to the right, the y-vector points 1cm upwards, but this can be changed arbitrarily as explained in Section 17.2. The default z-vector points to $\big(-\frac{1}{\sqrt{2}}\text{cm}, -\frac{1}{\sqrt{2}}\text{cm}\big)$.

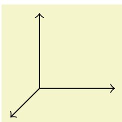
To specify the factors by which the vectors should be multiplied before being added, you use the following three options:

- x=⟨*factor*⟩ Factor by which the x-vector is multiplied. If this option is not given, `0` is used.
- y=⟨*factor*⟩ Works like `x`.
- z=⟨*factor*⟩ Works like `x`.



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (xyz cs:x=1);
  \draw (0,0) -- (xyz cs:y=1);
  \draw (0,0) -- (xyz cs:z=1);
\end{tikzpicture}
```

This coordinate system can also be selected implicitly. To do so, you just provide two or three comma-seperated factors (not dimensions).



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (1,0);
  \draw (0,0) -- (0,1,0);
  \draw (0,0) -- (0,0,1);
\end{tikzpicture}
```

Coordinate system canvas polar

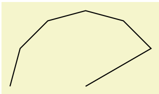
The `canvas polar` coordinate system allows you to specify polar coordinates. You provide an angle using the `angle=` option and a radius using the `radius=` option. This yields the point on the canvas that is at the given radius distance from the origin at the given degree. A degree of zero points to the right, a degree of 90 upward.

- angle=⟨*degrees*⟩ The angle of the coordinate. The angle must always be given in degrees and should be between −360 and 720.
- radius=⟨*dimension*⟩ The distance from the origin.

- x radius=⟨*dimension*⟩ A polar coordinate is, after all, just a point on a circle of the given ⟨*radius*⟩. When you provide an x-radius and also a y-radius, you specify an ellipse instead of a circle. The `radius` option has the same effect as specifiying identical `x radius` and `y radius` options.
- y radius=⟨*dimension*⟩ Works like `x radius`.

```
\tikz \draw (0,0) -- (canvas polar cs:angle=30,radius=1cm);
```

The implicit form for canvas polar coodinates is the following: you specify the angle and the distance, separated by a colon as in `(30:1cm)`.



```
\tikz \draw    (0cm,0cm) -- (30:1cm) -- (60:1cm) -- (90:1cm)
            -- (120:1cm) -- (150:1cm) -- (180:1cm);
```

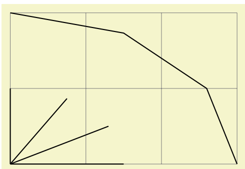
Two different radii are specified by writing `(30:1cm and 2cm)`.

For the implicit form, instead of an angle given as a number you can also use certain words. For example, `up` is the same as `90`, so that you can write `\tikz \draw (0,0) -- (2ex,0pt) -- +(up:1ex);` and get. Apart from `up` you can use `down`, `left`, `right`, `north`, `south`, `west`, `east`, `north east`, `north west`, `south east`, `south west`, all of which have their natural meaning.

Coordinate system xyz polar

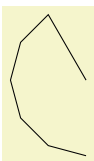
This coordinate system work similarly to the `canvas polar` system. However, the radius and the angle are interpreted in the xy-coordinate system, not in the canvas system. More detailedly, consider the circle or ellipse whose half axes are given by the current x-vector and the current y-vector. Then, consider the point that lies at a given angle on this ellipse, where an angle of zero is the same as the x-vector and an angle of 90 is the y-vector. Finally, multiply the resulting vector by the given radius factor. Voilà.

- angle=⟨*degrees*⟩ The angle of the coordinate interpreted in the ellipse whose axes are the x-vector and the y-vector.
- radius=⟨*factor*⟩ A factor by which the x-vector and y-vector are multiplied prior to forming the ellipse.
- x radius=⟨*dimension*⟩ A specific factor by which only the x-vector is multiplied.
- y radius=⟨*dimension*⟩ works like `x radius`.



```
\begin{tikzpicture}[x=1.5cm,y=1cm]
  \draw[help lines] (0cm,0cm) grid (3cm,2cm);

  \draw (0,0) -- (xyz polar cs:angle=0,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=30,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=60,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=90,radius=1);

  \draw (xyz polar cs:angle=0,radius=2)
     -- (xyz polar cs:angle=30,radius=2)
     -- (xyz polar cs:angle=60,radius=2)
     -- (xyz polar cs:angle=90,radius=2);
 \end{tikzpicture}
```

The implicit version of this option is the same as the implicit version of `canvas polar`, only you do not provide a unit.



```
\tikz[x={(0cm,1cm)},y={(-1cm,0cm)}]
  \draw  (0,0) -- (30:1) -- (60:1) -- (90:1)
            -- (120:1) -- (150:1) -- (180:1);
```

Coordinate system xy polar

This is just an alias for `xyz polar`, which some people might prefer as there is no x-coordinate involved in the `xyz polar` coordinates.

10.2.2 Barycentric Systems

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors $v_1, v_2, \ldots, v_n$ and numbers $\alpha_1, \alpha_2, \ldots, \alpha_n$. Then the barycentric coordinate specified by these vectors and numbers is

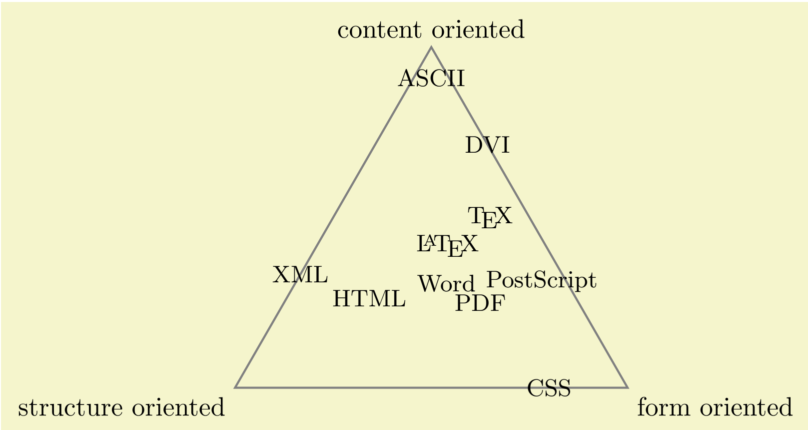
$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \cdots + \alpha_n}$$

The `barycentric cs` allows you to specifiy such coordiantes easily.

Coordinate system `barycentric`

For this coordinate system, the ⟨*coordinate specification*⟩ should be a comma-separated list of expressions of the form ⟨*node name*⟩=⟨*number*⟩. Note that (currently) the list should not contain any spaces before or after the ⟨*node name*⟩ (unlike normal key-value pairs).

The specified coordinate is now computed as follows: Each pair provides one vector and a number. The vector is the `center` anchor of the ⟨*node name*⟩. The number is the ⟨*number*⟩. Note that (currently) you cannot specify a different anchor, so that in order to use, say, the `north` anchor of a node you first have to create a new coordinate at this north anchor. (Using for instance `\coordinate(mynorth) at (mynode.north);`.)




```
\begin{tikzpicture}
  \coordinate (content)   at (90:3cm);
  \coordinate (structure) at (210:3cm);
  \coordinate (form)      at (-30:3cm);

  \node [above]       at (content)   {content oriented};
  \node [below left]  at (structure) {structure oriented};
  \node [below right] at (form)      {form oriented};

  \draw [thick,gray] (content.south) -- (structure.north east) -- (form.north west) -- cycle;

  \small
  \node at (barycentric cs:content=0.5,structure=0.1 ,form=1)    {PostScript};
  \node at (barycentric cs:content=1  ,structure=0   ,form=0.4)  {DVI};
  \node at (barycentric cs:content=0.5,structure=0.5 ,form=1)    {PDF};
  \node at (barycentric cs:content=0  ,structure=0.25,form=1)    {CSS};
  \node at (barycentric cs:content=0.5,structure=1   ,form=0)    {XML};
  \node at (barycentric cs:content=0.5,structure=1   ,form=0.4)  {HTML};
  \node at (barycentric cs:content=1  ,structure=0.2 ,form=0.8)  {\TeX};
  \node at (barycentric cs:content=1  ,structure=0.6 ,form=0.8)  {\LaTeX};
  \node at (barycentric cs:content=0.8,structure=0.8 ,form=1)    {Word};
  \node at (barycentric cs:content=1  ,structure=0.05,form=0.05) {ASCII};
\end{tikzpicture}
```

10.2.3 Node Coordinate System

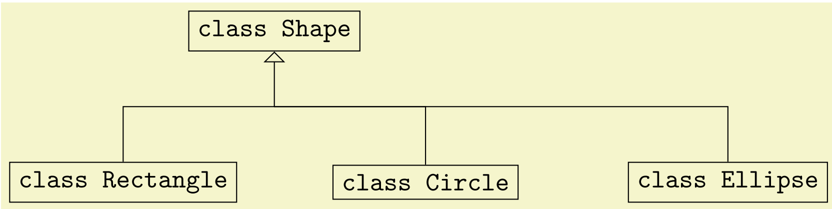
In PGF and in Ti*k*Z it is quite easy to define a node that you wish to reference at a later point. Once you have defined a node, there are different ways of referencing points of the node. To do so, you use the following coordinate system:

Coordinate system `node`

This coordinate system is used to reference a specific point inside or on the border of a previously defined node. It can be used in different ways, so let us go over them one by one.

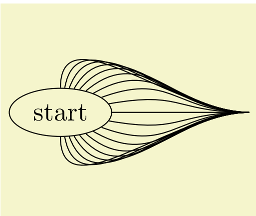
You can use three options to specify which coordinate you mean:

- `name=`⟨*node name*⟩ specifies the node in which you which to specify a coordinate. The ⟨*node name*⟩ is the name that was previously used to name the node using the `name=`⟨*node name*⟩ option or the special node name syntax.
- `anchor=`⟨*anchor*⟩ specifies an anchor of the node. Here is an example:



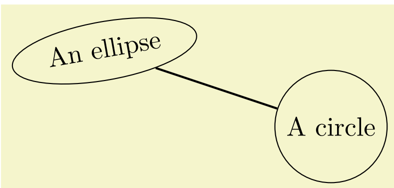

```
\begin{tikzpicture}
  \node (shape)   at (0,2)  [draw] {|class Shape|};
  \node (rect)    at (-2,0) [draw] {|class Rectangle|};
  \node (circle)  at (2,0)  [draw] {|class Circle|};
  \node (ellipse) at (6,0)  [draw] {|class Ellipse|};

  \draw (node cs:name=circle,anchor=north) |- (0,1);
  \draw (node cs:name=ellipse,anchor=north) |- (0,1);
  \draw[-open triangle 90] (node cs:name=rect,anchor=north)
        |- (0,1) -| (node cs:name=shape,anchor=south);
\end{tikzpicture}
```

- `angle=`⟨*degrees*⟩ It is also possible to provide an angle *instead* of an anchor. This coordinate refers to a point of the node's border where a ray shot from the center in the given angle hits the border. Here is an example:




```
\begin{tikzpicture}
  \node (start) [draw,shape=ellipse] {start};
  \foreach \angle in {-90, -80, ..., 90}
    \draw (node cs:name=start,angle=\angle)
      .. controls +(\angle:1cm) and +(-1,0) .. (2.5,0);
  \end{tikzpicture}
```

It is possible to provide *neither* the `anchor=` option nor the `angle=` option. In this case, Ti*k*Z will calculate an appropriate border position for you. Here is an example:



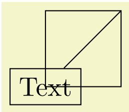

```
\begin{tikzpicture}
  \path (0,0)  node(a) [ellipse,rotate=10,draw] {An ellipse}
        (3,-1) node(b) [circle,draw]            {A circle};
  \draw[thick] (node cs:name=a) -- (node cs:name=b);
\end{tikzpicture}
```

Ti*k*Z will be reasonably clever at determining the border points that you "mean," but, naturally, this may fail in some situations. If Ti*k*Z fails to determine an appropriate border point, the center will be used instead.

Automatic computation of anchors works only with the line-to operations `--`, the vertical/horizontal versions `|-` and `-|`, and with the curve-to operation `..`. For other path commands, such as `parabola` or `plot`, the center will be used. If this is not desired, you should give a named anchor or an angle anchor.

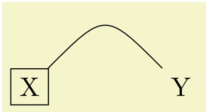
Note that if you use an automatic coordinate for both the start and the end of a line-to, as in `--(node cs:name=b)--`, then *two* border coordinates are computed with a move-to between them. This is usually exactly what you want.

If you use relative coordinates together with automatic anchor coordinates, the relative coordinates are computed relative to the node's center, not relative to the border point. Here is an example:




```
\tikz \draw (0,0) node(x) [draw] {Text}
            rectangle (1,1)
            (node cs:name=x) -- +(1,1);
```

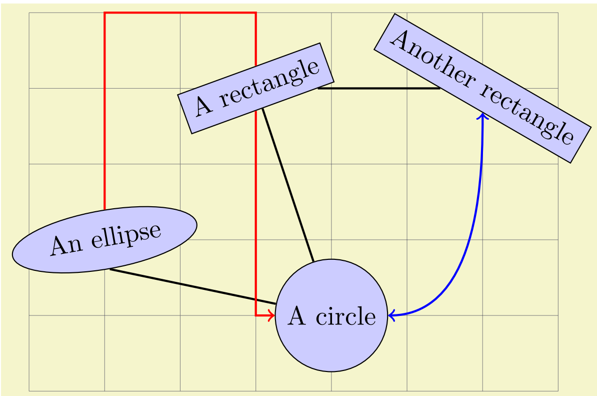
Similarly, in the following examples both control points are $(1, 1)$:




```
\tikz \draw (0,0) node(x) [draw] {X}
            (2,0) node(y) {Y}
            (node cs:name=x) .. controls +(1,1) and +(-1,1) ..
            (node cs:name=y);
```

The implicit way of specifying the node coordinate system is to simply use the name of the node in parentheses as in `(a)` or to specify a name together with an anchor or an angle separated by a dot as in `(a.north)` or `(a.10)`.

Here is a more complete example:




```
\begin{tikzpicture}[fill=blue!20]
  \draw[style=help lines] (-1,-2) grid (6,3);
  \path (0,0)  node(a) [ellipse,rotate=10,draw,fill]    {An ellipse}
        (3,-1) node(b) [circle,draw,fill]               {A circle}
        (2,2)  node(c) [rectangle,rotate=20,draw,fill]  {A rectangle}
        (5,2)  node(d) [rectangle,rotate=-30,draw,fill] {Another rectangle};
  \draw[thick] (a.south) -- (b) -- (c) -- (d);
  \draw[thick,red,->] (a) |- +(1,3) -| (c) |- (b);
  \draw[thick,blue,<->] (b) .. controls +(right:2cm) and +(down:1cm) .. (d);
\end{tikzpicture}
```

10.2.4 Intersection Coordinate Systems

Often you wish to specify a point that is on the intersection of two lines. For this, the following coordinate system is useful:

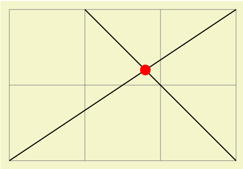
Coordinate system `intersection`

To specify the intersection of two line, you provide two lines using the following two options:

- `first line=(`⟨*first coordinate*⟩`)--(`⟨*second coordinate*⟩`)`
- `second line=(`⟨*first coordinate*⟩`)--(`⟨*second coordinate*⟩`)`

Note that you have to write `--` between the coordinate, but this does not mean that anything is added to the path. This is simply a special syntax.

The coordinate specified in this way is the intersection of the two lines. If the lines do not meet or if they are identical and arithmetical overflow error will result.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) coordinate (A) -- (3,2) coordinate (B)
        (1,2)                 -- (3,0);

  \fill[red] (intersection cs:
    first line={(A)--(B)},
    second line={(1,2)--(3,0)}) circle (2pt);
\end{tikzpicture}
```

The implicit way of specifying this coordinate system is to write `(intersection of` ⟨p_1⟩`--`⟨p_2⟩ `and` ⟨q_1⟩`--`⟨q_2⟩`)`. Note that there are *no* parentheses around the p_i and q_i. Thus, you would write `(intersection of A--B and 1,2--3,0)`.

A frequent special case of intersections is the intersection of a vertical line going through a point p and a horizontal line going through some other point q. For this situation there is another coordinate system.

Coordinate system `perpendicular`

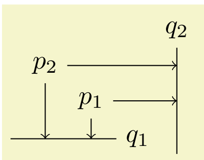
This coordinate system works the same way as `intersection`, only the lines are specified differently:

- `horizontal line through=(`⟨*coordinate*⟩`)` Specifies that one line is a horizontal line that goes through the given coordinate.
- `vertical line through=(`⟨*coordinate*⟩`)` Specifies that the other line is vertical and goes through the given coordinate.

The implicit syntax is to write `(`⟨p⟩ `|-` ⟨q⟩`)` or `(`⟨q⟩ `-|` ⟨p⟩`)`.

For example, `(2,1 |- 3,4)` and `(3,4 -| 2,1)` both yield the same as `(2,4)` (provided the xy-coordinate system has not been modified).

The most useful application of the syntax is to draw a line up to some point on a vertical or horizontal line. Here is an example:




```
\begin{tikzpicture}
  \path (30:1cm) node(p1) {$p_1$}   (75:1cm) node(p2) {$p_2$};

  \draw (-0.2,0) -- (1.2,0) node(xline)[right] {$q_1$};
  \draw (2,-0.2) -- (2,1.2) node(yline)[above] {$q_2$};

  \draw[->] (p1) -- (p1 |- xline);
  \draw[->] (p2) -- (p2 |- xline);
  \draw[->] (p1) -- (p1 -| yline);
  \draw[->] (p2) -- (p2 -| yline);
\end{tikzpicture}
```

10.2.5 Defining New Coordinate Systems

While the set of coordinate systems that TikZ can parse via their special syntax is fixed, it is possible and quite easy to define new explicitly named coordinate systems. For this, the following commands are used:

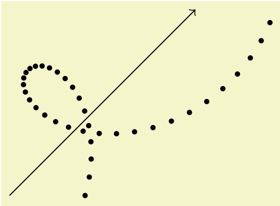
`\tikzdeclarecoordinatesystem{`⟨*name*⟩`}{`⟨*code*⟩`}`

This command declares a new coordinate system named ⟨*name*⟩ that can later on be used by writing `(`⟨*name*⟩ `cs:`⟨*arguments*⟩`)`. When TikZ encounters a coordinate specified in this way, the ⟨*arguments*⟩ are passed to ⟨*code*⟩ as argument `#1`.

It is now the job of ⟨*code*⟩ to make sense of the ⟨*arguments*⟩. At the end of ⟨*code*⟩, the two TEX dimensions `\pgf@x` and `\pgf@y` should be have the x- and y-canvas coordinate of the coordinate.

It is not necessary, but customary, to parse ⟨*arguments*⟩ using the key-value syntax. However, you can also parse it in any way you like.

In the following example, a coordinate system `cylindrical` is defined.



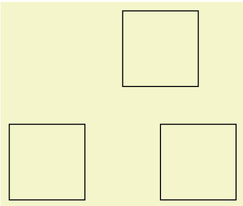
```
\makeatletter
\define@key{cylindricalkeys}{angle}{\def\myangle{#1}}
\define@key{cylindricalkeys}{radius}{\def\myradius{#1}}
\define@key{cylindricalkeys}{z}{\def\myz{#1}}
\tikzdeclarecoordinatesystem{cylindrical}%
{%
  \setkeys{cylindricalkeys}{#1}%
  \pgfpointadd{\pgfpointxyz{0}{0}{\myz}}{\pgfpointpolarxy{\myangle}{\myradius}}
}
\begin{tikzpicture}[z=0.2pt]
  \draw [->] (0,0,0) -- (0,0,350);
  \foreach \num in {0,10,...,350}
    \fill (cylindrical cs:angle=\num,radius=1,z=\num) circle (1pt);
\end{tikzpicture}
```

`\tikzaliascoordinatesystem{`⟨*new name*⟩`}{`⟨*old name*⟩`}`

Creates an alias of ⟨*old name*⟩.

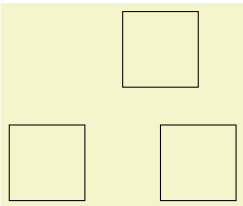
10.3 Relative and Incremental Coordinates

You can prefix coordinates by `++` to make them "relative." A coordinate such as `++(1cm,0pt)` means "1cm to the right of the previous position." Relative coordinates are often useful in "local" contexts:



```
\begin{tikzpicture}
  \draw (0,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (2,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (1.5,1.5) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
\end{tikzpicture}
```

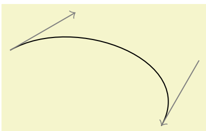
Instead of `++` you can also use a single `+`. This also specifies a relative coordinate, but it does not "update" the current point for subsequent usages of relative coordinates. Thus, you can use this notation to specify numerous points, all relative to the same "initial" point:



```
\begin{tikzpicture}
  \draw (0,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (2,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (1.5,1.5) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\end{tikzpicture}
```

There is one special situation, where relative coordinates are interpreted differently. If you use a relative coordinate as a control point of a Bézier curve, the following rule applies: First, a relative first control point is taken relative to the beginning of the curve. Second, a relative second control point is taken relative to the end of the curve. Third, a relative end point of a curve is taken relative to the start of the curve.

This special behavior makes it easy to specify that a curve should "leave or arrives from a certain direction" at the start or end. In the following example, the curve "leaves" at 30° and "arrives" at 60°:



```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

11 Syntax for Path Specifications

A *path* is a series of straight and curved line segments. It is specified following a `\path` command and the specification must follow a special syntax, which is described in the subsections of the present section.

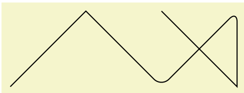
`\path`⟨*specification*⟩`;`

This command is available only inside a `{tikzpicture}` environment.

The ⟨*specification*⟩ is a long stream of *path operations*. Most of these path operations tell Ti*k*Z how the path is build. For example, when you write `--(0,0)`, you use a *line-to operation* and it means "continue the path from wherever you are to the origin."

At any point where Ti*k*Z expects a path operation, you can also give some graphic options, which is a list of options in brackets, such as `[rounded corners]`. These options can have different effects:

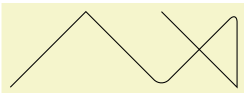
1. Some options take "immediate" effect and apply to all subsequent path operations on the path. For example, the `rounded corners` option will round all following corners, but not the corners "before" and if the `sharp corners` is given later on the path (in a new set of brackets), the rounding effect will end.



```
\tikz \draw (0,0) -- (1,1)
            [rounded corners] -- (2,0) -- (3,1)
            [sharp corners] -- (3,0) -- (2,1);
```

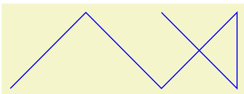
   Another example are the transformation options, which also apply only to subsequent coordinates.

2. The options that have immediate effect can be "scoped" by putting part of a path in curly braces. For example, the above example could also be written as follows:



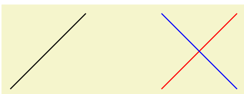
```
\tikz \draw (0,0) -- (1,1)
            {[rounded corners] -- (2,0) -- (3,1)}
            -- (3,0) -- (2,1);
```

3. Some options only apply to the path as a whole. For example, the `color=` option for determining the color used for, say, drawing the path always applies to all parts of the path. If several different colors are given for different parts of the path, only the last one (on the outermost scope) "wins":



```
\tikz \draw (0,0) -- (1,1)
            [color=red] -- (2,0) -- (3,1)
            [color=blue] -- (3,0) -- (2,1);
```

   Most options are of this type. In the above example, we would have had to "split up" the path into several `\path` commands:



```
\tikz{\draw (0,0) -- (1,1);
      \draw [color=red] (2,0) -- (3,1);
      \draw [color=blue] (3,0) -- (2,1);}
```

By default, the `\path` command does "nothing" with the path, it just "throws it away." Thus, if you write `\path(0,0)--(1,1);`, nothing is drawn in your picture. The only effect is that the area occupied by the picture is (possibly) enlarged so that the path fits inside the area. To actually "do" something with the path, an option like `draw` or `fill` must be given somewhere on the path. Commands like `\draw` do this implicitly.

Finally, it is also possible to give *node specifications* on a path. Such specifications can come at different locations, but they are always allowed when a normal path operation could follow. A node specification starts with `node`. Basically, the effect is to typeset the node's text as normal TeX text and to place it at the "current location" on the path. The details are explained in Section 13.

Note, however, that the nodes are *not* part of the path in any way. Rather, after everything has been done with the path what is specified by the path options (like filling and drawing the path due to a `fill` and a `draw` option somewhere in the ⟨*specification*⟩), the nodes are added in a post-processing step.

The following style influences scopes:

- `style=every path` This style is installed at the beginning of every path. This can be useful for (temporarily) adding, say, the `draw` option to everything in a scope.

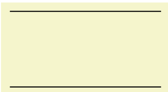
```
\begin{tikzpicture}[fill=examplefill] % only sets the color
  \tikzstyle{every path}=[draw]         % all paths are drawn
  \fill  (0,0) rectangle +(1,1);
  \shade (2,0) rectangle +(1,1);
\end{tikzpicture}
```

11.1 The Move-To Operation

The perhaps simplest operation is the move-to operation, which is specified by just giving a coordinate where a path operation is expected.

`\path` ... ⟨*coordinate*⟩ ... ;

The move-to operation normally starts a path at a certain point. This does not cause a line segment to be created, but it specifies the starting point of the next segment. If a path is already under construction, that is, if several segments have already been created, a move-to operation will start a new part of the path that is not connected to any of the previous segments.



```
\begin{tikzpicture}
  \draw (0,0) --(2,0) (0,1) --(2,1);
\end{tikzpicture}
```

In the specification `(0,0) --(2,0) (0,1) --(2,1)` two move-to operations are specified: `(0,0)` and `(0,1)`. The other two operations, namely `--(2,0)` and `--(2,1)` are line-to operations, described next.

11.2 The Line-To Operation

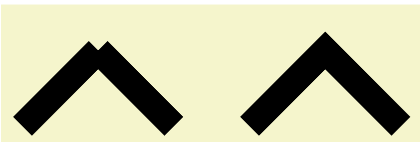
11.2.1 Straight Lines

`\path` ... `--`⟨*coordinate*⟩ ... ;

The line-to operation extends the current path from the current point in a straight line to the given coordinate. The "current point" is the endpoint of the previous drawing operation or the point specified by a prior move-to operation.

You use two minus signs followed by a coordinate in round brackets. You can add spaces before and after the `--`.

When a line-to operation is used and some path segment has just been constructed, for example by another line-to operation, the two line segments become joined. This means that if they are drawn, the point where they meet is "joined" smoothly. To appreciate the difference, consider the following two examples: In the left example, the path consists of two path segments that are not joined, but that happen to share a point, while in the right example a smooth join is shown.



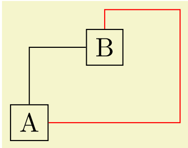
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) --(1,1)  (1,1) --(2,0);
  \draw (3,0) -- (4,1) -- (5,0);
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

11.2.2 Horizontal and Vertical Lines

Sometimes you want to connect two points via straight lines that are only horizontal and vertical. For this, you can use two path construction operations.

`\path ... -|⟨coordinate⟩ ...;`

This operation means "first horizontal, then vertical."




```
\begin{tikzpicture}
  \draw (0,0) node(a) [draw] {A}  (1,1) node(b) [draw] {B};
  \draw (a.north) |- (b.west);
  \draw[color=red] (a.east) -| (2,1.5) -| (b.north);
\end{tikzpicture}
```

`\path ... |-⟨coordinate⟩ ...;`

This operations means "first vertical, then horizontal."

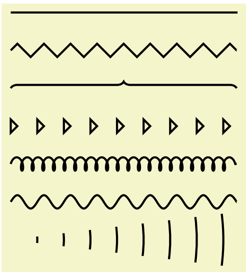
11.2.3 Snaked Lines

The line-to operation can not only be used to append straight lines to the path, but also "snaked" lines (called thus because they look a little bit like snakes seen from above).

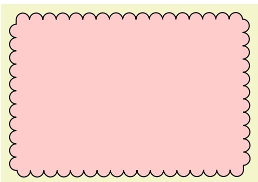
Ti*k*Z and PGF use a concept that I termed *snakes* for appending such "squiggly" lines. A snake specifies a way of extending a path between two points in a "fancy manner."

Normally, a snake will just connect the start point to the end point without starting new subpaths. Thus, a path containing a snaked line can, nevetheless, still be used for filling. However, this is not always the case. Some snakes consist of numerous unconnected segments. "Lines" consisting of such snakes cannot be used as the borders of enclosed areas.

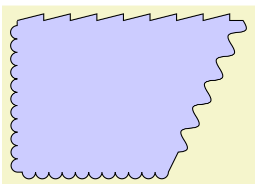
Here are some examples of snakes in action:



```
\begin{tikzpicture}[thick]
  \draw                                        (0,3)   -- (3,3);
  \draw[snake=zigzag]                          (0,2.5) -- (3,2.5);
  \draw[snake=brace]                           (0,2)   -- (3,2);
  \draw[snake=triangles]                       (0,1.5) -- (3,1.5);
  \draw[snake=coil,segment length=4pt]         (0,1)   -- (3,1);
  \draw[snake=coil,segment aspect=0]           (0,.5)  -- (3,.5);
  \draw[snake=expanding waves,segment angle=7] (0,0)   -- (3,0);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \filldraw[fill=red!20,snake=bumps] (0,0) rectangle (3,2);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \filldraw[fill=blue!20]              (0,3)
  [snake=saw]                       -- (3,3)
  [snake=coil,segment aspect=0]     -- (2,1)
  [snake=bumps]                     -| (0,3);
\end{tikzpicture}
```

No special path operation is needed to use a snake. Instead, you use the following option to "switch on" snaking:

- `snake=`⟨*snake name*⟩ This option causes the snake ⟨*snake name*⟩ to be used for subsequent line-to operations. So, whenever you use the `--` syntax to specify that a straight line should be added to the path, a snake to this path will be added instead. Snakes will also be used when you use the `-|` and `|-` syntax and also when you use the `rectangle` operation. Snakes will *not* be used when you use the curve-to operation nor when any other "curved" line is added to the path.

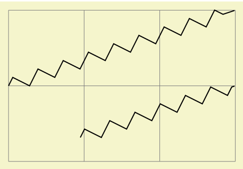
  This option has to be given anew for each path. However, you can also leave out the ⟨*snake name*⟩. In this case, the enclosing scope's ⟨*snake name*⟩ is used. Thus, you can specify a "standard" snake name for scope and then just say `\draw[snake]` every time this snake should actually be used.

The ⟨*snake name*⟩ `none` is special. It can be used to switch off snaking after it has been switched on on a path.

A bit strangely, no valid ⟨*snake names*⟩ are defined by Ti*k*Z by default. Instead, you have to include the library package `pgflibrarysnakes`. This package defines numerous snakes, see Section 31 for the complete list.

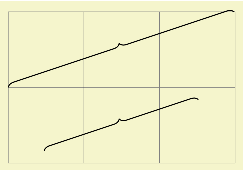
Most snakes can be configured. For example, for a snake that looks like a sine curve, you might wish to change the amplitude or the frequency. There are numerous options that influence these parameters. Not all options apply to all snakes, see Section 31 once more for details.

- `gap before snakes=`⟨*dimension*⟩ This option allows you to add a certain "gap" to the snake at its beginning. The snake will not start at the current point; instead the start point of the snake is move be ⟨*dimension*⟩ in the direction of the target.



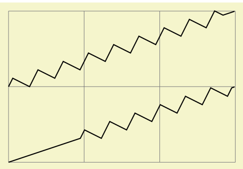
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw[snake=zigzag]                       (0,1) -- ++(3,1);
  \draw[snake=zigzag,gap before snake=1cm] (0,0) -- ++(3,1);
\end{tikzpicture}
```

- `gap after snake=`⟨*dimension*⟩ This option has the same effect as `gap before snake`, only it affects the end of the snake, which will "end early."
- `gap around snake=`⟨*dimension*⟩ This option sets the gap before and after the gap to ⟨*dimension*⟩.



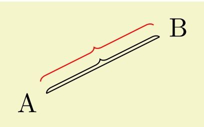
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw[snake=brace]                       (0,1) -- ++(3,1);
  \draw[snake=brace,gap around snake=5mm] (0,0) -- ++(3,1);
\end{tikzpicture}
```

- `line before snake=`⟨*dimension*⟩ This option works like `gap before snake`, only it will connect the current point with a straight line to the start of the snake.



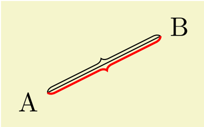
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw[snake=zigzag]                        (0,1) -- ++(3,1);
  \draw[snake=zigzag,line before snake=1cm] (0,0) -- ++(3,1);
\end{tikzpicture}
```

- `line after snake=`⟨*dimension*⟩ Works line `gap after snake`, only it adds a straight line.
- `line around snake=`⟨*dimension*⟩ Works line `gap around snake`, only it adds straight lines.
- `raise snake=`⟨*dimension*⟩ This option can be used with all snakes. It will offset the snake by "raising" it by ⟨*dimension*⟩. A negative ⟨*dimension*⟩ will lower the snake. Raising and lowering is always relative to the line along which the snake is drawn. Here is an example:




```
\begin{tikzpicture}
  \node (a) {A};
  \node (b) at (2,1) {B};
  \draw                                (a) -- (b);
  \draw[snake=brace]                   (a) -- (b);
  \draw[snake=brace,raise snake=5pt,red] (a) -- (b);
\end{tikzpicture}
```

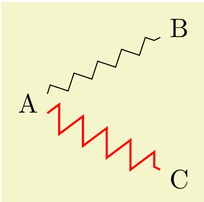
- `mirror snake` This option causes the snake to be "reflected along the path." This is best understood by looking at an example:




```
\begin{tikzpicture}
  \node (a) {A};
  \node (b) at (2,1) {B};
  \draw                                  (a) -- (b);
  \draw[snake=brace]                     (a) -- (b);
  \draw[snake=brace,mirror snake,red,thick] (a) -- (b);
\end{tikzpicture}
```

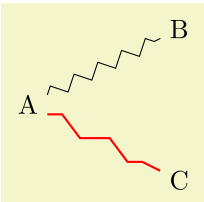
  This option can be used with every snake and can be combined with the `raise snake` option.

- `segment amplitude=`⟨*dimension*⟩ This option sets the "amplitude" of the snake. For a snake that is a sine wave this would be the amplitude of this line. For other snakes this value typically describes how far the snakes "rises above" or "falls below" the path. For some snakes, this value is ignored.

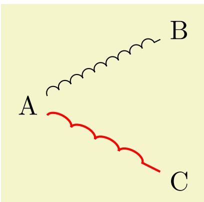



```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=zigzag]                                    (a) -- (b);
  \draw[snake=zigzag,segment amplitude=5pt,red,thick] (a) -- (c);
\end{tikzpicture}
```

- `segment length=`⟨*dimension*⟩ This option sets the length of each "segment" of a snake. For a sine wave this would be the wave length, for other snakes it is the length of each "repetitive part" of the snake.

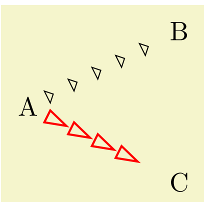



```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=zigzag]                                 (a) -- (b);
  \draw[snake=zigzag,segment length=20pt,red,thick] (a) -- (c);
\end{tikzpicture}
```



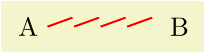

```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=bumps]                                 (a) -- (b);
  \draw[snake=bumps,segment length=20pt,red,thick] (a) -- (c);
\end{tikzpicture}
```

- `segment object length=`⟨*dimension*⟩ This option sets the length of the objects inside each segment of a snake. This option is only used for snakes in which each segment contains an object like a triangle or a star.

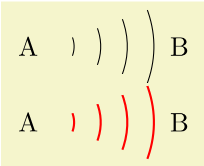



```
\begin{tikzpicture}
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=triangles]                                        (a) -- (b);
  \draw[snake=triangles,segment object length=8pt,red,thick] (a) -- (c);
\end{tikzpicture}
```

- `segment angle=`⟨*degrees*⟩ This option sets an angle that is interpreted in a snake-specific way. For example, the `waves` and `expanding waves` snakes interpret this as (half the) opening angle of the wave. The `border` snake uses this value for the angle of the little ticks.

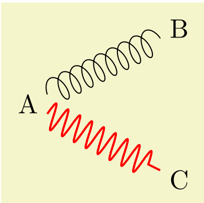



```
\begin{tikzpicture}[segment amplitude=10pt]
  \node (a) {A}   node (b) at (2,0) {B};
  \draw[snake=border]                             (a) -- (b);
  \draw[snake=border,segment angle=20,red,thick] (a) -- (b);
\end{tikzpicture}
```




```
\begin{tikzpicture}[segment amplitude=10pt]
  \node (a)           {A}   node (b)  at (2,0)  {B};
  \node (a') at (0,-1) {A}   node (b') at (2,-1) {B};
  \draw[snake=expanding waves]                             (a)  -- (b);
  \draw[snake=expanding waves,segment angle=20,red,thick] (a') -- (b');
\end{tikzpicture}
```

- segment aspect=⟨*ratio*⟩ This option sets an aspect ratio that is interpreted in a snake-specific way. For example, for the `coils` snake this describes the "direction" from which the coil is viewed.




```
\begin{tikzpicture}[segment amplitude=5pt,segment length=5pt]
  \node (a) {A}   node (b) at (2,1) {B}  node (c) at (2,-1) {C};
  \draw[snake=coil]                              (a) -- (b);
  \draw[snake=coil,segment aspect=0,red,thick]  (a) -- (c);
\end{tikzpicture}
```

It is possible to define new snakes, but this cannot be done inside TikZ. You need to use the command `\pgfdeclaresnake` from the basic level directly, see Section 46.

The following styles define combinations of segment settings that may be useful:

- style=snake triangles 45 Installs a snake the consists of little triangles with an opening angle of 45°.
- style=snake triangles 60 Installs a snake the consists of little triangles with an opening angle of 60°.
- style=snake triangles 90 Installs a snake the consists of little triangles with an opening angle of 90°.

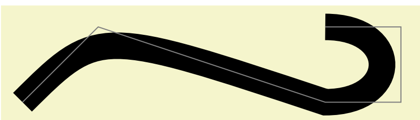
11.3 The Curve-To Operation

The curve-to operation allows you to extend a path using a Bézier curve.

`\path ... ..controls`⟨*c*⟩`and`⟨*d*⟩`..`⟨*y*⟩` ...;`

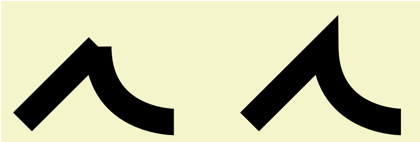
This operation extends the current path from the current point, let us call it x, via a curve to a the current point y. The curve is a cubic Bézier curve. For such a curve, apart from y, you also specify two control points c and d. The idea is that the curve starts at x, "heading" in the direction of c. Mathematically spoken, the tangent of the curve at x goes through c. Similarly, the curve ends at y, "coming from" the other control point, d. The larger the distance between x and c and between d and y, the larger the curve will be.

If the "`and`⟨*d*⟩" part is not given, d is assumed to be equal to c.



```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) .. controls (1,1) .. (4,0)
                              .. controls (5,0) and (5,1) .. (4,1);
  \draw[color=gray] (0,0) -- (1,1) -- (4,0) -- (5,0) -- (5,1) -- (4,1);
\end{tikzpicture}
```

As with the line-to operation, it makes a difference whether two curves are joined because they resulted from consecutive curve-to or line-to operations, or whether they just happen to have the same ending:



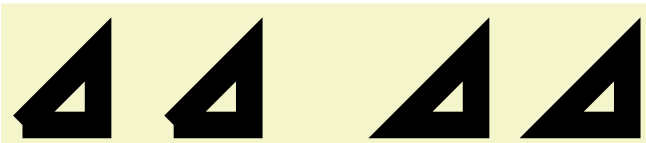
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
  \draw (3,0) -- (4,1) .. controls (4,0) and (5,0) .. (5,0);
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

11.4 The Cycle Operation

\path ... **--cycle** ...;

This operation adds a straight line from the current point to the last point specified by a move-to operation. Note that this need not be the beginning of the path. Furthermore, a smooth join is created between the first segment created after the last move-to operation and the straight line appended by the cycle operation.

Consider the following example. In the left example, two triangles are created using three straight lines, but they are not joined at the ends. In the second example cycle operations are used.



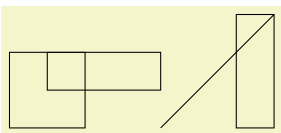
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) -- (1,0) -- (0,0) (2,0) -- (3,1) -- (3,0) -- (2,0);
  \draw (5,0) -- (6,1) -- (6,0) -- cycle (7,0) -- (8,1) -- (8,0) -- cycle;
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

11.5 The Rectangle Operation

A rectangle can obviously be created using four straight lines and a cycle operation. However, since rectangles are needed so often, a special syntax is available for them.

\path ... **rectangle**⟨*corner*⟩ ...;

When this operation is used, one corner will be the current point, another corner is given by ⟨*corner*⟩, which becomes the new current point.

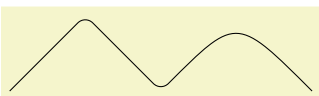


```
\begin{tikzpicture}
  \draw (0,0) rectangle (1,1);
  \draw (.5,1) rectangle (2,0.5) (3,0) rectangle (3.5,1.5) -- (2,0);
\end{tikzpicture}
```

11.6 Rounding Corners

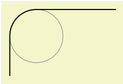
All of the path construction operations mentioned up to now are influenced by the following option:

- **rounded corners**=⟨*inset*⟩ When this option is in force, all corners (places where a line is continued either via line-to or a curve-to operation) are replaced by little arcs so that the corner becomes smooth.



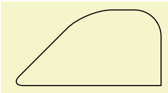
```
\tikz \draw [rounded corners] (0,0) -- (1,1)
          -- (2,0) .. controls (3,1) .. (4,0);
```

The ⟨*inset*⟩ describes how big the corner is. Note that the ⟨*inset*⟩ is *not* scaled along if you use a scaling option like `scale=2`.



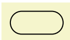
```
\begin{tikzpicture}
  \draw[color=gray,very thin] (10pt,15pt) circle (10pt);
  \draw[rounded corners=10pt] (0,0) -- (0pt,25pt) -- (40pt,25pt);
\end{tikzpicture}
```

You can switch the rounded corners on and off "in the middle of path" and different corners in the same path can have different corner radii:



```
\begin{tikzpicture}
  \draw (0,0) [rounded corners=10pt] -- (1,1) -- (2,1)
                         [sharp corners] -- (2,0)
                   [rounded corners=5pt] -- cycle;
\end{tikzpicture}
```

Here is a rectangle with rounded corners:



```
\tikz \draw[rounded corners=1ex] (0,0) rectangle (20pt,2ex);
```

You should be aware, that there are several pitfalls when using this option. First, the rounded corner will only be an arc (part of a circle) if the angle is 90°. In other cases, the rounded corner will still be round, but "not as nice."

Second, if there are very short line segments in a path, the "rounding" may cause inadverted effects. In such case it may be necessary to temporarily switch off the rounding using `sharp corners`.

- `sharp corners` This options switches off any rounding on subsequent corners of the path.

11.7 The Circle and Ellipse Operations

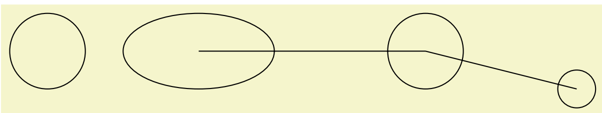
A circle can be approximated well using four Bézier curves. However, it is difficult to do so correctly. For this reason, a special syntax is available for adding such an approximation of a circle to the current path.

`\path ... circle(`⟨*radius*⟩`) ...;`

The center of the circle is given by the current point. The new current point of the path will remain to be the center of the circle.

`\path ... ellipse(`⟨*half width*⟩ `and` ⟨*half height*⟩`) ...;`

Note that you can add spaces after `ellipse`, but you have to place spaces around `and`.



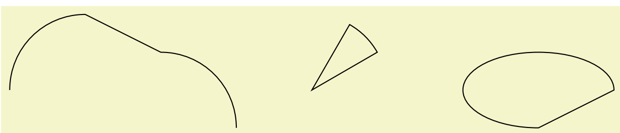
```
\begin{tikzpicture}
  \draw (1,0) circle (.5cm);
  \draw (3,0) ellipse (1cm and .5cm) -- ++(3,0) circle (.5cm)
    -- ++(2,-.5) circle (.25cm);
\end{tikzpicture}
```

11.8 The Arc Operation

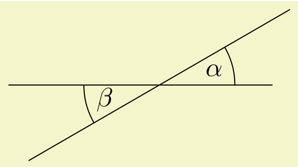
The *arc operation* allows you to add an arc to the current path.

`\path ... arc(`⟨*start angle*⟩`:`⟨*end angle*⟩`:`⟨*radius*⟩ `and` ⟨*half height*⟩`) ...;`

The arc operation adds a part of a circle of the given radius between the given angles. The arc will start at the current point and will end at the end of the arc.



```
\begin{tikzpicture}
  \draw (0,0) arc (180:90:1cm) -- (2,.5) arc (90:0:1cm);
  \draw (4,0) -- +(30:1cm) arc (30:60:1cm) -- cycle;
  \draw (8,0) arc (0:270:1cm and .5cm) -- cycle;
\end{tikzpicture}
```



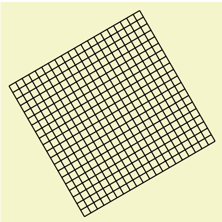

```
\begin{tikzpicture}
  \draw (-1,0) -- +(3.5,0);
  \draw (1,0) ++(210:2cm) -- +(30:4cm);
  \draw (1,0) +(0:1cm) arc (0:30:1cm);
  \draw (1,0) +(180:1cm) arc (180:210:1cm);
  \path (1,0) ++(15:.75cm) node{$\alpha$};
  \path (1,0) ++(15:-.75cm) node{$\beta$};
\end{tikzpicture}
```

11.9 The Grid Operation

You can add a grid to the current path using the `grid` path operation.

`\path` ... `grid[`⟨*options*⟩`]`⟨*corner*⟩ ... ;

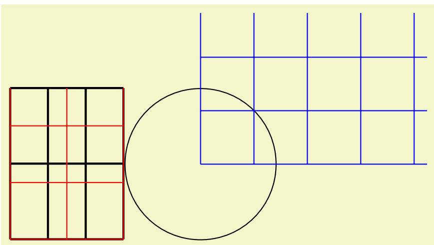
This operations adss a grid filling a rectangle whose two corners are given by ⟨*corner*⟩ and by the previous coordinate. Thus, the typical way in which a grid is drawn is `\draw (1,1) grid (3,3);`, which yields a grid filling the rectangle whose corners are at $(1,1)$ and $(3,3)$. All coordinate transformations apply to the grid.



```
\tikz[rotate=30] \draw[step=1mm] (0,0) grid (2,2);
```

The ⟨*options*⟩, which are local to the `grid` operation, can be used to influence the appearance of the grid. The stepping of the grid is governed by the following options:

- `step=`⟨*number or dimension or coordinate*⟩ sets the stepping in both the x and y-direction. If a dimension is provided, this is used directly. If a number is provided, this number is interpreted in the xy-coordinate system. For example, if you provide the number `2`, then the x-step is twice the x-vector and the y-step is twice the y-vector set by the `x=` and `y=` options. Finally, if you provide a coordinate, then the x-part of this coordinate will be used as the x-step and the y-part will be used as the y-coordinate.

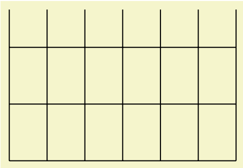


```
\begin{tikzpicture}[x=.5cm]
  \draw[thick] (0,0) grid [step=1]     (3,2);
  \draw[red]   (0,0) grid [step=.75cm] (3,2);
\end{tikzpicture}
\begin{tikzpicture}
  \draw       (0,0) circle (1);
  \draw[blue] (0,0) grid [step=(45:1)] (3,2);
\end{tikzpicture}
```

A complication arises when the x- and/or y-vector do not point along the axes. Because of this, the actual rule for computing the x-step and the y-step is the following: As the

x- and y-steps we use the x- and y-components or the following two vectors: The first vector is either $(\langle$*x-grid-step-number*$\rangle, 0)$ or $(\langle$*x-grid-step-dimension*$\rangle, 0\text{pt})$, the second vector is $(0, \langle$*y-grid-step-number*$\rangle)$ or $(0\text{pt}, \langle$*x-grid-step-dimension*$\rangle)$.

- `xstep=`⟨*dimension or number*⟩ sets the stepping in the x-direction.



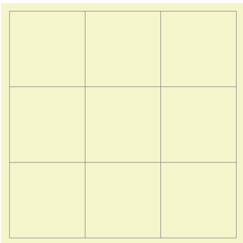
```
\tikz \draw (0,0) grid [xstep=.5,ystep=.75] (3,2);
```

- `ystep=`⟨*dimension or number*⟩ sets the stepping in the y-direction.

It is important to note that the grid is always "phased" such that it contains the point $(0, 0)$ if that point happens to be inside the rectangle. Thus, the grid does *not* always have an intersection at the corner points; this occurs only if the corner points are multiples of the stepping. Note that due to rounding errors, the "last" lines of a grid may be omitted. In this case, you have to add an epsilon to the corner points.

The following style is useful for drawing grids:

- `style=help lines` This style makes lines "subdued" by using thin gray lines for them. However, this style is not installed automatically and you have to say for example:



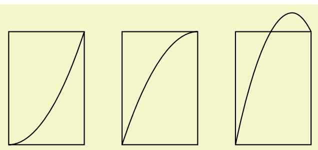
```
\tikz \draw[style=help lines] (0,0) grid (3,3);
```

11.10 The Parabola Operation

The `parabola` path operation continues the current path with a parabola. A parabola is a (shifted and scaled) curve defined by the equation $f(x) = x^2$ and looks like this: $\cup$.

`\path` ... `parabola[`⟨*options*⟩`]bend`⟨*bend coordinate*⟩⟨*coordinate*⟩ ... ;

This operation adds a parabola through the current point and the given ⟨*coordinate*⟩. If the `bend` is given, it specifies where the bend should go; the ⟨*options*⟩ can also be used to specify where the bend is. By default, the bend is at the old current point.



```
\begin{tikzpicture}
  \draw               (0,0) rectangle              (1,1.5)
                      (0,0) parabola               (1,1.5);
  \draw[xshift=1.5cm] (0,0) rectangle              (1,1.5)
                      (0,0) parabola[bend at end]  (1,1.5);
  \draw[xshift=3cm]   (0,0) rectangle              (1,1.5)
                      (0,0) parabola bend (.75,1.75) (1,1.5);
\end{tikzpicture}
```

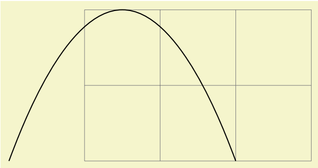
The following options influence parabolas:

- `bend=`⟨*coordinate*⟩ Has the same effect as saying `bend`⟨*coordinate*⟩ outside the ⟨*options*⟩. The option specifies that the bend of the parabola should be at the given ⟨*coordinate*⟩. You have to take care yourself that the bend position is a "valid" position; which means that if there is no parabola of the form $f(x) = ax^2 + bx + c$ that goes through the old current point, the given bend, and the new current point, the result will not be a parabola.

  There is one special property of the ⟨*coordinate*⟩: When a relative coordinate is given like `+(0,0)`, the position relative to which this coordinate is "flexible." More precisely, this position lies somewhere on a line from the old current point to the new current point. The exact position depends on the next option.

- `bend pos`=⟨*fraction*⟩ Specifies where the "previous" point is relative to which the bend is calculated. The previous point will be at the ⟨*fraction*⟩th part of the line from the old current point to the new current point.

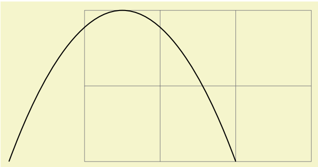
  The idea is the following: If you say `bend pos=0` and `bend +(0,0)`, the bend will be at the old current point. If you say `bend pos=1` and `bend +(0,0)`, the bend will be at the new current point. If you say `bend pos=0.5` and `bend +(0,2cm)` the bend will be 2cm above the middle of the line between the start and end point. This is most useful in situations such as the following:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[bend pos=0.5] bend +(0,2) +(3,0);
\end{tikzpicture}
```

  In the above example, the `bend +(0,2)` essentially means "a parabola that is 2cm high" and `+(3,0)` means "and 3cm wide." Since this situation arises often, there is a special shortcut option:

- `parabola height`=⟨*dimension*⟩ This option has the same effect as if you had written the following instead: `[bend pos=0.5,bend={+(0pt,`⟨*dimension*⟩`)}]`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[parabola height=2cm] +(3,0);
\end{tikzpicture}
```

The following styles are useful shortcuts:

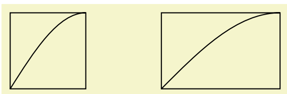
- `style=bend at start` This places the bend at the start of a parabola. It is a shortcut for the following options: `bend pos=0,bend={+(0,0)}`.
- `style=bend at end` This places the bend at the end of a parabola.

11.11 The Sine and Cosine Operation

The `sin` and `cos` operations are similar to the `parabola` operation. They, too, can be used to draw (parts of) a sine or cosine curve.

\path ... `sin`⟨*coordinate*⟩ ...;

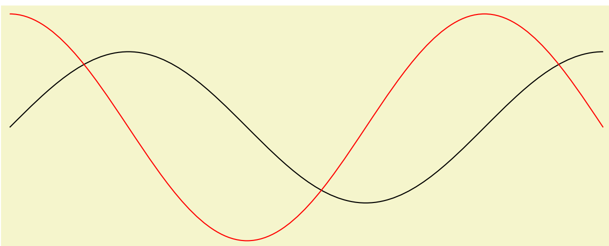
The effect of `sin` is to draw a scaled and shifted version of a sine curve in the interval $[0, \pi/2]$. The scaling and shifting is done in such a way that the start of the sine curve in the interval is at the old current point and that the end of the curve in the interval is at ⟨*coordinate*⟩. Here is an example that should clarify this:



```
\tikz \draw (0,0) rectangle (1,1)     (0,0) sin (1,1)
            (2,0) rectangle +(1.57,1) (2,0) sin +(1.57,1);
```

\path ... `cos`⟨*coordinate*⟩ ...;

This operation works similarly, only a cosine in the interval $[0, \pi/2]$ is drawn. By correctly alternating `sin` and `cos` operations, you can create a complete sine or cosine curve:



```
\begin{tikzpicture}[xscale=1.57]
  \draw (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0) sin (5,1);
  \draw[color=red] (0,1.5) cos (1,0) sin (2,-1.5) cos (3,0) sin (4,1.5) cos (5,0);
\end{tikzpicture}
```

Note that there is no way to (conveniently) draw an interval on a sine or cosine curve whose end points are not multiples of $\pi/2$.

11.12 The Plot Operation

The `plot` operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

Since the syntax and the behaviour of this command are a bit complex, they are described in the separated Section 16.

11.13 The To Path Operation

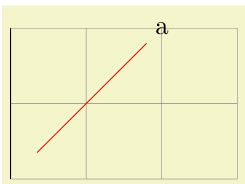
The `to` operation is used to add a user-defined path from the previous coordinate to the following coordinate. When you write `(a) to (b)`, a straight line is added from `a` to `b`, exactly as if you had written `(a) -- (b)`. However, if you write `(a) to [out=135,in=45] (b)` a curve is added to the path, which leaves at an angle of 135° at `a` and arrives at an angle of 45° at `b`. This is because the options `in` and `out` trigger a special path to be used instead of the straight line.

`\path ... to[`⟨*options*⟩`]` ⟨*nodes*⟩ `(`⟨*coordinate*⟩`) ...;`

This path operation inserts the path current set via the `to path` option at the current position. The ⟨*options*⟩ can be used to modify (perhaps implicitly) the `to path` and to setup how the path will be rendered.

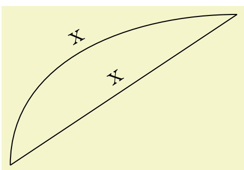
Before the `to path` is inserted, a number of macros are setup that can "help" the `to path`. These are `\tikztostart`, `\tikztotarget`, and `\tikztonodes`; they are explained in the following.

Start and Target Coordinates. The `to` operation is always followed by a ⟨*coordinate*⟩, called the target coordinate. The macro `\tikztotarget` is set to this coordinate (without the parantheses). There is also a *start coordinate*, which is the coordinate preceding the `to` operation. This coordinate can be accessed via the macro `\tikztostart`. In the following example, for the first `to`, the macro `\tikztostart` is `0pt,0pt` and the `\tikztotarget` is `0,2`. For the second `to`, the macro `\tikztostart` is `10pt,10pt` and `\tikztotarget` is `a`.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \draw      (0,0)       to (0,2);
  \node      (a)         at (2,2) {a};
  \draw[red] (10pt,10pt) to (a);
\end{tikzpicture}
```

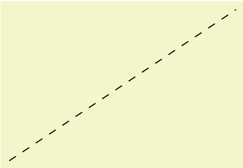
Nodes on tos. It is possible to add nodes to the paths constructed by a `to` operation. To do so, you specify the nodes between the `to` keyword and the coordinate (if there are options to the `to` operation, these come first). The effect of `(a) to node {x} (b)` (typically) is the same as if you had written `(a) -- node {x} (b)`, namely that the node is placed on the to. This can be used to add labels to tos:




```
\begin{tikzpicture}
  \draw (0,0) to node [sloped,above] {x} (3,2);

  \draw (0,0) to[out=90,in=180] node [sloped,above] {x} (3,2);
\end{tikzpicture}
```

Styles for nodes. In addition to the ⟨*options*⟩ given after the `to` operation, the following style is also set at the beginning of the to path:

- `style=every to` This style is installed at the beginning of every to. By default, it is set to `draw`.



```
\begin{tikzpicture}
  \tikzstyle{every to}=[draw,dashed]
  \path (0,0) to (3,2);
\end{tikzpicture}
```

Options. The ⟨*options*⟩ given with the `to` allow you to influence the appearance of the `to path`. Mostly, these options are used to change the `to path`. This can be used to change the path from a straight line to, say, a curve.

The path used is set using the following option:

- `to path=`⟨*path*⟩ Whenever an `to` operation is used, the ⟨*path*⟩ is inserted. More precisely, the following path is added:

  `[every to,`⟨*options*⟩`]` ⟨*path*⟩

  The ⟨*options*⟩ are the options given to the `to` operation, the ⟨*path*⟩ is the path set by this option `to path`.

  Inside the ⟨*path*⟩, different macros are used to reference the from- and to-coordinates. In detail, these are:

  – `\tikztostart` will expand to the from-coordinate (without the parantheses).
  – `\tikztotarget` will expand to the to-coordinate.
  – `\tikztonodes` will expand to the nodes between the `to` operation and the coordinate. Furthermore, these nodes will have the `pos` option set implicitly.

  Let us have a look at a simple example. The standard straight line for an to is achieved by the following ⟨*path*⟩:

  `-- (\tikztotarget) \tikztonodes`

  Indeed, this is the default setting for the path. When we write `(a) to (b)`, the ⟨*path*⟩ will expand to `(a) -- (b)`, when we write

  `(a) to[red] node {x} (b)`

  the ⟨*path*⟩ will expand to

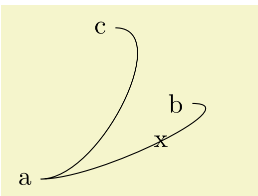
  `(a) -- (b) node[pos] {x}`

  It is not possible to specify the path

  `-- \tikztonodes (\tikztotarget)`

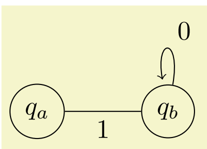
  since TikZ does not allow one to have a macro after `--` that expands to a node.

  Now let us have a look at how we can modify the ⟨*path*⟩ sensibly. The simplest way is to use a curve.




```
\begin{tikzpicture}[to path={
    .. controls +(1,0) and +(1,0) .. (\tikztotarget) \tikztonodes}]

  \node (a) at (0,0) {a};
  \node (b) at (2,1) {b};
  \node (c) at (1,2) {c};

  \draw (a) to node {x} (b)
        (a) to          (c);
\end{tikzpicture}
```

  Here is another example:




```
\tikzstyle{my loop}=[->,to path={
  .. controls +(80:1) and +(100:1) .. (\tikztotarget) \tikztonodes}]
\tikzstyle{my state}=          [circle,draw]

\begin{tikzpicture}[shorten >=2pt]
  \node [my state] (a) at (210:1) {$q_a$};
  \node [my state] (b) at (330:1) {$q_b$};

  \draw (a) to           node[below]       {1} (b)
            to [my loop] node[above right] {0} (b);
\end{tikzpicture}
```

- execute at begin to=⟨*code*⟩ The ⟨*code*⟩ is executed prior to the to. This can be used to draw one or more additional paths or to do additional computations.
- executed at end to=⟨*code*⟩ Works like the previous option, only this code is executed after the to path has been added.
- style=every to This style is installed at the beginning of every to. It is empty by default.

There are a number of predefined `to paths`, see Section 32 for a reference.

11.14 The Scoping Operation

When Ti*k*Z encounters and opening or a closing brace (`{` or `}`) at some point where a path operation should come, it will open or close a scope. All options that can be applied "locally" will be scoped inside the scope. For example, if you apply a transformation like `[xshift=1cm]` inside the scoped area, the shifting only applies to the scope. On the other hand, an option like `color=red` does not have any effect inside a scope since it can only be applied to the path as a whole.

11.15 The Node Operation

There are teo more operations that can be found in paths: `node` and `edge`. The first is used to add a so-called node to a path. This operation is special in the following sense: It does not change the current path in any way. In other words, this operation is not really a path operation, but has an effect that is "external" to the path. The `edge` operation has similar effect in that it adds something *after* the main parth has been drawn. However, it works like the `to` operation, that is, it adds a `to` path to the picture after the main path has been drawn.

Since these operations are quite complex, they are described in the separate Section 13.

11.16 The PGF-Extra Operation

In some cases you may need to "do some calculations or some other stuff" while a path is constructed. For this, you would like to suspend the construction of the path and suspend Ti*k*Z's parsing of the path, you would then like to have some TeX code executed, and would then like to resume the parsing of the path. This effect can be achieved using the following path operation `\pgfextra`. Note that this operation should only be used by real experts and should only be used deep inside clever macros, not on normal paths.

`\pgfextra{`⟨*code*⟩`}`

This command may only be used inside a Ti*k*Z path. There it is used like a normal path operation. The construction of the path is temporarily suspended and the ⟨*code*⟩ is executed. Then, the path construction is resumed.


```
\newdimen\mydim
\begin{tikzpicture}
  \mydim=1cm
  \draw (0pt,\mydim) \pgfextra{\mydim=2cm} -- (0pt,\mydim);
\end{tikzpicture}
```

`\pgfextra`⟨*code*⟩ `\endpgfextra`

This is an alternative syntax for the `\pgfextra` command. If the code following `\pgfextra` does not start with a brace, the ⟨*code*⟩ is executed until `\endpgfextra` is encountered. What actually happens is that `\pgfextra` that is not followed by a brace completely shuts down the Ti*k*Z parse and `\endpgfextra` is a normal macro that restarts the parser.


```
\newdimen\mydim
\begin{tikzpicture}
  \mydim=1cm
  \draw (0pt,\mydim)
    \pgfextra \mydim=2cm \endpgfextra -- (0pt,\mydim);
\end{tikzpicture}
```

12 Actions on Paths

Once a path has been constructed, different things can be done with it. It can be drawn (or stroked) with a "pen," it can be filled with a color or shading, it can be used for clipping subsequent drawing, it can be used to specify the extend of the picture—or any combination of these actions at the same time.

To decide what is to be done with a path, two methods can be used. First, you can use a special-purpose command like `\draw` to indicate that the path should be drawn. However, commands like `\draw` and `\fill` are just abbreviations for special cases of the more general method: Here, the `\path` command is used to specify the path. Then, options encountered on the path indicate what should be done with the path.

For example, `\path (0,0) circle (1cm);` means "This is a path consisting of a circle around the origin. Do not do anything with it (throw it away)." However, if the option `draw` is encountered anywhere on the path, the circle will be drawn. "Anywhere" is any point on the path where an option can be given, which is everywhere where a path command like `circle (1cm)` or `rectangle (1,1)` or even just `(0,0)` would also be allowed. Thus, the following commands all draw the same circle:

```
\path [draw] (0,0) circle (1cm);
\path (0,0) [draw] circle (1cm);
\path (0,0) circle (1cm) [draw];
```

Finally, `\draw (0,0) circle (1cm);` also draws a path, because `\draw` is an abbreviation for `\path [draw]` and thus the command expands to the first line of the above example.

Similarly, `\fill` is an abbreviation for `\path[fill]` and `\filldraw` is an abbreviation for the command `\path[fill,draw]`. Since options accumulate, the following commands all have the same effect:

```
\path [draw,fill]   (0,0) circle (1cm);
\path [draw] [fill] (0,0) circle (1cm);
\path [fill] (0,0) circle (1cm) [draw];
\draw [fill] (0,0) circle (1cm);
\fill (0,0) [draw] circle (1cm);
\filldraw (0,0) circle (1cm);
```

In the following subsection the different actions are explained that can be performed on a path. The following commands are abbreviations for certain sets of actions, but for many useful combinations there are no abbreviations:

`\draw`

Inside `{tikzpicture}` this is an abbreviation for `\path[draw]`.

`\fill`

Inside `{tikzpicture}` this is an abbreviation for `\path[fill]`.

`\filldraw`

Inside `{tikzpicture}` this is an abbreviation for `\path[fill,draw]`.

`\pattern`

Inside `{tikzpicture}` this is an abbreviation for `\path[pattern]`.

`\shade`

Inside `{tikzpicture}` this is an abbreviation for `\path[shade]`.

`\shadedraw`

Inside `{tikzpicture}` this is an abbreviation for `\path[shade,draw]`.

`\clip`

Inside `{tikzpicture}` this is an abbreviation for `\path[clip]`.

`\useasboundingbox`

Inside `{tikzpicture}` this is an abbreviation for `\path[use as bounding box]`.

12.1 Specifying a Color

The most unspecific option for setting colors is the following:

- `color=`⟨*color name*⟩This option sets the color that is used for fill, drawing, and text inside the current scope. Any special settings for filling colors or drawing colors are immediately "overruled" by this option.

  The ⟨*color name*⟩ is the name of a previously defined color. For LaTeX users, this is just a normal "LaTeX-color" and the `xcolor` extensions are allows. Here is an example:


  ```
  \tikz \fill[color=red!20] (0,0) circle (1ex);
  ```

  It is possible to "leave out" the `color=` part and you can also write:


  ```
  \tikz \fill[red!20] (0,0) circle (1ex);
  ```

  What happens is that every option that TikZ does not know, like `red!20`, gets a "second chance" as a color name.

  For plain TeX users, it is not so easy to specify colors since plain TeX has no "standardized" color naming mechanism. Because of this, PGF emulates the `xcolor` package, though the emulation is *extremely basic* (more precisely, what I could hack together in two hours or so). The emulation allows you to do the following:

  – Specify a new color using `\definecolor`. Only the two color models `gray` and `rgb` are supported.

    Example: `\definecolor{orange}{rgb}{1,0.5,0}`

  – Use `\colorlet` to define a new color based on an old one. Here, the `!` mechanism is supported, though only "once" (use multiple `\colorlet` for more fancy colors).

    Example: `\colorlet{lightgray}{black!25}`

  – Use `\color{`⟨*color name*⟩`}` to set the color in the current TeX group. `\aftergroup`-hackery is used to restore the color after the group.

As pointed out above, the `color=` option applies to "everything" (except to shadings), which is not always what you want. Because of this, there are several more specialized color options. For example, the `draw=` option sets the color used for drawing, but does not modify the color used for filling. These color options are documented where the path action they influence is described.

12.2 Drawing a Path

You can draw a path using the following option:

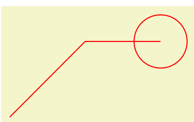
- `draw=`⟨*color*⟩ Causes the path to be drawn. "Drawing" (also known as "stroking") can be thought of as picking up a pen and moving it along the path, thereby leaving "ink" on the canvas.

  There are numerous parameters that influence how a line is drawn, like the thickness or the dash pattern. These options are explained below.

  If the optional ⟨*color*⟩ argument is given, drawing is done using the given ⟨*color*⟩. This color can be different from the current filling color, which allows you to draw and fill a path with different colors. If no ⟨*color*⟩ argument is given, the last usage of the `color=` option is used.

  If the special color name `none` is given, this option causes drawing to be "switched off." This is useful if a style has previously switched on drawing and you locally wish to undo this effect.

  Although this option is normally used on paths to indicate that the path should be drawn, it also makes sense to use the option with a `{scope}` or `{tikzpicture}` environment. However, this will *not* cause all path to drawn. Instead, this just sets the ⟨*color*⟩ to be used for drawing paths inside the environment.

  ```
  \begin{tikzpicture}
    \path[draw=red] (0,0) -- (1,1) -- (2,1) circle (10pt);
  \end{tikzpicture}
  ```

The following subsections list the different options that influence how a path is drawn. All of these options only have an effect if the `draw` options is given (directly or indirectly).

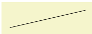
12.2.1 Graphic Parameters: Line Width, Line Cap, and Line Join

- line width=⟨*dimension*⟩ Specifies the line width. Note the space. Default: `0.4pt`.

```
\tikz \draw[line width=5pt] (0,0) -- (1cm,1.5ex);
```

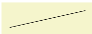
There are a number of predefined styles that provide more "natural" ways of setting the line width. You can also redefine these styles. Remember that you can leave out the `style=` when setting a style.

- `style=ultra thin` Sets the line width to 0.1pt.



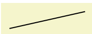
```
\tikz \draw[ultra thin] (0,0) -- (1cm,1.5ex);
```

- `style=very thin` Sets the line width to 0.2pt.



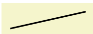
```
\tikz \draw[very thin] (0,0) -- (1cm,1.5ex);
```

- `style=thin` Sets the line width to 0.4pt.



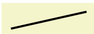
```
\tikz \draw[thin] (0,0) -- (1cm,1.5ex);
```

- `style=semithick` Sets the line width to 0.6pt.



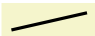
```
\tikz \draw[semithick] (0,0) -- (1cm,1.5ex);
```

- `style=thick` Sets the line width to 0.8pt.



```
\tikz \draw[thick] (0,0) -- (1cm,1.5ex);
```

- `style=very thick` Sets the line width to 1.2pt.

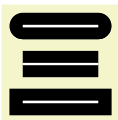


```
\tikz \draw[very thick] (0,0) -- (1cm,1.5ex);
```

- `style=ultra thick` Sets the line width to 1.6pt.

```
\tikz \draw[ultra thick] (0,0) -- (1cm,1.5ex);
```

- cap=⟨*type*⟩ Specifies how lines "end." Permissible ⟨*type*⟩ are `round`, `rect`, and `butt` (default). They have the following effects:

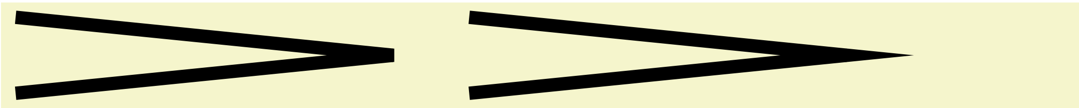


```
\begin{tikzpicture}
  \begin{scope}[line width=10pt]
    \draw[cap=rect]  (0,0 ) -- (1,0);
    \draw[cap=butt]  (0,.5) -- (1,.5);
    \draw[cap=round] (0,1 ) -- (1,1);
  \end{scope}
  \draw[white,line width=1pt]
    (0,0 ) -- (1,0) (0,.5) -- (1,.5) (0,1 ) -- (1,1);
\end{tikzpicture}
```

- join=⟨*type*⟩ Specifies how lines "join." Permissible ⟨*type*⟩ are `round`, `bevel`, and `miter` (default). They have the following effects:

```
\begin{tikzpicture}[line width=10pt]
  \draw[join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
  \useasboundingbox (0,1.5); % make bounding box bigger
\end{tikzpicture}
```

- `miter limit`=⟨*factor*⟩ When you use the miter join and there is a very sharp corner (a small angle), the miter join may protrude very far over the actual joining point. In this case, if it were to protrude by more than ⟨*factor*⟩ times the line width, the miter join is replaced by a bevel join. Default value is `10`.



```
\begin{tikzpicture}[line width=5pt]
  \draw                  (0,0) -- ++(5,.5) -- ++(-5,.5);
  \draw[miter limit=25] (6,0) -- ++(5,.5) -- ++(-5,.5);
  \useasboundingbox (14,0); % make bounding box bigger
\end{tikzpicture}
```

12.2.2 Graphic Parameters: Dash Pattern

- `dash pattern`=⟨*dash pattern*⟩ Sets the dashing pattern. The syntax is the same as in METAFONT. For example `on 2pt off 3pt on 4pt off 4pt` means "draw 2pt, then leave out 3pt, then draw 4pt once more, then leave out 4pt again, repeat".

```
\begin{tikzpicture}[dash pattern=on 2pt off 3pt on 4pt off 4pt]
  \draw (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

- `dash phase`=⟨*dash phase*⟩ Shifts the start of the dash pattern by ⟨*phase*⟩.

```
\begin{tikzpicture}[dash pattern=on 20pt off 10pt]
  \draw[dash phase=0pt] (0pt,3pt) -- (3.5cm,3pt);
  \draw[dash phase=10pt] (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

As for the line thickness, some predefined styles allow you to set the dashing conveniently.

- `style=solid` Shorthand for setting a solid line as "dash pattern." This is the default.

```
\tikz \draw[solid] (0pt,0pt) -- (50pt,0pt);
```

- `style=dotted` Shorthand for setting a dotted dash pattern.

```
\tikz \draw[dotted] (0pt,0pt) -- (50pt,0pt);
```

- `style=densely dotted` Shorthand for setting a densely dotted dash pattern.

```
\tikz \draw[densely dotted] (0pt,0pt) -- (50pt,0pt);
```

- `style=loosely dotted` Shorthand for setting a loosely dotted dash pattern.

```
\tikz \draw[loosely dotted] (0pt,0pt) -- (50pt,0pt);
```

- `style=dashed` Shorthand for setting a dashed dash pattern.

```
\tikz \draw[dashed] (0pt,0pt) -- (50pt,0pt);
```

- `style=densely dashed` Shorthand for setting a densely dashed dash pattern.

```
\tikz \draw[densely dashed] (0pt,0pt) -- (50pt,0pt);
```

- `style=loosely dashed` Shorthand for setting a loosely dashed dash pattern.

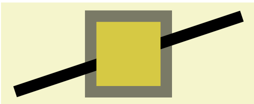
```
\tikz \draw[loosely dashed] (0pt,0pt) -- (50pt,0pt);
```

12.2.3 Graphic Parameters: Draw Opacity

When a line is drawn, it will normally "obscure" everything behind it as if you has used perfectly opaque ink. It is also possible to ask Ti*k*Z to use an ink that is a little bit (or a big bit) transparent. To do so, use the following option:

- `draw opacity`=⟨*value*⟩ This option sets "how transparent" lines should be. A value of `1` means "fully opaque" or "not transparent at all," a value of `0` means "fully transparent" or "invisible." A value of `0.5` yields lines that are semitransparent.

  Note that when you use PostScript as your output format, this option works only with recent versions of GhostScript.

  ```
  \begin{tikzpicture}[line width=1ex]
    \draw (0,0) -- (3,1);
    \filldraw [fill=examplefill,draw opacity=0.5] (1,0) rectangle (2,1);
  \end{tikzpicture}
  ```

Note that the `draw opacity` options only sets the opacity of drawn lines. The opacity of fillings is set using the option `fill opacity` (documented in Section 12.3.3. The option `opacity` sets both at the same time.

- `opacity`=⟨*value*⟩ Sets both the drawing and filling opacity to ⟨*value*⟩.

  The following predefined styles make it easier to use this option:

  – `style=transparent` Makes everything totally transparent and, hence, invisible.

  ```
  \tikz{\fill[red]                   (0,0)   rectangle (1,0.5);
        \fill[transparent,red] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=ultra nearly transparent` Makes everything, well, ultra nearly transparent.

  ```
  \tikz{\fill[red]                           (0,0)   rectangle (1,0.5);
        \fill[ultra nearly transparent] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=very nearly transparent`

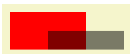
  ```
  \tikz{\fill[red]                          (0,0)   rectangle (1,0.5);
        \fill[very nearly transparent] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=nearly transparent`

  ```
  \tikz{\fill[red]                     (0,0)   rectangle (1,0.5);
        \fill[nearly transparent] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=semitransparent`

  ```
  \tikz{\fill[red]                  (0,0)   rectangle (1,0.5);
        \fill[semitransparent] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=nearly opaque`

  ```
  \tikz{\fill[red]                (0,0)   rectangle (1,0.5);
        \fill[nearly opaque] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=very nearly opaque`

  ```
  \tikz{\fill[red]                     (0,0)   rectangle (1,0.5);
        \fill[very nearly opaque] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=ultra nearly opaque`

  ```
  \tikz{\fill[red]                      (0,0)   rectangle (1,0.5);
        \fill[ultra nearly opaque] (0.5,0) rectangle (1.5,0.25); }
  ```

  – `style=opaque` This yields completely opaque drawings, which is the default.

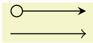
  ```
  \tikz{\fill[red]    (0,0)   rectangle (1,0.5);
        \fill[opaque] (0.5,0) rectangle (1.5,0.25); }
  ```

12.2.4 Graphic Parameters: Arrow Tips

When you draw a line, you can add arrow tips at the ends. It is only possible to add one arrow tip at the start and one at the end. If the path consists of several segments, only the last segment gets arrow tips. The behavior for paths that are closed is not specified and may change in the future.

- `arrows=`⟨*start arrow kind*⟩`-`⟨*end arrow kind*⟩ This option sets the start and end arrow tips (an empty value as in `->` indicates that no arrow tip should be drawn at the start).

  Note: Since the arrow option is so often used, you can leave out the text `arrows=`. What happens is that every option that contains a `-` is interpreted as an arrow specification.

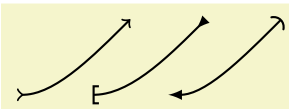


```
\begin{tikzpicture}
  \draw[->]        (0,0)   -- (1,0);
  \draw[o-stealth] (0,0.3) -- (1,0.3);
\end{tikzpicture}
```

  The permissible values are all predefined arrow tips, though you can also define new arrow tip kinds as explained in Section 48. This is often necessary to obtain "double" arrow tips and arrow tips that have a fixed size. Since `pgflibraryarrows` is loaded by default, all arrow tips described in Section 18 are available.

  One arrow tip kind is special: `>` (and all arrow tip kinds containing the arrow tip kind such as `<<` or `>|`). This arrow tip type is not fixed. Rather, you can redefine it using the `>=` option, see below.

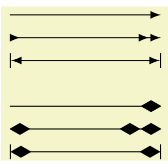
  Example: You can also combine arrow tip types as in



```
\begin{tikzpicture}[thick]
  \draw[to reversed-to]   (0,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \draw[[-latex reversed] (1,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \draw[latex-)]          (2,0) .. controls +(.5,0) and +(-.5,-.5) .. +(1.5,1);
  \useasboundingbox (-.1,-.1) rectangle (3.1,1.1); % make bounding box bigger
\end{tikzpicture}
```

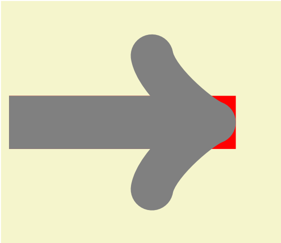
- `>=`⟨*end arrow kind*⟩ This option can be used to redefine the "standard" arrow tip `>`. The idea is that different people have different ideas what arrow tip kind should normally be used. I prefer the arrow tip of TeX's `\to` command (which is used in things like $f\colon A \to B$). Other people will prefer LaTeX's standard arrow tip, which looks like this: . Since the arrow tip kind `>` is certainly the most "natural" one to use, it is kept free of any predefined meaning. Instead, you can change it by saying `>=to` to set the "standard" arrow tip kind to TeX's arrow tip, whereas `>=latex` will set it to LaTeX's arrow tip and `>=stealth` will use a PSTRICKS-like arrow tip.

  Apart from redefining the arrow tip kind `>` (and `<` for the start), this option also redefines the following arrow tip kinds: `>` and `<` as the swapped version of ⟨*end arrow kind*⟩, `<<` and `>>` as doubled versions, `>>` and `<<` as swapped doubled versions, and `|<` and `>|` as arrow tips ending with a vertical bar.



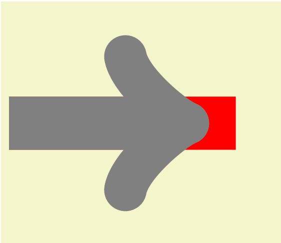
```
\begin{tikzpicture}[scale=2]
  \begin{scope}[>=latex]
    \draw[->]     (0pt,6ex) -- (1cm,6ex);
    \draw[>->>]   (0pt,5ex) -- (1cm,5ex);
    \draw[|<->|]  (0pt,4ex) -- (1cm,4ex);
  \end{scope}
  \begin{scope}[>=diamond]
    \draw[->]     (0pt,2ex) -- (1cm,2ex);
    \draw[>->>]   (0pt,1ex) -- (1cm,1ex);
    \draw[|<->|]  (0pt,0ex) -- (1cm,0ex);
  \end{scope}
\end{tikzpicture}
```

- `shorten >=`⟨*dimension*⟩ This option will shorten the end of lines by the given ⟨*dimension*⟩. If you specify an arrow tip, lines are already shortened a bit such that the arrow tip touches the specified endpoint and does not "protrude over" this point. Here is an example:



```
\begin{tikzpicture}[line width=20pt]
  \useasboundingbox (0,-1.5) rectangle (3.5,1.5);
  \draw[red]        (0,0) -- (3,0);
  \draw[gray,->]    (0,0) -- (3,0);
\end{tikzpicture}
```

The `shorten >` option allows you to shorten the end on the line *additionally* by the given distance. This option can also be useful if you have not specified an arrow tip at all.

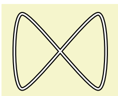


```
\begin{tikzpicture}[line width=20pt]
  \useasboundingbox (0,-1.5) rectangle (3.5,1.5);
  \draw[red]                      (0,0) -- (3,0);
  \draw[-to,shorten >=10pt,gray] (0,0) -- (3,0);
\end{tikzpicture}
```

- `shorten <=`⟨*dimension*⟩ works like `shorten >`, but for the start.

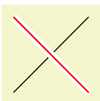
12.2.5 Graphic Parameters: Double Lines and Bordered Lines

- `double=`⟨*core color*⟩ This option causes "two" lines to be drawn instead of a single one. However, this is not what really happens. In reality, the path is drawn twice. First, with the normal drawing color, secondly with the ⟨*core color*⟩, which is normally `white`. Upon the second drawing, the line width is reduced. The net effect is that it appears as if two lines had been drawn and this works well even with complicated, curved paths:



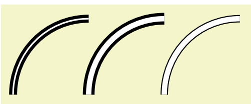
```
\tikz \draw[double]
  plot[smooth cycle] coordinates{(0,0) (1,1) (1,0) (0,1)};
```

You can also use the doubling option to create an effect in which a line seems to have a certain "border":



```
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
  \draw[draw=white,double=red,very thick] (0,1) -- (1,0);
\end{tikzpicture}
```

- `double distance=`⟨*dimension*⟩ Sets the distance the "two" lines are spaced apart (default is 0.6pt). In reality, this is the thickness of the line that is used to draw the path for the second time. The thickness of the *first* time the path is drawn is twice the normal line width plus the given ⟨*dimension*⟩. As a side-effect, this option "selects" the `double` option.



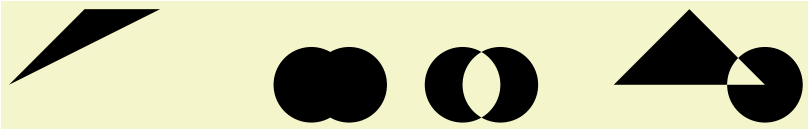
```
\begin{tikzpicture}
  \draw[very thick,double]              (0,0) arc (180:90:1cm);
  \draw[very thick,double distance=2pt] (1,0) arc (180:90:1cm);
  \draw[thin,double distance=2pt]       (2,0) arc (180:90:1cm);
\end{tikzpicture}
```

12.3 Filling a Path

To fill a path, use the following option:

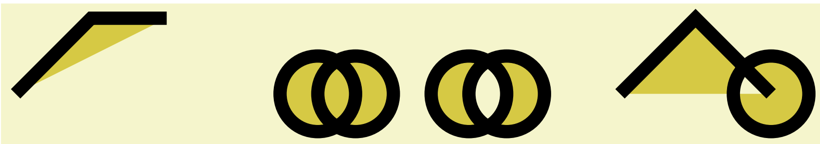
- `fill=`⟨*color*⟩ This option causes the path to be filled. All unclosed parts of the path are first closed, if necessary. Then, the area enclosed by the path is filled with the current filling color, which is either the last color set using the general `color=` option or the optional color ⟨*color*⟩. For self-intersection paths and for paths consisting of several closed areas, the "enclosed area" is somewhat complicated to define and two different definitions exist, namely the nonzero winding number rule and the even odd rule, see the explanation of these options, below.

Just as for the **draw** option, setting ⟨*color*⟩ to **none** disables filling locally.



```
\begin{tikzpicture}
  \fill (0,0) -- (1,1) -- (2,1);
  \fill (4,0) circle (.5cm)  (4.5,0) circle (.5cm);
  \fill[even odd rule] (6,0) circle (.5cm)  (6.5,0) circle (.5cm);
  \fill (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

If the **fill** option is used together with the **draw** option (either because both are given as options or because a **\filldraw** command is used), the path is filled *first*, then the path is drawn *second*. This is especially useful if different colors are selected for drawing and for filling. Even if the same color is used, there is a difference between this command and a plain **fill**: A "filldrawn" area will be slightly larger than a filled area because of the thickness of the "pen."



```
\begin{tikzpicture}[fill=examplefill,line width=5pt]
  \filldraw (0,0) -- (1,1) -- (2,1);
  \filldraw (4,0) circle (.5cm)  (4.5,0) circle (.5cm);
  \filldraw[even odd rule] (6,0) circle (.5cm)  (6.5,0) circle (.5cm);
  \filldraw (8,0) -- (9,1) -- (10,0) circle (.5cm);
\end{tikzpicture}
```

12.3.1 Graphic Parameters: Fill Pattern

Instead of filling a path with a single solid color, it is also possible to fill it with a *tiling pattern*. Imagine a small tile that contains a simple picture like a star. Then these tiles are (conceptually) repeated infinitely in all directions, but clipped against the path.

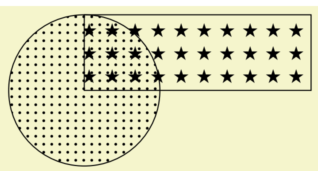
Tiling patterns come in two variants: *inherently colored patterns* and *form-only patterns*. An inherently colored pattern is, say, a red star with a black border and will always look like this. A form-only pattern may have a different color each time it is used, only the form of the pattern will stay the same. As such, form-only patterhns do not have any colors of their own, but when it is used the current *pattern color* is used as its color.

Patterns are not overly flexible. In particular, it is not possible to change the size or orientation of a pattern without declaring a new pattern. For complicated case, it may be easier to use two nested **\foreach** statements to simulate a pattern, but patterns are rendered *much* more quickly than simulated ones.

- **pattern**=⟨*name*⟩ This option causes the path to be filled with a pattern. If the ⟨*name*⟩ is given, this pattern is used, otherwise the pattern set in the enclosing scope is used. As for the **draw** and **fill** options, setting ⟨*name*⟩ to **none** disables filling locally.

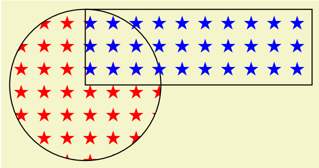
  The pattern works like a fill color. In particular, setting a new fill color will fill the path with a solid color once more.

  Strangely, no ⟨*name*⟩s are permissible by default. You neet to load for instance **pgflibrarypatterns**, see Section 26, to install predefined patterns.

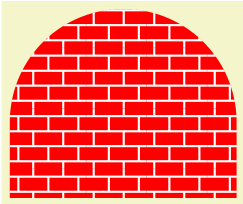


```
\begin{tikzpicture}
  \draw[pattern=dots] (0,0) circle (1cm);
  \draw[pattern=fivepointed stars] (0,0) rectangle (3,1);
\end{tikzpicture}
```

- `pattern color`=⟨*color*⟩ This option is used to set the color to be used for form-only patterns. This option has no effect on inherently colored patterns.



```
\begin{tikzpicture}
  \draw[pattern color=red,pattern=fivepointed stars]  (0,0) circle (1cm);
  \draw[pattern color=blue,pattern=fivepointed stars] (0,0) rectangle (3,1);
\end{tikzpicture}
```

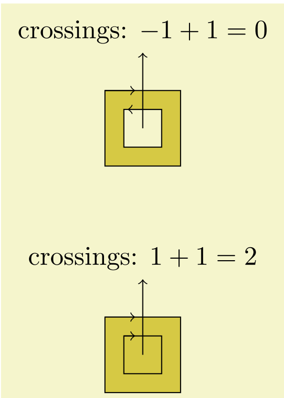


```
\begin{tikzpicture}
  \def\mypath{(0,0) -- +(0,1) arc (180:0:1.5cm) -- +(0,-1)}
  \fill    [red]                                  \mypath;
  \pattern[pattern color=white,pattern=bricks] \mypath;
\end{tikzpicture}
```

12.3.2 Graphic Parameters: Interior Rules

The following two options can be used to decide how interior points should be determined:

- `nonzero rule` If this rule is used (which is the default), the following method is used to determine whether a given point is "inside" the path: From the point, shoot a ray in some direction towards infinity (the direction is chosen such that no strange borderline cases occur). Then the ray may hit the path. Whenever it hits the path, we increase or decrease a counter, which is initially zero. If the ray hits the path as the path goes "from left to right" (relative to the ray), the counter is increased, otherwise it is decreased. Then, at the end, we check whether the counter is nonzero (hence the name). If so, the point is deemed to lie "inside," otherwise it is "outside." Sounds complicated? It is.

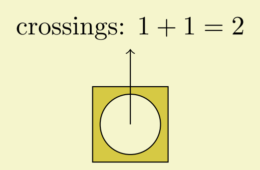



```
\begin{tikzpicture}
  \filldraw[fill=examplefill]
  % Clockwise rectangle
  (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
  % Counter-clockwise rectangle
  (0.25,0.25) -- (0.75,0.25) -- (0.75,0.75) -- (0.25,0.75) -- cycle;

  \draw[->] (0,1) -- (.4,1);
  \draw[->] (0.75,0.75) -- (0.3,.75);

  \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $-1+1 = 0$};

  \begin{scope}[yshift=-3cm]
    \filldraw[fill=examplefill]
    % Clockwise rectangle
    (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
    % Clockwise rectangle
    (0.25,0.25) -- (0.25,0.75) -- (0.75,0.75) -- (0.75,0.25) -- cycle;

    \draw[->] (0,1) -- (.4,1);
    \draw[->] (0.25,0.75) -- (0.4,.75);

    \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $1+1 = 2$};
  \end{scope}
\end{tikzpicture}
```

- `even odd rule` This option causes a different method to be used for determining the inside and outside of paths. While it is less flexible, it turns out to be more intuitive.

  With this method, we also shoot rays from the point for which we wish to determine whether it is inside or outside the filling area. However, this time we only count how often we "hit" the path and declare the point to be "inside" if the number of hits is odd.

  Using the even-odd rule, it is easy to "drill holes" into a path.



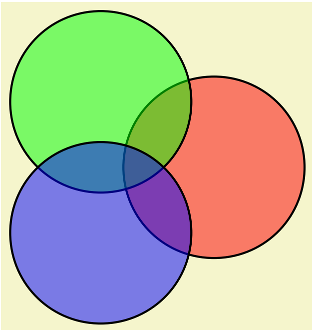

```
\begin{tikzpicture}
  \filldraw[fill=examplefill,even odd rule]
    (0,0) rectangle (1,1) (0.5,0.5) circle (0.4cm);
  \draw[->] (0.5,0.5) -- +(0,1) [above] node{crossings: $1+1 = 2$};
\end{tikzpicture}
```

12.3.3 Graphic Parameters: Fill Opacity

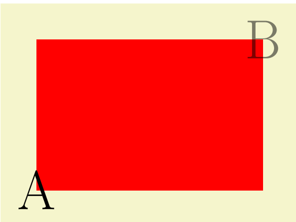
Analogously to the `draw opacity`, you can also set the filling opacity:

- `fill opacity`=⟨*value*⟩ This option sets the opacity of fillings. In addition to filling operations, this opacity also applies to text and images.

  Note, again, that when you use PostScript as your output format, this option works only with recent versions of GhostScript.



```
\begin{tikzpicture}[thick,fill opacity=0.5]
  \filldraw[fill=red]   (0:1cm)    circle (12mm);
  \filldraw[fill=green] (120:1cm)  circle (12mm);
  \filldraw[fill=blue]  (-120:1cm) circle (12mm);
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \fill[red] (0,0) rectangle (3,2);

  \node                    at (0,0) {\huge A};
  \node[fill opacity=0.5] at (3,2) {\huge B};
\end{tikzpicture}
```

12.4 Shading a Path

You can shade a path using the `shade` option. A shading is like a filling, only the shading changes its color smoothly from one color to another.

- `shade` Causes the path to be shaded using the currently selected shading (more on this later). If this option is used together with the `draw` option, then the path is first shaded, then drawn.

  It is not an error to use this option together with the `fill` option, but it makes no sense.


```
\tikz \shade (0,0) circle (1ex);
```

```
\tikz \shadedraw (0,0) circle (1ex);
```

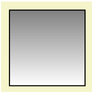
For some shadings it is not really clear how they can "fill" the path. For example, the `ball` shading normally looks like this: . How is this supposed to shade a rectangle? Or a triangle?

To solve this problem, the predefined shadings like `ball` or `axis` fill a large rectangle completely in a sensible way. Then, when the shading is used to "shade" a path, what actually happens is that the path is temporarily used for clipping and then the rectangular shading is drawn, scaled and shifted such that all parts of the path are filled.

12.4.1 Choosing a Shading Type

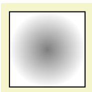
The default shading is a smooth transition from gray to white and from above to bottom. However, other shadings are also possible, for example a shading that will sweep a color from the center to the corners outward. To choose the shading, you can use the `shading=` option, which will also automatically invoke the `shade` option. Note that this does *not* change the shading color, only the way the colors sweep. For changing the colors, other options are needed, which are explained below.

- `shading=`⟨*name*⟩ This selects a shading named ⟨*name*⟩. The following shadings are predefined:
  - `axis` This is the default shading in which the color changes gradually between three horizontal lines. The top line is at the top (uppermost) point of the path, the middle is in the middle, the bottom line is at the bottom of the path.

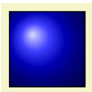
    ```
    \tikz \shadedraw [shading=axis] (0,0) rectangle (1,1);
    ```

    The default top color is gray, the default bottom color is white, the default middle is the "middle" of these two.
  - `radial` This shading fills the path with a gradual sweep from a certain color in the middle to another color at the border. If the path is a circle, the outer color will be reached exactly at the border. If the shading is not a circle, the outer color will continue a bit towards the corners. The default inner color is gray, the default outer color is white.

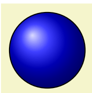
    

    ```
    \tikz \shadedraw [shading=radial] (0,0) rectangle (1,1);
    ```

  - `ball` This shading fills the path with a shading that "looks like a ball." The default "color" of the ball is blue (for no particular reason).

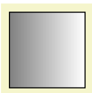
    

    ```
    \tikz \shadedraw [shading=ball] (0,0) rectangle (1,1);
    ```

    ```
    \tikz \shadedraw [shading=ball] (0,0) circle (.5cm);
    ```

- `shading angle=`⟨*degrees*⟩ This option rotates the shading (not the path!) by the given angle. For example, we can turn a top-to-bottom axis shading into a left-to-right shading by rotating it by 90°.

  ```
  \tikz \shadedraw [shading=axis,shading angle=90] (0,0) rectangle (1,1);
  ```

You can also define new shading types yourself. However, for this, you need to use the basic layer directly, which is, well, more basic and harder to use. Details on how to create a shading appropriate for filling paths are given in Section 55.3.

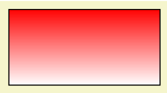
12.4.2 Choosing a Shading Color

The following options can be used to change the colors used for shadings. When one of these options is given, the `shade` option is automatically selected and also the "right" shading.

- `top color=`⟨*color*⟩ This option prescribes the color to be used at the top in an `axis` shading. When this option is given, several things happen:

  1. The `shade` option is selected.

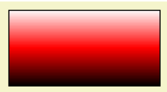
2. The `shading=axis` option is selected.
3. The middle color of the axis shading is set to the average of the given top color ⟨*color*⟩ and of whatever color is currently selected for the bottom.
4. The rotation angle of the shading is set to 0.



```
\tikz \draw[top color=red] (0,0) rectangle (2,1);
```

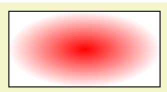
- `bottom color`=⟨*color*⟩ This option works like `top color`, only for the bottom color.
- `middle color`=⟨*color*⟩ This option specifies the color for the middle of an axis shading. It also sets the `shade` and `shading=axis` options, but it does not change the rotation angle.

  Note: Since both `top color` and `bottom color` change the middle color, this option should be given *last* if all of these options need to be given:



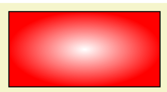
```
\tikz \draw[top color=white,bottom color=black,middle color=red]
  (0,0) rectangle (2,1);
```

- `left color`=⟨*color*⟩ This option does exactly the same as `top color`, except that the shading angle is set to 90°.
- `right color`=⟨*color*⟩ Works like `left color`.
- `inner color`=⟨*color*⟩ This option sets the color used at the center of a `radial` shading. When this option is used, the `shade` and `shading=radial` options are set.



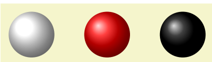
```
\tikz \draw[inner color=red] (0,0) rectangle (2,1);
```

- `outer color`=⟨*color*⟩ This option sets the color used at the border and outside of a `radial` shading.



```
\tikz \draw[outer color=red,inner color=white]
  (0,0) rectangle (2,1);
```

- `ball color`=⟨*color*⟩ This option sets the color used for the ball shading. It sets the `shade` and `shading=ball` options. Note that the ball will never "completely" have the color ⟨*color*⟩. At its "highlight" spot a certain amount of white is mixed in, at the border a certain amount of black. Because of this, it also makes sense to say `ball color=white` or `ball color=black`



```
\begin{tikzpicture}
  \shade[ball color=white] (0,0) circle (2ex);
  \shade[ball color=red] (1,0) circle (2ex);
  \shade[ball color=black] (2,0) circle (2ex);
\end{tikzpicture}
```

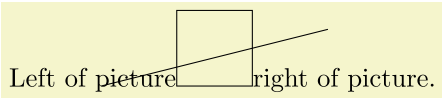
12.5 Establishing a Bounding Box

PGF is reasonably good at keeping track of the size of your picture and reserving just the right amount of space for it in the main document. However, in some cases you may want to say things like "do not count this for the picture size" or "the picture is actually a little large." For this you can use the option `use as bounding box` or the command `\useasboundingbox`, which is just a shorthand for `\path[use as bounding box]`.

- **use as bounding box** Normally, when this option is given on a path, the bounding box of the present path is used to determine the size of the picture and the size of all *subsequent* paths are ignored. However, if there were previous path operations that have already established a larger bounding box, it will not be made smaller by this operation.

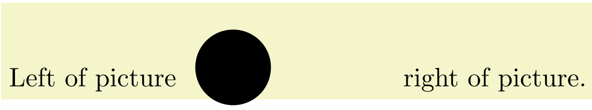
  In a sense, `use as bounding box` has the same effect as clipping all subsequent drawing against the current path—without actually doing the clipping, only making PGF treat everything as if it were clipped.

  The first application of this option is to have a `{tikzpicture}` overlap with the main text:




```
Left of picture\begin{tikzpicture}
  \draw[use as bounding box] (2,0) rectangle (3,1);
  \draw (1,0) -- (4,.75);
\end{tikzpicture}right of picture.
```

  In a second application this option can be used to get better control over the white space around the picture:

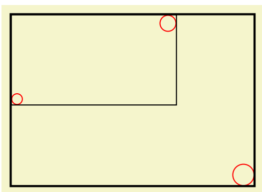



```
Left of picture
\begin{tikzpicture}
  \useasboundingbox (0,0) rectangle (3,1);
  \fill (.75,.25) circle (.5cm);
\end{tikzpicture}
right of picture.
```

  Note: If this option is used on a path inside a TEX group (scope), the effect "lasts" only till the end of the scope. Again, this behavior is the same as for clipping.

There is a node that allows you to get the size of the current bounding box. The `current bounding box` node has the `rectangle` shape and its size is always the size of the current bounding box.

Similarly, the `current path bounding box` node has the `rectangle` hape and the size of the bounding box of the current path.



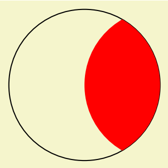
```
\begin{tikzpicture}
  \draw[red] (0,0) circle (2pt);
  \draw[red] (2,1) circle (3pt);

  \draw (current bounding box.south west) rectangle
        (current bounding box.north east);

  \draw[red] (3,-1) circle (4pt);

  \draw[thick] (current bounding box.south west) rectangle
               (current bounding box.north east);
\end{tikzpicture}
```

12.6 Using a Path For Clipping

To use a path for clipping, use the `clip` option.

- **clip** This option causes all subsequent drawings to be clipped against the current path and the size of subsequent paths will not be important for the picture size. If you clip against a self-intersecting path, the even-odd rule or the nonzero winding number rule is used to determine whether a point is inside or outside the clipping region.

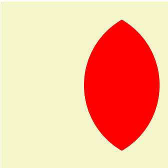
  The clipping path is a graphic state parameter, so it will be reset at the end of the current scope. Multiple clippings accumulate, that is, clipping is always done against the intersection of all clipping areas that have been specified inside the current scopes. The only way of enlarging the clipping area is to end a {scope}.



```
\begin{tikzpicture}
  \draw[clip] (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

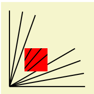
It is usually a *very* good idea to apply the clip option only to the first path command in a scope.

If you "only wish to clip" and do not wish to draw anything, you can use the \clip command, which is a shorthand for \path[clip].



```
\begin{tikzpicture}
  \clip (0,0) circle (1cm);
  \fill[red] (1,0) circle (1cm);
\end{tikzpicture}
```

To keep clipping local, use {scope} environments as in the following example:



```
\begin{tikzpicture}
  \draw (0,0) -- ( 0:1cm);
  \draw (0,0) -- (10:1cm);
  \draw (0,0) -- (20:1cm);
  \draw (0,0) -- (30:1cm);
  \begin{scope}[fill=red]
    \fill[clip] (0.2,0.2) rectangle (0.5,0.5);

    \draw (0,0) -- (40:1cm);
    \draw (0,0) -- (50:1cm);
    \draw (0,0) -- (60:1cm);
  \end{scope}
  \draw (0,0) -- (70:1cm);
  \draw (0,0) -- (80:1cm);
  \draw (0,0) -- (90:1cm);
\end{tikzpicture}
```

There is a slightly annoying catch: You cannot specify certain graphic options for the command used for clipping. For example, in the above code we could not have moved the fill=red to the \fill command. The reasons for this have to do with the internals of the PDF specification. You do not want to know the details. It is best simply not to specify any options for these commands.

13 Nodes and Edges

13.1 Overview

In the present section, the usage of *nodes* in Ti*k*Z is explained. A node is typically a rectangle or circle or another simple shape with some text on it.

Nodes are added to paths using the special path operation `node`. Nodes *are not part of the path itself.* Rather, they are added to the picture after the path has been drawn.

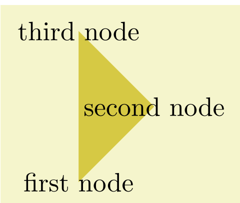
In Section 13.2 the basic syntax of the node operation is explained, followed in Section 13.3 by the syntax for multi-part nodes, which are nodes that contain several different text parts. After this, the different options for the text in nodes are explained. In Section 13.5 the concept of *anchors* is introduced along with their usage. In Section 13.6 the different ways transformations affect nodes are studied. Sections 13.7 and 13.8 are about placing nodes on or next to straight lines and curves. In Section 13.10 it is explained how a node can be used as a "pseudo-coordinate." Section 13.11 introduces the `edge` operation, which works similar to the `to` operation and also similar to the `node` operation. Section 13.13 lists the predefined shapes. Finally, Section 13.14 explains the special `after node path` options.

13.2 Nodes and Their Shapes

In the simplest case, a node is just some text that is placed at some coordinate. However, a node can also have a border drawn around it or have a more complex background and foreground. Indeed, some nodes do not have a text at all, but consist solely of the background. You can name nodes so that you can reference their coordinates later in the same picture or, if certain precautions are taken as explained in Section 13.12, also in different pictures.

There are no special TEX commands for adding a node to a picture; rather, there is path operation called `node` for this. Nodes are created whenever Ti*k*Z encounters `node` or `coordinate` at a point on a path where it would expect a normal path operation (like `-- (1,1)` or `sin (1,1)`). It is also possible to give node specifications *inside* certain path operations as explained later.

The node operation is typically followed by some options, which apply only to the node. Then, you can optionally *name* the node by providing a name in round braces. Lastly, for the `node` operation you must provide some label text for the node in curly braces, while for the `coordinate` operation you may not. The node is placed at the current position of the path *after the path has been drawn.* Thus, all nodes are drawn "on top" of the path and retained until the path is complete. If there are several nodes on a path, they are drawn on top of the path in the order they are encountered.




```
\tikz \fill[fill=examplefill]
     (0,0) node {first node}
  -- (1,1) node {second node}
  -- (0,2) node {third node};
```

The syntax for specifying nodes is the following:

`\path ... node[⟨options⟩](⟨name⟩)at(⟨coordinate⟩){⟨text⟩} ...;`

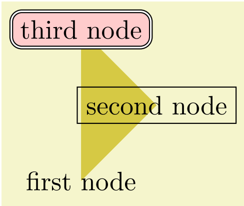
The effect of `at` is to place the node at the coordinate given after `at` and not, as would normally be the case, at the last position. The `at` syntax is not available when a node is given inside a path operation (it would not make any sense, there).

The `(⟨name⟩)` is a name for later reference and it is optional. You may also add the option `name=`⟨*name*⟩ to the ⟨*option*⟩ list; it has the same effect.

- `name=`⟨*node name*⟩ assigns a name to the node for later reference. Since this is a "high-level" name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a node. Thus, you can name a node just `1` or perhaps `start of chart` or even `y_1`. Your node name should *not* contain any punctuation like a dot, a comma, or a colon since these are used to detect what kind of coordinate you mean when you reference a node.
- `at=`⟨*coordinate*⟩ is another way of specifying ath `at` coordinate.

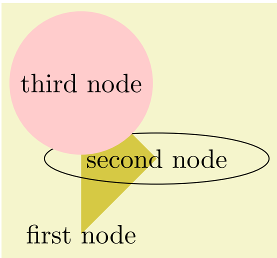
The ⟨*options*⟩ is an optional list of options that *apply only to the node* and have no effect outside. The other way round, most "outside" options also apply to the node, but not all. For example, the "outside" rotation does not apply to nodes (unless some special options are used, sigh). Also, the outside path action, like `draw` or `fill`, never applies to the node and must be given in the node (unless some special other options are used, deep sigh).

As mentioned before, we can add a border and even a background to a node:




```
\tikz \fill[fill=examplefill]
      (0,0) node {first node}
   -- (1,1) node[draw] {second node}
   -- (0,2) node[fill=red!20,draw,double,rounded corners] {third node};
```

The "border" is actually just a special case of a much more general mechanism. Each node has a certain *shape* which, by default, is a rectangle. However, we can also ask TikZ to use a circle shape instead or an ellipse shape (you have to include `pgflibraryshapes` for the latter shape):




```
\tikz \fill[fill=examplefill]
      (0,0) node{first node}
   -- (1,1) node[ellipse,draw] {second node}
   -- (0,2) node[circle,fill=red!20] {third node};
```

In the future, there might be much more complicated shapes available such as, say, a shape for a resistor or a shape for a UML class. Unfortunately, creating new shapes is a bit tricky and makes it necessary to use the basic layer directly. Life is hard.

To select the shape of a node, the following option is used:

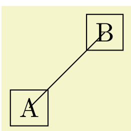
- `shape`=⟨*shape name*⟩ select the shape either of the current node or, when this option is not given inside a node but somewhere outside, the shape of all nodes in the current scope.

  Since this option is used often, you can leave out the `shape=`. When TikZ encounters an option like `circle` that it does not know, it will, after everything else has failed, check whether this option is the name of some shape. If so, that shape is selected as if you had said `shape=`⟨*shape name*⟩.

  By default, the following shapes are available: `rectangle`, `circle`, `coordinate`, and, when the package `pgflibraryshapes` is loaded, also `ellipse`. Details of these shapes, like their anchors and size options, are discussed in Section 13.13.

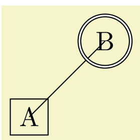
The following styles influences how nodes are rendered:

- `style=every node` This style is installed at the beginning of every node.




```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \draw (0,0) node {A} -- (1,1) node {B};
\end{tikzpicture}
```

- `style=every` ⟨*shape*⟩ `node` These styles are installed at the beginning of a node of a given ⟨*shape*⟩. For example, `every rectangle node` is used for rectangle nodes, and so on.




```
\begin{tikzpicture}
  \tikzstyle{every rectangle node}=[draw]
  \tikzstyle{every circle node}=   [draw,double]
  \draw (0,0) node[rectangle] {A} -- (1,1) node[circle] {B};
\end{tikzpicture}
```

There is a special syntax for specifying "light-weighed" nodes:

`\path ... coordinate[`⟨*options*⟩`](`⟨*name*⟩`)at(`⟨*coordinate*⟩`) ...;`

This has the same effect as

`node[shape=coordinate][`⟨*options*⟩`](`⟨*name*⟩`)at(`⟨*coordinate*⟩`){}`,

where the `at` part might be missing.

Since nodes are often the only path operation on paths, there are two special commands for creating paths containing only a node:

`\node`

Inside `{tikzpicture}` this is an abbreviation for `\path node`.

`\coordinate`

Inside `{tikzpicture}` this is an abbreviation for `\path coordinate`.

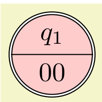
13.3 Multi-Part Nodes

Most nodes just have a single simple text label. However, nodes of a more complicated shapes might be made up from several *node parts*. For example, in automata theory a so-called Moore state has a state name, drawn in the upper part of the state circle, and an output text, drawn in the lower part of the state circle. These two parts are quite independent. Similarly, a UML class shape would have a name part, a method part, and an attributes part. Different molecule shape might use parts for the different atoms to be drawn at the different positions, and so on.

Both PGF and Ti*k*Z support such multipart nodes. On the lower level, PGF provides a system for specifying that a shape consists of several parts. On the Ti*k*Z level, you specify the different node parts by using the following command:

`\nodepart{`⟨*part name*⟩`}`

This command can only be used inside the ⟨*text*⟩ argument of a `node` path operation. It works a little bit like a `\part` command in LaTeX. It will stop the typesetting of whatever node part was typeset until now and then start putting all following text into the node part named ⟨*part name*⟩—until another `\partname` is encountered or until the node ⟨*text*⟩ ends.

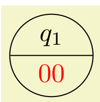



```
\begin{tikzpicture}
  \node [circle split,draw,double,fill=red!20]
  {
    % No \nodepart has been used, yet. So, the following is put in the
    % ``text'' node part by default.
    $q_1$
    \nodepart{lower} % Ok, end ``text'' part, start ``output'' part
    $00$
  }; % output part ended.
\end{tikzpicture}
```

You will have to lookup which parts are defined by a shape.

The following styles influences node parts:

- `style=every ⟨part name⟩ node part` This style is installed at the beginning of every node part named ⟨*part name*⟩.

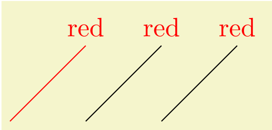



```
\tikzstyle{every lower node part}=[red]
\tikz \node [circle split,draw] {$q_1$ \nodepart{lower} $00$};
```

13.4 Options for the Text in Nodes

The simplest option for the text in nodes is its color. Normally, this color is just the last color installed using `color=`, possibly inherited from another scope. However, it is possible to specificly set the color used for text using the following option:

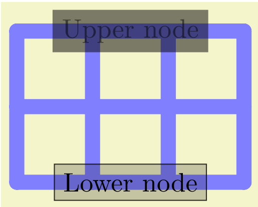
- `text=`⟨*color*⟩ Sets the color to be used for text labels. A `color=` option will immediately override this option.




```
\begin{tikzpicture}
  \draw[red]        (0,0) -- +(1,1) node[above]     {red};
  \draw[text=red]   (1,0) -- +(1,1) node[above]     {red};
  \draw             (2,0) -- +(1,1) node[above,red] {red};
\end{tikzpicture}
```

Just like the color itself, you may also wish to set the opacity of the text only. For this, use the following option:

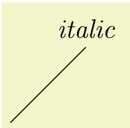
- text opacity=⟨*value*⟩ Sets the opacity of text labels.




```
\begin{tikzpicture}
  \draw[line width=2mm,blue!50,cap=round] (0,0) grid (3,2);
  \tikzstyle{every node}=[fill,draw]

  \node[opacity=0.5] at (1.5,2) {Upper node};
  \node[draw opacity=0.8,fill opacity=0.2,text opacity=1]
    at (1.5,0) {Lower node};
\end{tikzpicture}
```

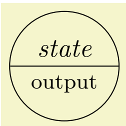
Next, you may wish to adjust the font used for the text. Use the following option for this:

- font=⟨*font commands*⟩ Sets the font used for text labels.




```
\begin{tikzpicture}
  \draw[font=\itshape] (1,0) -- +(1,1) node[above] {italic};
\end{tikzpicture}
```

A perhaps more useful example is the following:




```
\tikzstyle{every text node part}=[font=\itshape]
\tikzstyle{every lower node part}=[font=\footnotesize]
\tikz \node [circle split,draw] {state \nodepart{lower} output};
```

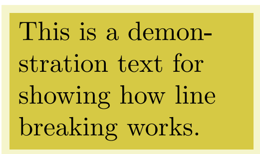
Normally, when a node is typeset, all the text you give in the braces is but in one long line (in an `\hbox`, to be precise) and the node will become as wide as necessary.

You can change this behaviour using the following options. They allow you to limit the width of a node (naturally, at the expense of its height).

- text width=⟨*dimension*⟩ This option will put the text of a node in a box of the given width (more precisely, in a `{minipage}` of this width; for plain TeX a rudimentary "minipage emulation" is used).

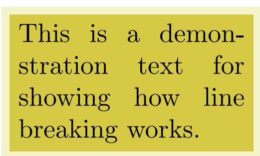
  If the node text is not as wide as ⟨*dimension*⟩, it will nevertheless be put in a box of this width. If it is larger, line breaking will be done.

  By default, when this option is given, a ragged right border will be used. This is sensible since, typically, these boxes are narrow and justifying the text looks ugly.




```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm]
  {This is a demonstration text for showing how line breaking works.};
```

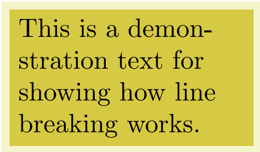
- text justified causes the text to be justified instead of (right)ragged. Use this only with pretty broad nodes.




```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text justified]
  {This is a demonstration text for showing how line breaking works.};
```

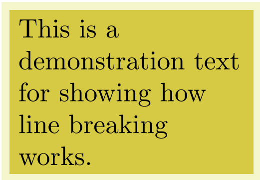
In the above example, TeX complains (rightfully) about three very badly typeset lines. (For this manual I asked TeX to stop complaining by using `\hbadness=10000`, but this is a foul deed, indeed.)

- `text ragged` causes the text to be typeset with a ragged right. This uses the original plain TeX definition of a ragged right border, in which TeX will try to balance the right border as well as possible. This is the default.



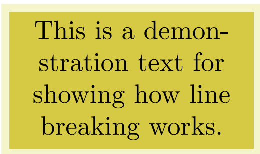

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text ragged]
  {This is a demonstration text for showing how line breaking works.};
```

- `text badly ragged` causes the right border to be ragged in the LaTeX-style, in which no balancing occurs. This looks ugly, but it may be useful for very narrow boxes and when you wish to avoid hyphenations.



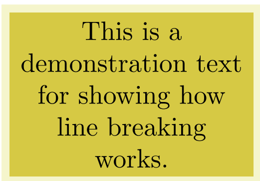

```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text badly ragged]
  {This is a demonstration text for showing how line breaking works.};
```

- `text centered` centers the text, but tries to balance the lines.




```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text centered]
  {This is a demonstration text for showing how line breaking works.};
```

- `text badly centered` centers the text, without balancing the lines.

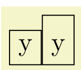



```
\tikz \draw (0,0) node[fill=examplefill,text width=3cm,text badly centered]
  {This is a demonstration text for showing how line breaking works.};
```

In addition to changing the width of nodes, you can also change the height of nodes. This can be done in two ways: First, you can use the option `minimum height`, which ensures that the height of the whole node is at least the given height (this option is described in more detail later). Second, you can use the option `text height`, which sets the height of the text itself, more precisely, of the TeX text box of the text. Note that the `text height` typically is not the height of the shape's box: In addition to the `text height`, an internal `inner sep` is added as extra space and the text depth is also taken into account.

I recommend using `minimum size` instead of `text height` except for special situations.

- `text height`=⟨*dimension*⟩ Sets the height of the text boxes in shapes. Thus, when you write something like `node {text}`, the `text` is first typeset, resulting in some box of a certain height. This height is then replaced by the height `text height`. The resulting box is then used to determine the size of the shape, which will typically be larger. When you write `text height=` without specifying anything, the "natural" size of the text box remains unchanged.



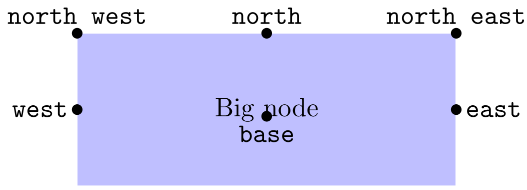

```
\tikz \node[draw]                 {y};
\tikz \node[draw,text height=10pt] {y};
```

- `text depth`=⟨*dimension*⟩ This option works like `text height`, only for the depth of the text box. This option is mostly useful when you need to ensure a uniform depth of text boxes that need to be aligned.

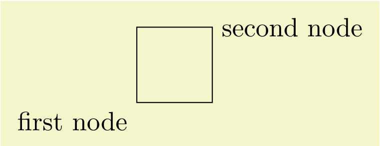
13.5 Placing Nodes Using Anchors

When you place a node at some coordinate, the node is centered on this coordinate by default. This is often undesirable and it would be better to have the node to the right or above the actual coordinate.

PGF uses a so-called anchoring mechanism to give you a very fine control over the placement. The idea is simple: Imaging a node of rectangular shape of a certain size. PGF defines numerous anchor positions in the shape. For example to upper right corner is called, well, not "upper right anchor," but the `north east` anchor of the shape. The center of the shape has an anchor called `center` on top of it, and so on. Here are some examples (a complete list is given in Section 13.13).



Now, when you place a node at a certain coordinate, you can ask Ti*k*Z to place the node shifted around in such a way that a certain anchor is at the coordinate. In the following example, we ask Ti*k*Z to shift the first node such that its `north east` anchor is at coordinate `(0,0)` and that the `west` anchor of the second node is at coordinate `(1,1)`.



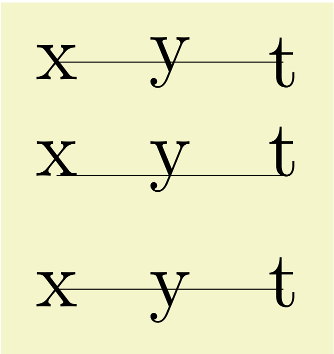

```
\tikz \draw           (0,0) node[anchor=north east] {first node}
          rectangle (1,1) node[anchor=west] {second node};
```

Since the default anchor is `center`, the default behaviour is to shift the node in such a way that it is centered on the current position.

- `anchor`=⟨*anchor name*⟩ causes the node to be shifted such that it's anchor ⟨*anchor name*⟩ lies on the current coordinate.

  The only anchor that is present in all shapes is `center`. However, most shapes will at least define anchors in all "compass directions." Furthermore, the standard shapes also define a `base` anchor, as well as `base west` and `base east`, for placing things on the baseline of the text.

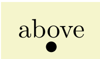
  The standard shapes also define a `mid` anchor (and `mid west` and `mid east`). This anchor is half the height of the character "x" above the base line. This anchor is useful for vertically centering multiple nodes that have different heights and depth. Here is an example:



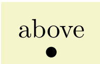

```
\begin{tikzpicture}[scale=3,transform shape]
  % First, center alignment -> wobbles
  \draw[anchor=center] (0,1)  node{x} -- (0.5,1)  node{y} -- (1,1)  node{t};
  % Second, base alignment -> no wobble, but too high
  \draw[anchor=base]   (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t};
  % Third, mid alignment
  \draw[anchor=mid]    (0,0)  node{x} -- (0.5,0)  node{y} -- (1,0)  node{t};
\end{tikzpicture}
```

Unfortunately, while perfectly logical, it is often rather counter-intuitive that in order to place a node *above* a given point, you need to specify the `south` anchor. For this reason, there are some useful options that allow you to select the standard anchors more intuitively:

- `above=`⟨*offset*⟩ does the same as `anchor=south`. If the ⟨*offset*⟩ is specified, the node is additionally shifted upwards by the given ⟨*offset*⟩.

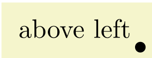



```
\tikz \fill (0,0) circle (2pt) node[above] {above};
```

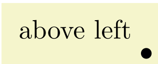



```
\tikz \fill (0,0) circle (2pt) node[above=2pt] {above};
```

- `above left=`⟨*offset*⟩ does the same as `anchor=south east`. If the ⟨*offset*⟩ is specified, the node is additionally shifted upwards and right by ⟨*offset*⟩.

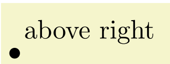



```
\tikz \fill (0,0) circle (2pt) node[above left] {above left};
```




```
\tikz \fill (0,0) circle (2pt) node[above left=2pt] {above left};
```

- `above right=`⟨*offset*⟩ does the same as `anchor=south west`.




```
\tikz \fill (0,0) circle (2pt) node[above right] {above right};
```

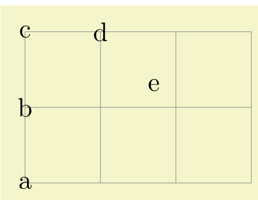
- `left=`⟨*offset*⟩ does the same as `anchor=east`.


```
\tikz \fill (0,0) circle (2pt) node[left] {left};
```

- `right=`⟨*offset*⟩ does the same as `anchor=west`.
- `below=`⟨*offset*⟩ does the same as `anchor=north`.
- `below left=`⟨*offset*⟩ does the same as `anchor=north east`.
- `below right=`⟨*offset*⟩ does the same as `anchor=north west`.

A second set of options behaves similarly, namely the `above of`, `below of`, and so on options. They cause the same anchors to be set as the options without `of`, however, their parameter is different: You must provide the name of another node. The current node will then be placed, say, above this specified node at a distance given by the option `node distance`.

- `above of=`⟨*node*⟩ This option causes the node to be placed at the distance `node distance` above of ⟨*node*⟩. The anchor is `center`.




```
\begin{tikzpicture}[node distance=1cm]
  \draw[help lines] (0,0) grid (3,2);
  \node (a)                    {a};
  \node (b) [above of=a]       {b};
  \node (c) [above of=b]       {c};
  \node (d) [right of=c]       {d};
  \node (e) [below right of=d] {e};
\end{tikzpicture}
```

- `above left of=`⟨*node*⟩ Works like `above of`, only the node is now put above and left. The `node distance` is the Euclidean distance between the two nodes, not the L_1-distance.
- `above right of=`⟨*node*⟩ works similarly.

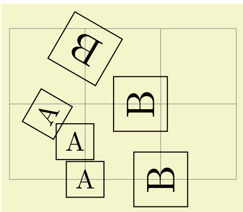
- left of=⟨*node*⟩ works similarly.
- right of=⟨*node*⟩ works similarly.
- below of=⟨*node*⟩ works similarly.
- below left of=⟨*node*⟩ works similarly.
- below right of=⟨*node*⟩ works similarly.
- node distance=⟨*dimension*⟩ sets the distance between nodes that are placed using the ... of options. Note that this distance is the distance between the centers of the nodes, not the distance between their borders.

13.6 Transformations

It is possible to transform nodes, but, by default, transformations do not apply to nodes. The reason is that you usually do *not* want your text to be scaled or rotated even if the main graphic is transformed. Scaling text is evil, rotating slightly less so.

However, sometimes you *do* wish to transform a node, for example, it certainly sometimes makes sense to rotate a node by 90 degrees. There are two ways in which you can achieve this:

1. You can use the following option:
   - transform shape causes the current "external" transformation matrix to be applied to the shape. For example, if you said \tikz[scale=3] and then say node[transform shape] {X}, you will get a "huge" X in your graphic.
2. You can give transformation option *inside* the option list of the node. *These* transformations always apply to the node.




```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \draw[style=help lines] (0,0) grid (3,2);
  \draw              (1,0) node{A}
                     (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=30]  (1,0) node{A}
                     (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=60]  (1,0) node[transform shape] {A}
                     (2,0) node[transform shape,rotate=90,scale=1.5] {B};
\end{tikzpicture}
```

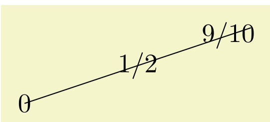
13.7 Placing Nodes on a Line or Curve Explicitly

Until now, we always placed node on a coordinate that is mentioned in the path. Often, however, we wish to place nodes on "the middle" of a line and we do not wish to compute these coordinates "by hand." To facilitate such placements, TikZ allows you to specify that a certain node should be somewhere "on" a line. There are two ways of specifying this: Either explicitly by using the pos option or implicitly by placing the node "inside" a path operation. These two ways are described in the following.

- pos=⟨*fraction*⟩ When this option is given, the node is not anchored on the last coordinate. Rather, it is anchored on some point on the line from the previous coordinate to the current point. The ⟨*fraction*⟩ dictates how "far" on the line the point should be. A ⟨*fraction*⟩ or 0 is the previous coordinate, 1 is the current one, everything else is in between. In particular, 0.5 is the middle.

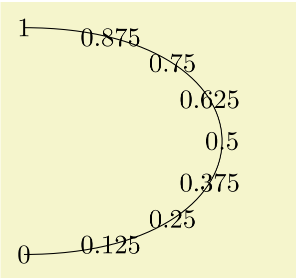
  Now, what is "the previous line"? This depends on the previous path construction operation.

  In the simplest case, the previous path operation was a "line-to" operation, that is, a --⟨*coordinate*⟩ operation:



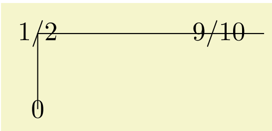

```
\tikz \draw (0,0) -- (3,1)
    node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

The next case is the curve-to operation (the `..` operation). In this case, the "middle" of the curve, that is, the position `0.5` is not necessarily the point at the exact half distance on the line. Rather, it is some point at "time" 0.5 of a point traveling from the start of the curve, where it is at time 0, to the end of the curve, which it reaches at time 0.5. The "speed" of the point depends on the length of the support vectors (the vectors that connect the start and end points to the control points). The exact math is a bit complicated (depending on your point of view, of course); you may wish to consult a good book on computer graphics and Bézier curves if you are intrigued.

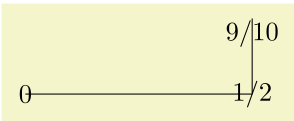



```
\tikz \draw (0,0) .. controls +(right:3.5cm) and +(right:3.5cm) .. (0,3)
  \foreach \p in {0,0.125,...,1} {node[pos=\p]{\p}};
```

Another interesting case are the horizontal/vertical line-to operations `|-` and `-|`. For them, the position (or time) `0.5` is exactly the corner point.




```
\tikz \draw (0,0) |- (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

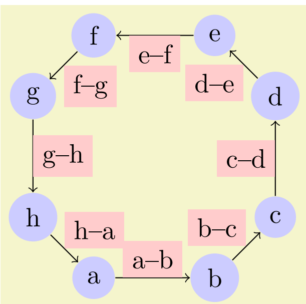



```
\tikz \draw (0,0) -| (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

For all other path construction operations, *the position placement does not work*, currently. This will hopefully change in the future (especially for the arc operation).

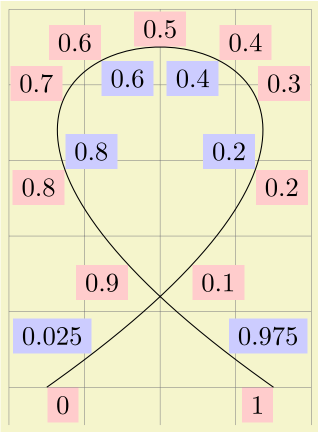
- `auto=`⟨*direction*⟩ This option causes an anchor positions to be calculated automatically according to the following rule. Consider a line between to points. If the ⟨*direction*⟩ is `left`, then the anchor is chosen such that the node is to the left of this line. If the ⟨*direction*⟩ is `right`, then the node is to the right of this line. Leaving out ⟨*direction*⟩ causes automatic placement to be enabled with the last value of `left` or `right` used. A ⟨*direction*⟩ of `false` disables automatic placement. This happens also whenever an anchor is given explicitly by the `anchor` option or by one of the `above`, `below`, etc. options.

  This option only has an effect for nodes that are placed on lines or curves.

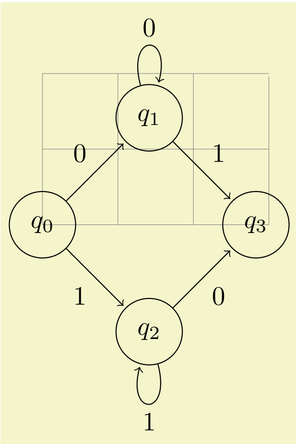



```
\begin{tikzpicture}[scale=.8,auto=left]
  \tikzstyle{every node}=[circle,fill=blue!20]
  \node (a) at (-1,-2) {a};
  \node (b) at ( 1,-2) {b};
  \node (c) at ( 2,-1) {c};
  \node (d) at ( 2, 1) {d};
  \node (e) at ( 1, 2) {e};
  \node (f) at (-1, 2) {f};
  \node (g) at (-2, 1) {g};
  \node (h) at (-2,-1) {h};

  \tikzstyle{every node}=[fill=red!20]
  \foreach \from/\to in {a/b,b/c,c/d,d/e,e/f,f/g,g/h,h/a}
    \draw [->] (\from) -- (\to) node[midway] {\from--\to};
\end{tikzpicture}
```

- `swap` This option exchanges the roles of `left` and `right` in automatic placement. That is, if `left` is the current `auto` placement, `right` is set instead and the other way round.

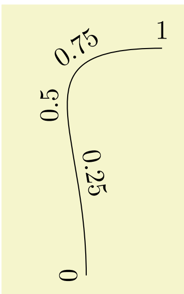



```
\begin{tikzpicture}[auto]
  \draw[help lines,use as bounding box] (0,-.5) grid (4,5);

  \draw (0.5,0) .. controls (9,6) and (-5,6) .. (3.5,0)
    \foreach \pos in {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
      {node [pos=\pos,swap,fill=red!20] {\pos}}
    \foreach \pos in {0.025,0.2,0.4,0.6,0.8,0.975}
      {node [pos=\pos,fill=blue!20] {\pos}};
\end{tikzpicture}
```

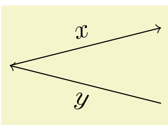



```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,auto]
  \draw[help lines] (0,0) grid (3,2);

  \node[state] (q_0)                        {$q_0$};
  \node[state] (q_1) [above right of=q_0] {$q_1$};
  \node[state] (q_2) [below right of=q_0] {$q_2$};
  \node[state] (q_3) [below right of=q_1] {$q_3$};

  \path[->] (q_0) edge                node        {0} (q_1)
                  edge                node [swap] {1} (q_2)
            (q_1) edge                node        {1} (q_3)
                  edge [loop above] node          {0} ()
            (q_2) edge                node [swap] {0} (q_3)
                  edge [loop below] node          {1} ();
\end{tikzpicture}
```

- `sloped` This option causes the node to be rotated such that a horizontal line becomes a tangent to the curve. The rotation is normally done in such a way that text is never "upside down." To get upside-down text, use can use `[rotate=180]` or `[allow upside down]`, see below.

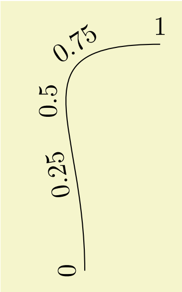



```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    \foreach \p in {0,0.25,...,1} {node[sloped,above,pos=\p]{\p}};
```

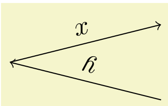



```
\begin{tikzpicture}[->]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

- `allow upside down=`⟨*true or false*⟩ If set to `true`, Ti*k*Z will not "righten" upside down text.



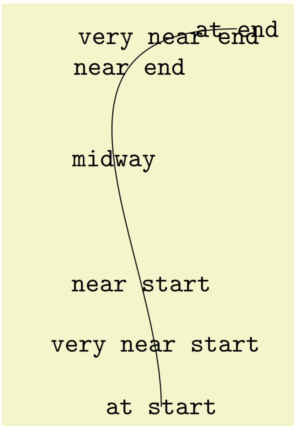

```
\tikz [allow upside down]
  \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    \foreach \p in {0,0.25,...,1} {node[sloped,above,pos=\p]{\p}};
```




```
\begin{tikzpicture}[->,allow upside down]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

There exist styles for specifying positions a bit less "technically":

- `style=midway` is set to `pos=0.5`.




```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:3cm) .. (1,5)
       node[at end]          {|at end|}
       node[very near end]   {|very near end|}
       node[near end]        {|near end|}
       node[midway]          {|midway|}
       node[near start]      {|near start|}
       node[very near start] {|very near start|}
       node[at start]        {|at start|};
```

- `style=near start` is set to `pos=0.25`.
- `style=near end` is set to `pos=0.75`.
- `style=very near start` is set to `pos=0.125`.
- `style=very near end` is set to `pos=0.875`.
- `style=at start` is set to `pos=0`.
- `style=at end` is set to `pos=1`.

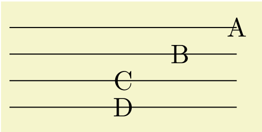
13.8 Placing Nodes on a Line or Curve Implicitly

When you wish to place a node on the line `(0,0) -- (1,1)`, it is natural to specify the node not following the `(1,1)`, but "somewhere in the middle." This is, indeed, possible and you can write `(0,0) -- node{a} (1,1)` to place a node midway between `(0,0)` and `(1,1)`.

What happens is the following: The syntax of the line-to path operation is actually `-- node`⟨*node specification*⟩⟨*coordinate*⟩. (It is even possible to give multiple nodes in this way.) When the optional `node` is encountered, that is, when the `--` is directly followed by `node`, then the specification(s) are read and "stored away." Then, after the ⟨*coordinate*⟩ has finally been reached, they are inserted again, but with the `pos` option set.

There are two things to note about this: When a node specification is "stored," its catcodes become fixed. This means that you cannot use overly complicated verbatim text in them. If you really need, say, a verbatim text, you will have to put it in a normal node following the coordinate and add the `pos` option.

Second, which `pos` is chosen for the node? The position is inherited from the surrounding scope. However, this holds only for nodes specified in this implicit way. Thus, if you add the option `[near end]` to a scope, this does not mean that *all* nodes given in this scope will be put on near the end of lines. Only the nodes for which an implicit `pos` is added will be placed near the end. Typically, this is what you want. Here are some examples that should make this clearer:



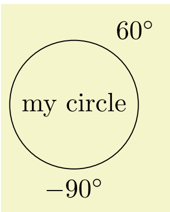

```
\begin{tikzpicture}[near end]
  \draw (0cm,4em) -- (3cm,4em) node{A};
  \draw (0cm,3em) --           node{B}          (3cm,3em);
  \draw (0cm,2em) --           node[midway] {C} (3cm,2em);
  \draw (0cm,1em) -- (3cm,1em) node[midway] {D} ;
\end{tikzpicture}
```

Like the line-to operation, the curve-to operation `..` also allows you to specify nodes "inside" the operation. After both the first `..` and also after the second `..` you can place node specifications. Like for the `--` operation, these will be collected and then reinserted after the operation with the `pos` option set.

13.9 The Label and Pin Options

In addition to the `node` path operation, nodes can also be added using the `label` and the `pin` option. This is mostly useful for simple nodes.

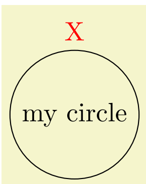
- `label=[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*text*⟩ When this option is given to a `node` operation, it causes *another* node to be added to the path after the current node has been finished. This extra node will have the text ⟨*text*⟩. It is placed according to the following rule: Suppose the `node` currently under construction is called `main node` and let us call the label node `label node`. Then the anchor of `label node` is placed at `main node.`⟨*angle*⟩. The anchor that is chosen depends on the ⟨*angle*⟩. If the ⟨*angle*⟩ lies between $-3°$ and $+3°$, then the anchor `west` is chosen, which causes `label node` to be placed right of the right end `main node`. If ⟨*angle*⟩ lies between 4° and 86°, the anchor `south west` is chosen, causing the `label node` to be placed above and right of the `main node`; and so on.



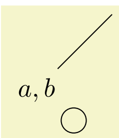

```
\tikz
  \node [circle,draw,label=60:$60^\circ$,label=below:$-90^\circ$] {my circle};
```

As can be seen in the above example, instead of specifying ⟨*angle*⟩ as a number, it is also possible to use `left`, `right`, `above`, `above left`, and so on.

You can pass ⟨*options*⟩ to the node `label node`. For this, you provide the options in square brackets before the ⟨*angle*⟩. If you do so, you need to add braces around the whole argument of the `label` option and this is also the case if you have brackets or commas or semicolons or anything special in the ⟨*text*⟩.




```
\tikz \node [circle,draw,label={[red]above:X}] {my circle};
```

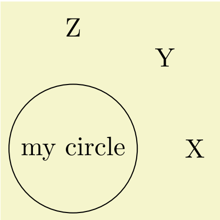



```
\begin{tikzpicture}
  \node [circle,draw,label={[name=label node]above left:$a,b$}] {};
  \draw (label node) -- +(1,1);
\end{tikzpicture}
```

If you provide multiple `label` options, then multiple extra label nodes are added in the order they are given.

The following styles influence how labels are drawn:

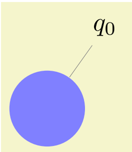
– `label distance`=⟨*distance*⟩ The ⟨*distance*⟩ is additionally inserted between the main node and the label node. The default is `0pt`.




```
\tikz[label distance=5mm]
  \node [circle,draw,label=right:X,
                   label=above right:Y,
                   label=above:Z]       {my circle};
```

– `style=every label` This style is used in every node created by the `label` option. The default is `draw=none,fill=none`.

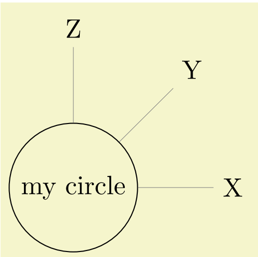
- `pin=[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*text*⟩ This is option is quite similar to the `label` option, but there is one difference: In addition to adding a extra node to the picture, it also adds an edge from this node to the main node. This causes the node to look like a pin that has been added to the main node:




```
\tikz \node [circle,fill=blue!50,minimum size=1cm,pin=60:$q_0$] {};
```

The meaning of the ⟨*options*⟩ and the ⟨*angle*⟩ and the ⟨*text*⟩ is exactly the same as for the `node` option. Only, the options and styles the influence the way pins look are different:

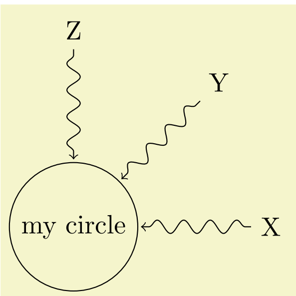
– `pin distance`=⟨*distance*⟩ This ⟨*distance*⟩ is used instead of the `label distance` for the distance between the main node and the label node. The default is `3ex`.




```
\tikz[pin distance=1cm]
  \node [circle,draw,pin=right:X,
                   pin=above right:Y,
                   pin=above:Z]       {my circle};
```

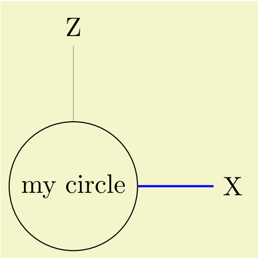
– `style=every pin` This style is used in every node created by the `pin` option. The default is `draw=none,fill=none`.

– `style=every pin edge` This style is used in every edge created by the `pin` optins. The default is `help lines`.

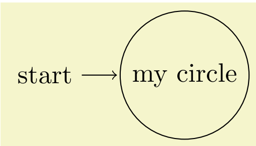



```
\tikzstyle{every pin edge}=[<-,shorten <=1pt,snake=snake,line before snake=4pt]
\tikz[pin distance=15mm]
  \node [circle,draw,pin=right:X,
                   pin=above right:Y,
                   pin=above:Z]       {my circle};
```

– `pin edge`=⟨*options*⟩ This option can be used to set the options that are to be used in the edge created by the `pin` option. The default is empty.




```
\tikz[pin distance=10mm]
  \node [circle,draw,pin={[pin edge={blue,thick}]right:X},
                   pin=above:Z]       {my circle};
```




```
\tikzstyle{every pin edge}=[]
\tikzstyle{initial}=[pin={[pin distance=5mm,
                           pin edge={<-,shorten <=1pt}]left:start}]
\tikz \node [circle,draw,initial] {my circle};
```

13.10 Connecting Nodes: Using Nodes as Coordinates

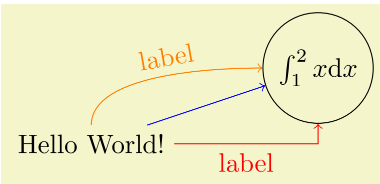
Once you have defined a node and given it a name, you can use this name to reference it. This can be done in two ways, see also Section 10.2.3. Suppose you have said `\path(0,0) node(x) {Hello World!};` in order to define a node named `x`.

1. Once the node `x` has been defined, you can use `(x.`⟨*anchor*⟩`)` wherever you would normally use a normal coordinate. This will yield the position at which the given ⟨*anchor*⟩ is in the picture. Note that transformations do not apply to this coordinate, that is, `(x.north)` will be the northern anchor of `x` even if you have said `scale=3` or `xshift=4cm`. This is usually what you would expect.

2. You can also just use `(x)` as a coordinate. In most cases, this gives the same coordinate as `(x.center)`. Indeed, if the `shape` of `x` is `coordinate`, then `(x)` and `(x.center)` have exactly the same effect.

   However, for most other shapes, some path construction operations like `--` try to be "clever" when this they are asked to draw a line from such a coordinate or to such a coordinate. When you say `(x)--(1,1)`, the `--` path operation will not draw a line from the center of `x`, but *from the border* of `x` in the direction going towards `(1,1)`. Likewise, `(1,1)--(x)` will also have the line end on the border in the direction coming from `(1,1)`.

   In addition to `--`, the curve-to path operation `..` and the path operations `-|` and `|-` will also handle nodes without anchors correctly. Here is an example, see also Section 10.2.3:




```
\begin{tikzpicture}
  \path (0,0) node             (x) {Hello World!}
        (3,1) node[circle,draw](y) {$\int_1^2 x \mathrm d x$};

  \draw[->,blue]   (x) -- (y);
  \draw[->,red]    (x) -| node[near start,below] {label} (y);
  \draw[->,orange] (x) .. controls +(up:1cm) and +(left:1cm) .. node[above,sloped] {label} (y);
\end{tikzpicture}
```

13.11 Connecting Nodes: Using the Edge Operation

The `edge` operation works like a `to` operation that is added after the main path has been drawn, much like a node is added after the main path has been drawn. This allows you to have each `edge` to have a different appearance. As the `node` operation, an `edge` temporarily suspends the construction of the current path and a new path p is constructed. This new path p will be drawn after the main path has been drawn. Note that p can be totally different from the main path with respect to its options. Also note that if there are several `to` and/or `node` operations in the main path, each creates its own path(s) and they are drawn in the order that they are encountered on the path.

\path ... edge[⟨*options*⟩] ⟨*nodes*⟩ (⟨*coordinate*⟩) ...;

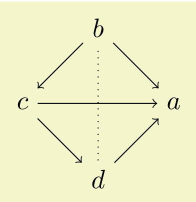
The effect of the edge operation is that after the main path the following path is added to the picture:

\path[every edge,⟨*options*⟩] (\tikztostart) ⟨*path*⟩;

Here, ⟨*path*⟩ is the to path. Note that, unlike the path added by the to operation, the (\tikztostart) is added before the ⟨*path*⟩ (which is unnecessary for the to operation, since this coordinate is already part of the main path).

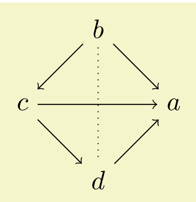
The \tikztostart is the last coordinate on the path just before the edge operation, just as for the node or to operations. However, there is one exception to this rule: If the edge operation is directly preceded by a node operation, then this just-declared node is the start coordinate (and not, as would normally be the case, the coordinate where this just-declared node is placed – a small, but subtle difference). In this regard, edge differs from both node and to.

If there are several edge operations in a row, the start coordinate is the same for all of them as their target coordiantes are not, after all, part of the main path. The start coordinate is, thus, the coordinate preceding the first edge operation. This is similar to nodes insofar as the edge operation does not modify the current path at all. In particular, it does not change the last coordinate visited, see the following example:



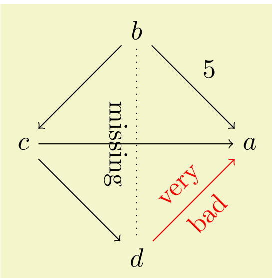

```
\begin{tikzpicture}
  \node (a) at   (0:1) {$a$};
  \node (b) at  (90:1) {$b$} edge [->]     (a);
  \node (c) at (180:1) {$c$} edge [->]     (a)
                             edge [<-]     (b);
  \node (d) at (270:1) {$d$} edge [->]     (a)
                             edge [dotted] (b)
                             edge [<-]     (c);
\end{tikzpicture}
```

A different way of specifying the above graph using the edge operation is the following:




```
\begin{tikzpicture}
  \foreach \name/\angle in {a/0,b/90,c/180,d/270}
    \node (\name) at (\angle:1) {$\name$};

  \path[->] (b) edge (a)
                edge (c)
                edge [-,dotted] (d)
            (c) edge (a)
                edge (d)
            (d) edge (a);
\end{tikzpicture}
```

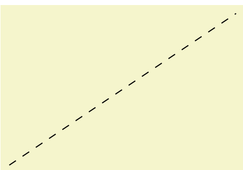
As can be seen, the path of the edge operation inherits the options from the main path, but you can locally overrule them.




```
\begin{tikzpicture}
  \foreach \name/\angle in {a/0,b/90,c/180,d/270}
    \node (\name) at (\angle:1.5) {$\name$};

  \path[->] (b) edge            node[above right]  {$5$}     (a)
                edge                                         (c)
                edge [-,dotted] node[below,sloped] {missing} (d)
            (c) edge                                         (a)
                edge                                         (d)
            (d) edge [red]      node[above,sloped] {very}
                                node[below,sloped] {bad}     (a);
\end{tikzpicture}
```

Instead of every to, the style every edge is installed at the beginning of the main path.

- style=every edge this is is draw by default.



```
\begin{tikzpicture}
  \tikzstyle{every to}=[draw,dashed]
  \path (0,0) to (3,2);
\end{tikzpicture}
```

13.12 Referencing Nodes Outside the Current Pictures

13.12.1 Referencing a Node in a Different Picture

It is possible (but not quite trivial) to reference nodes in pictures other than the current one. This means that you can create a picture and a node therein and, later, you can draw a line from some other position to this node.

To reference nodes in different pictures, proceed as follows:

1. You need to add the `remember picture` option to all pictures that contain nodes that you wish to reference and also to all pictures from which you wish to reference a node in another picture.

2. You need to add the `overlay` option to paths or to whole pictures that contain references to nodes in different pictures. (This option switches the computation of the bounding box off.)

3. You need to use a driver that supports picture remembering (currently, this is only pdfTEX). With the pdfTEX driver you also need to run TEX twice.

(For more details on what is going on behind the scenes, see Section 49.3.2.)

Let us have a look at the effect of these options.

- `remember picture=`⟨*true or false*⟩ This option tells Ti*k*Z that it should attempt to remember the position of the current picture on the page. This attempt may fail depending on which backend driver is used. Also, even if remembering works, the position may only be available on a second run of TEX.

  Provided that remebering works, you may consider saying

  ```
  \tikzsytle{every picture}+=[remember picture]
  ```

  to make Ti*k*Z remember all pictures. This will add one line in the `.aux` file for each picture in your document – which typically is not very much. Then, you do not have to worry about remembered pictures at all.

- `overlay` This option is mainly intended for use when nodes in other pictures are referenced, but you can also use it in other situations. The effect of this option is that everything within the current scope is not taken into consideration when the bounding box of the current picture is computed.

  You need to specify this option on all paths (or at least on all parts of paths) that contain a reference to a node in another picture. The reason is that, otherwise, Ti*k*Z will attempt to make the current picture large enough to encompass *the node in the other picture.* However, on a second run of TEX this will create an even bigger picture, leading to larger and larger pictures. Unless you know what you are doing, I suggest specifying the `overlay` option with all pictures that contain references to other pictures.

Let us now have a look at a few examples. These examples work only if this document is processed with a driver that supports picture remembering.

Inside the current text we place two pictures, containing nodes named `n1` and `n2`, using

```
\tikz[remember picture] \node[circle,fill=red!50] (n1) {};
```

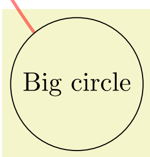
which yields , and

```
\tikz[remember picture] \node[fill=blue!50] (n2) {};
```

yielding the node . To connect these nodes, we create another picture using the `overlay` option and also the `remember picture` option.

```
\begin{tikzpicture}[remember picture,overlay]
  \draw[->,very thick] (n1) -- (n2);
\end{tikzpicture}
```

Note that the last picture is seemingly empty. What happens is that it has zero size and contains an arrow that lies well outside its bounds. As a last example, we connect a node in another picture to the first two nodes. Here, we provide the `overlay` option only with the line that we do not wish to count as part of the picture.




```
\begin{tikzpicture}[remember picture]
  \node (c) [circle,draw] {Big circle};

  \draw [overlay,->,very thick,red,opacity=.5]
    (c) to[bend left] (n1) (n1) -| (n2);
\end{tikzpicture}
```

13.12.2 Referencing the Current Page Node – Absolute Positioning

There is a special node called `current page` that can be used to access the current page. It is a node of shape rectangle whose `south west` anchor is the lower left corner of the page and whose `north east` anchor is the upper right corner of the page. While this node is handled in a special way internally, you can reference it as if it were defined in some remembered picture other than the current one. Thus, by giving the `remembered picture` and the `overlay` options to a picture, you can position nodes *absolutely* on a page.

The first example places some text in the lower left corner of the current page:

```
\begin{tikzpicture}[remember picture,overlay]
  \node [xshift=1cm,yshift=1cm] at (current page.south west)
        [text width=7cm,fill=red!20,rounded corners,above right]
  {
    This is an absolutely positioned text in the
    lower left corner. No shipout-hackery is used.
  };
\end{tikzpicture}
```

The next example adds a circle in the middle of the page.

```
\begin{tikzpicture}[remember picture,overlay]
  \draw [line width=1mm,opacity=.25]
    (current page.center) circle (3cm);
\end{tikzpicture}
```

The final example overlays some text over the page (depending on where this example is found on the page, the text may also be behind the page).

```
\begin{tikzpicture}[remember picture,overlay]
  \node [rotate=60,scale=10,text opacity=0.2]
    at (current page.center) {Example};
\end{tikzpicture}
```

13.13 Predefined Shapes

PGF and Ti*k*Z define three shapes, by default:

- `rectangle`,
- `circle`, and
- `coordinate`.

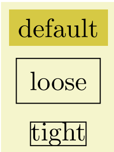
Example

This is an absolutely positioned text in the lower left corner. No shipout-hackery is used.

By loading library packages, you can define more shapes like ellipses or diamonds; see the library section for the complete list of shapes.

The exact behaviour of these shapes differs, shapes defined for more special purposes (like a, say, transistor shape) will have even more custom behaviors. However, there are some options that apply to most shapes:

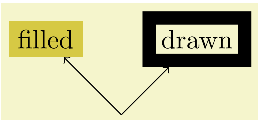
- `inner sep`=⟨*dimension*⟩ An additional (invisible) separation space of ⟨*dimension*⟩ will be added inside the shape, between the text and the shape's background path. The effect is as if you had added appropriate horizontal and vertical skips at the beginning and end of the text to make it a bit "larger." The default `inner sep` is the size of a normal space.




```
\begin{tikzpicture}
  \draw (0,0)     node[inner sep=0pt,draw] {tight}
        (0cm,2em) node[inner sep=5pt,draw] {loose}
        (0cm,4em) node[fill=examplefill]   {default};
\end{tikzpicture}
```

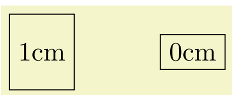
- `inner xsep`=⟨*dimension*⟩ Specifies the inner separation in the x-direction, only.
- `inner ysep`=⟨*dimension*⟩ Specifies the inner separation in the y-direction, only.
- `outer sep`=⟨*dimension*⟩ This option adds an additional (invisible) separation space of ⟨*dimension*⟩ outside the background path. The main effect of this option is that all anchors will move a little "to the outside."

  The default for this option is half the line width. When the default is used and when the background path is draw, the anchors will lie exactly on the "outside border" of the path (not on the path itself). When the shape is filled, but not drawn, this may not be desirable. In this case, the `outer sep` should be set to zero point.



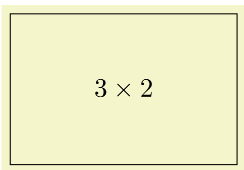

```
\begin{tikzpicture}
  \draw[line width=5pt]
    (0,0) node[outer sep=0pt,fill=examplefill]     (f) {filled}
    (2,0) node[inner sep=.5\pgflinewidth+2pt,draw] (d) {drawn};

  \draw[->] (1,-1) -- (f);
  \draw[->] (1,-1) -- (d);
\end{tikzpicture}
```

- `outer xsep`=⟨*dimension*⟩ Specifies the outer separation in the x-direction, only.
- `outer ysep`=⟨*dimension*⟩ Specifies the outer separation in the y-direction, only.
- `minimum height`=⟨*dimension*⟩ This option ensures that the height of the shape (including the inner, but ignoring the outer separation) will be at least ⟨*dimension*⟩. Thus, if the text plus the inner separation is not at least as large as ⟨*dimension*⟩, the shape will be enlarged appropriately. However, if the text is already larger than ⟨*dimension*⟩, the shape will not be shrunk.



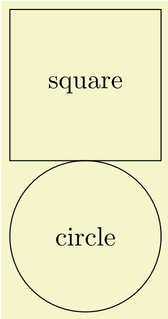

```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=1cm,draw] {1cm}
        (2,0) node[minimum height=0cm,draw] {0cm};
\end{tikzpicture}
```

- `minimum width`=⟨*dimension*⟩ same as `minimum height`, only for the width.



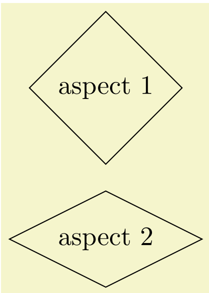

```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=2cm,minimum width=3cm,draw] {$3 \times 2$};
\end{tikzpicture}
```

- minimum size=⟨*dimension*⟩ sets both the minimum height and width at the same time.



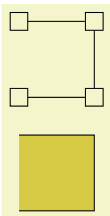

```
\begin{tikzpicture}
  \draw (0,0)  node[minimum size=2cm,draw] {square};
  \draw (0,-2) node[minimum size=2cm,draw,circle] {circle};
\end{tikzpicture}
```

- aspect=⟨*aspect ratio*⟩ sets a desired aspect ratio for the shape. For the `diamond` shape, this option sets the ratio between width and height of the shape.




```
\begin{tikzpicture}
  \draw (0,0)  node[aspect=1,diamond,draw] {aspect 1};
  \draw (0,-2) node[aspect=2,diamond,draw] {aspect 2};
\end{tikzpicture}
```

The `coordinate` shape is handled in a special way by Ti*k*Z. When a node `x` whose shape is `coordinate` is used as a coordinate `(x)`, this has the same effect as if you had said `(x.center)`. None of the special "line shortening rules" apply in this case. This can be useful since, normally, the line shortening causes paths to be segmented and they cannot be used for filling. Here is an example that demonstrates the difference:



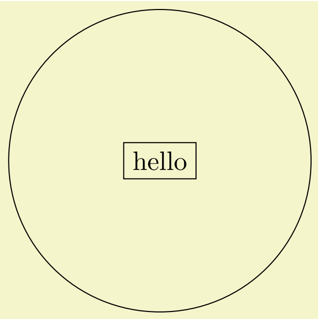
```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \path[yshift=1.5cm,shape=rectangle]
    (0,0) node(a1){} (1,0) node(a2){}
    (1,1) node(a3){} (0,1) node(a4){};
  \filldraw[fill=examplefill] (a1) -- (a2) -- (a3) -- (a4);

  \path[shape=coordinate]
    (0,0) coordinate(b1) (1,0) coordinate(b2)
    (1,1) coordinate(b3) (0,1) coordinate(b4);
  \filldraw[fill=examplefill] (b1) -- (b2) -- (b3) -- (b4);
\end{tikzpicture}
```

13.14 Executing Code After Nodes

It is possible to add a path right after a node using the option `after node path`. The idea is that a style might use this option to add some additional stuff to the node that has just been typeset.

- after node path=⟨*path*⟩ The ⟨*path*⟩ is added to the main path right after the node, as if you had given the path thereafter. This option can only be given inside the option list of a node and multiple calls of this option accumulate.

  Inside the ⟨*path*⟩ you have access to the node that has just been created via the macro `\tikzlastnode`.




```
\tikz
  \draw node [draw,after node path={(\tikzlastnode) circle (2cm)}]
    {hello};
```

Note that in the above example, if we had written `\path` instead of `\draw`, the circle would not have been drawn since the circle is part of the main path, not part of the node itself.

`\tikzaddafternodepathoption`{⟨*code*⟩}

This command allows you to specify that the ⟨*code*⟩ should be executed at the beginning of the `after node path` of the current node. The code will also be executed immediately, but also again at the beginning of an `after node path`.

14 Matrices and Alignment

14.1 Overview

When creating pictures, one often faces the problem of correctly aligning parts of the picture. For example, you might wish that the base lines of certain nodes should be on the same line and some further nodes should be below these nodes with, say, their centers on a vertical lines. There are different ways of solving such problems. For example, by making clever use of anchors, nearly all such alignment problems can be solved. However, this often leads to complicated code. An often simpler way is to use *matrices*, the use of which is explaied in the current section.

A Ti*k*Z matrix is similar to LaTeX's `{tabular}` or `{array}` environment, only instead of text each cell contains a little picture or a node. The sizes of the cells are automatically adjusted such that they are large enough to contain all the cell contents.

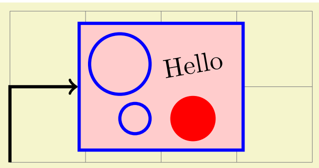
Matrices are a powerful tool and they need to handled with some care. For impatient readers who skip the rest of this section: you *must* end *every* row with `\\`. In particular, the last row *must* be ended with `\\`.

Many of the ideas implemented in Ti*k*Z's matrix support are due to Mark Wibrow – many thanks to Mark at this point!

14.2 Matrices are Nodes

Matrices are special in many ways, but for most purposes matrices are treated like nodes. This means, that you use the `node` path command to create a matrix and you only use a special option, namely the `matrix` option, to signal that the node will contain a matrix. Instead of the usual TeX-box that makes up the `text` part of the node's shape, the matrix is used. Thus, in particular, a matrix can have a shape, this shape can be drawn or filled, it can be used in a tree, and so on. Also, you can refer to the different anchors of a matrix.

- `matrix=`⟨*true or false*⟩ This option can be passed to a `node` path command. It signals that the node will contain a matrix. The default parameter is `true` and should usually be omitted.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,2);
  \node [matrix,fill=red!20,draw=blue,very thick] (my matrix) at (2,1)
  {
    \draw (0,0)   circle (4mm); & \node[rotate=10] {Hello};        \\
    \draw (0.2,0) circle (2mm); & \fill[red]   (0,0) circle (3mm); \\
  };

  \draw [very thick,->] (0,0) |- (my matrix.west);
\end{tikzpicture}
```

The exact syntax of the matrix is explained in the course of this section.

- `style=every matrix` This style is used in every matrix. It is empty by default.

Even more so than nodes, matrices will often be the only object on a path. Because of this, there is a special abbreviation for creating matrices:

`\matrix`

Inside `{tikzpicture}` this is an abbreviation for `\path node[matrix]`.

Even though matrices are nodes, some options do not have the same effect as for normal nodes:

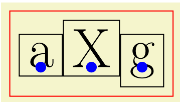
1. Rotations and scaling have no effect on a matrix as a whole (however, you can still transform the contents of the cells normally). Before the matrix is typeset, the rotational and scaling part of the transformation matrix is reset.
2. For multi-part shapes you can only set the `text` part of the node.
3. All options starting with `text` such as `text width` have no effect.

14.3 Cell Pictures

A matrix consists of rows of *cells*. Each row (including the last one!) is ended by the command `\\`. The character `&` is used to separate cells. Inside each cell, you must place commands for drawing a picture, called the *cell picture* in the following. (However, cell pictures are not enclosed in a complete `{pgfpicture}` environment, they are a bit more light-weight. The main difference is that cell pictures cannot have layers.) It is not necessary to specify beforehand how many rows or columns there are going to be and if a row contains less cell pictures than another line, empty cells are automatically added as needed.

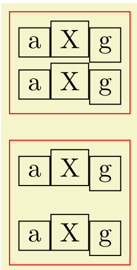
14.3.1 Alignment of Cell Pictures

For each cell picture a bounding box is computed. These bounding boxes and the origins of the cell pictures determine how the cells are aligned. Let us start with the rows: Consider the cell pictures on the first row. Each has a bounding box and somewhere inside this bounding box the origin of the cell picture can be found (the origin might even lie outside the bounding box, but let us ignore this problem for the moment). The cell pictures are then shifted around such that all origins lie on the same horizontal line. This may make it necessary to shift some cell pictures upwards and other downwards, but it can be done and this yields the vertical alignment of the cell pictures this row. The top of the row is then given by the top of the "highest" cell picture in the row, the bottom of the row is given by the bottom of the lowest cell picture. (To be more precise, the height of the row is the maximum y-value of any of the bounding boxes and the depth of the row is the negated minimum y-value of the bounding boxes).



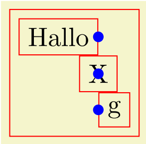

```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw=black,anchor=base,font=\huge]

  \matrix [draw=red]
  {
    \node {a}; \fill[blue] (0,0) circle (2pt); &
    \node {X}; \fill[blue] (0,0) circle (2pt); &
    \node {g}; \fill[blue] (0,0) circle (2pt); \\
  };
\end{tikzpicture}
```

Each row is aligned in this fashion: For each row the cell pictures are vertically aligned such that the origins lie on the same line. Then the second row is placed below the first row such that the bottom of the first row touches the top of the second row (unless a `row sep` is used to add a bit of space). Then the bottom of the second row touches the top of the third row, and so on. Typically, each row will have an individual height and depth.

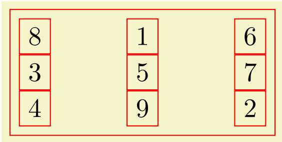



```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw=black,anchor=base]

  \matrix [draw=red]
  {
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
  };

  \matrix [row sep=3mm,draw=red] at (0,-2)
  {
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
  };
\end{tikzpicture}
```

Let us now have a look at the columns. The rules for how the pictures on any given column are aligned are very similar to the row alignment: Consider all cell pictures in the first column. Each is shifted horizontally such that the origins lie on the same vertical line. Then, the left end of the column is at the left end of the bounding box that protrudes furthest to the left. The right end of the column is at the right end of the bounding box that protrudes furthest to the left. This fixes the horizontal alignment of the cell pictures in the first column and the same happens the cell pictures in the other columns. Then, the right end of the first column touches the left end of the second column (unless `column sep` is used). The right end of the second column touches the left end of the third column, and so on. (Internally, two columns are actually used to achieve the desired horizontal alignment, but that is only and implementation detail.)




```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \matrix [draw=red]
  {
    \node[left]  {Hallo}; \fill[blue] (0,0) circle (2pt); \\
    \node        {X};     \fill[blue] (0,0) circle (2pt); \\
    \node[right] {g};     \fill[blue] (0,0) circle (2pt); \\
  };
\end{tikzpicture}
```

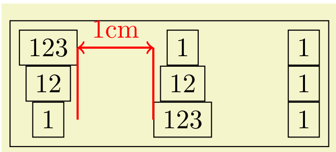



```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \matrix [draw=red,column sep=1cm]
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```

14.3.2 Setting and Adjusting Column and Row Spacing

There are different ways of setting and adjusting the spacing between columns and rows. First, you can use the options `column sep` and `row sep` to set a default spacing for all rows and all columns. Second, you can add options to the `&` character and the `\\` command to adjust the spacing between two specific columns or rows. Additionally, you can specify whether the space between two columns or rows should be considered between the origins of cells in the column or row or between their borders.

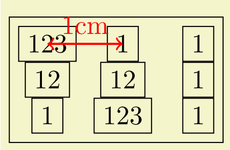
- `column sep`=⟨*spacing list*⟩ This option sets a default space that is added between every two columns. This space can be positive or negative and is zero by default. The ⟨*spacing list*⟩ normally contains a single dimension like `2pt`.




```
\begin{tikzpicture}
  \matrix [draw,column sep=1cm,nodes=draw]
  {
    \node(a) {123}; & \node (b) {1};   & \node {1}; \\
    \node    {12};  & \node     {12};  & \node {1}; \\
    \node(c) {1};   & \node (d) {123}; & \node {1}; \\
  };
  \draw [red,thick]  (a.east) -- (a.east |- c)
                     (d.west) -- (d.west |- b);
  \draw [<->,red,thick] (a.east) -- (d.west |- b)
    node [above,midway] {1cm};
\end{tikzpicture}
```

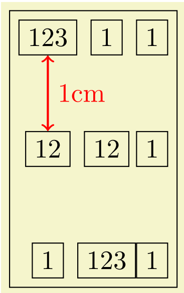
More generally, the ⟨*spacing list*⟩ may contain a whole list of numbers, separated by commas, and occurrences of the two key words `between origins` and `between borders`. The effect of specifying such a list is the following: First, all numbers occurring in the list are simply added to compute the final spacing. Second, concerning the two keywords, the last occurrence of one of the keywords is important. If the last occurrence is `between borders` or if neither occurs, then the space is inserted between the two columns normally. However, if the last occurs is `between origins`, then the following happens: The distance between the columns is adjusted such that the difference between the origins of all the cells in the first column (remember that they all lie on straight line) and the origins of all the cells in the second column is exactly the given distance.

The `between origins` option can only be used for columns mentioned in the first row, that is, you cannot specify this option for columns introduced only in later rows.

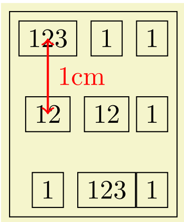



```
\begin{tikzpicture}
  \matrix [draw,column sep={1cm,between origins},nodes=draw]
  {
    \node(a) {123}; & \node (b) {1};   & \node {1}; \\
    \node    {12};  & \node     {12};  & \node {1}; \\
    \node    {1};   & \node     {123}; & \node {1}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {1cm};
\end{tikzpicture}
```

- `row sep`=⟨*spacing list*⟩ This option works like `column sep`, only for rows. Here, too, you can specify whether the space is added between the lower end of the first row and the upper end of the second row, or whether the space is computed between the origins of the two rows.

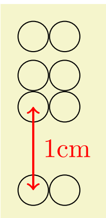



```
\begin{tikzpicture}
  \matrix [draw,row sep=1cm,nodes=draw]
  {
    \node (a) {123}; & \node {1};   & \node {1}; \\
    \node (b) {12};  & \node {12};  & \node {1}; \\
    \node     {1};   & \node {123}; & \node {1}; \\
  };
  \draw [<->,red,thick] (a.south) -- (b.north) node [right,midway] {1cm};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,row sep={1cm,between origins},nodes=draw]
  {
    \node (a) {123}; & \node {1};   & \node {1}; \\
    \node (b) {12};  & \node {12};  & \node {1}; \\
    \node     {1};   & \node {123}; & \node {1}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [right,midway] {1cm};
\end{tikzpicture}
```

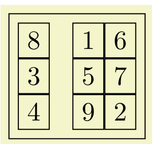
The row-end command `\\` allows you to provide an optional argument, which must be a dimension. This dimension will be added to the list in `row sep`. This means that, firstly, any numbers you list in this argument will be added as an extra row separation between the line being ended and the next line and, secondly, you can use the keywords `between origins` and `between borders` to locally overrule the standard setting for this line pair.



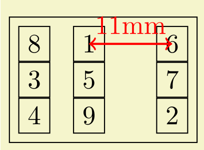

```
\begin{tikzpicture}
  \matrix [row sep=1mm]
  {
    \draw (0,0) circle (2mm); & \draw (0,0) circle (2mm); \\
    \draw (0,0) circle (2mm); & \draw (0,0) circle (2mm); \\[-1mm]
    \draw (0,0) coordinate (a) circle (2mm); &
    \draw (0,0) circle (2mm); \\[1cm,between origins]
    \draw (0,0) coordinate (b) circle (2mm); &
    \draw (0,0) circle (2mm); \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [right,midway] {1cm};
\end{tikzpicture}
```

The cell separation character `&` also takes an optional argument, which must also be a spacing list. This spacing list is added to the `column sep` having a similar effect as the option for the `\\` command for rows.

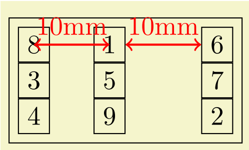
This optional spacing list can only be given the first time a new column is started (usually in the first row), subsequent usages of this option in later rows have no effect.




```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep=1mm]
  {
    \node {8}; &[2mm] \node{1}; &[-1mm] \node {6}; \\
    \node {3}; &      \node{5}; &       \node {7}; \\
    \node {4}; &      \node{9}; &       \node {2}; \\
  };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep=1mm]
  {
    \node {8}; &[2mm] \node(a){1}; &[1cm,between origins] \node(b){6}; \\
    \node {3}; &      \node   {5}; &                      \node   {7}; \\
    \node {4}; &      \node   {9}; &                      \node   {2}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {11mm};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep={1cm,between origins}]
  {
    \node (a) {8}; & \node (b) {1}; &[between borders] \node (c) {6}; \\
    \node     {3}; & \node     {5}; &                  \node     {7}; \\
    \node     {4}; & \node     {9}; &                  \node     {2}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm};
  \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway] {10mm};
\end{tikzpicture}
```

14.3.3 Cell Styles and Options

For following style and option are useful for changing the appearance of the all cell pictures:

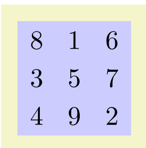
- `style=every cell` This style is installed at the beginning of each cell picture. Note that setting this style to `draw` will *not* cause all nodes to be drawn since the `draw` option has to be passed to each node individually.

  Inside this style (and inside all cells), the current row and column number are accessible via the counters `\pgfmatrixcurrentrow` and `\pgfmatrixcurrentcolumn`.

- `cells=`⟨*options*⟩ This option adds the ⟨*options*⟩ to the style `every cell`. It is just a shorthand for `set style={{every cell}+=[`⟨*options*⟩`]}`.

- `nodes=`⟨*options*⟩ This option adds the ⟨*options*⟩ to the style `every node`. It is just a shorthand for `set style={{every node}+=[`⟨*options*⟩`]}`.

  The main use of this option is the install some options for the nodes *inside* the matrix that should not apply to the matrix *itself*.




```
\begin{tikzpicture}
  \matrix [nodes={fill=blue!20,minimum size=5mm}]
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```

The next set of styles can be used to change the appearance of certain rows, columns, or cells. If more than one of these styles is defined, they are executed in the below order (the `every cell` style is executed before all of the below).

- `style=column` ⟨*number*⟩ This style is used for every cell in column ⟨*number*⟩.
- `style=every odd column` This style is used for every cell in an odd column.
- `style=every even column` This style is used for every cell in an even column.
- `style=row` ⟨*number*⟩ This style is used for every cell in row ⟨*number*⟩.
- `style=every odd row` This style is used for every cell in an odd row.
- `style=every even row` This style is used for every cell in an even row.
- `style=row` ⟨*row number*⟩ `column` ⟨*column number*⟩ This style is used for the cell in row ⟨*row number*⟩ and column ⟨*column number*⟩.

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
  \tikzstyle{row 1}=[red]
  \tikzstyle{column 2}=[green!50!black]
  \tikzstyle{row 3 column 3}=[blue]

  \matrix
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```

You can use the `column` ⟨*number*⟩ option to change the alignment for different columns.

123	456	789
12	45	78
1	4	7

```
\begin{tikzpicture}
  \tikzstyle{column 1}=[anchor=base west]
  \tikzstyle{column 2}=[anchor=base east]
  \tikzstyle{column 3}=[anchor=base]
  \matrix
  {
    \node {123}; & \node{456}; & \node {789}; \\
    \node {12}; & \node{45}; & \node {78}; \\
    \node {1}; & \node{4}; & \node {7}; \\
  };
\end{tikzpicture}
```

In many matrices all cell pictures have nearly the same code. For example, cells typically start with `\node{` and end `};`. The following options allow you to execute such code in all cells:

- `execute at begin cell`=⟨*code*⟩ The code will be executed at the beginning of each nonempty cell.
- `execute at end cell`=⟨*code*⟩ The code will be executed at the end of each nonempty cell.
- `execute at empty cell`=⟨*code*⟩ The code will be executed inside each empty cell.

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
  \tikzstyle{matrix of nodes}=[
    execute at begin cell=\node\bgroup,
    execute at end cell=\egroup;%
  ]
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

8	1	–
3	–	7
–	–	2

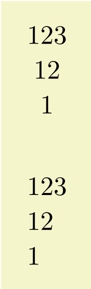
```
\begin{tikzpicture}
  \tikzstyle{matrix of nodes}=[
    execute at begin cell=\node\bgroup,
    execute at end cell=\egroup;,%
    execute at empty cell=\node{--};%
  ]
  \matrix [matrix of nodes]
  {
    8 & 1 &   \\
    3 &   & 7 \\
      &   & 2 \\
  };
\end{tikzpicture}
```

The `matrix` library defines a number of styles that make use of the above options.

14.4 Anchoring a Matrix

Since matrices are nodes, they can be anchored in the usual fashion using the `anchor` option. However, there are two ways to influence this placement further. First, the following option is often useful:

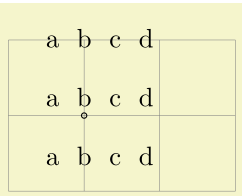
- matrix anchor=⟨*anchor*⟩ This option has the same effect as `anchor`, but the option applies only to the matrix itself, not to the cells inside. If you just say `anchor=north` as an option to the matrix node, all nodes inside matrix will also have this anchor, unless it is explicitly set differently for each node. By comparison, `matrix anchor` sets the anchor for the matrix, but for the nodes inside the value of `anchor` remain unchanged.




```
\begin{tikzpicture}
  \matrix [matrix anchor=west] at (0,0)
  {
    \node {123}; \\ % still center anchor
    \node {12}; \\
    \node {1}; \\
  };
  \matrix [anchor=west] at (0,-2)
  {
    \node {123}; \\ % inherited west anchor
    \node {12}; \\
    \node {1}; \\
  };
\end{tikzpicture}
```

The second way to anchor a matrix is to use *an anchor of a node inside the matrix*. For this, the `anchor` option has a special effect when given as an argument to a matrix:

- `anchor`=⟨*anchor or node.anchor*⟩ Normally, the argument of this option refers to anchor of the matrix node, which is the node than includes all of the stuff of the matrix. However, you can also provide an argument of the form ⟨*node*⟩`.`⟨*anchor*⟩ where ⟨*node*⟩ must be node defined inside the matrix and ⟨*anchor*⟩ is an anchor of this node. In this case, the whole matrix is shifted around in such a way that this particular anchor of this particular node lies at the `at` position of the matrix. The same is true for `matrix anchor`.

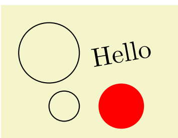



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \matrix[matrix anchor=inner node.south,anchor=base,row sep=3mm] at (1,1)
  {
    \node {a}; & \node             {b}; & \node {c}; & \node {d}; \\
    \node {a}; & \node(inner node) {b}; & \node {c}; & \node {d}; \\
    \node {a}; & \node             {b}; & \node {c}; & \node {d}; \\
  };
  \draw (inner node.south) circle (1pt);
\end{tikzpicture}
```

14.5 Considerations Concerning Active Characters

Even though Ti*k*Z seems to use `&` to separate cells, PGF actually uses a different command to separate cells, namely the command `\pgfmatrixnextcell` and using a normal `&` character will normally fail. What happens is that, Ti*k*Z makes `&` an active character and then defines this character to be equal to `\pgfmatrixnextcell`. In most situations this will work nicely, but sometimes `&` cannot be made active; for instance because the matrix is used in an argument of some macro or the matrix contains nodes that contain normal `{tabular}` environments. In this case you can use the following option to avoid having to type `\pgfmatrixnextcell` each time:

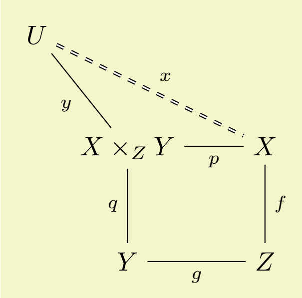
- `ampersand replacement`=⟨*macro name or empty*⟩ If a macro name is provided, this macro will be defined to be equal to `\pgfmatrixnextcell` inside matrices and `&` will not be made active. For instance, you could say `ampersand replacement=\&` and then use & to separate columns as in the following example:



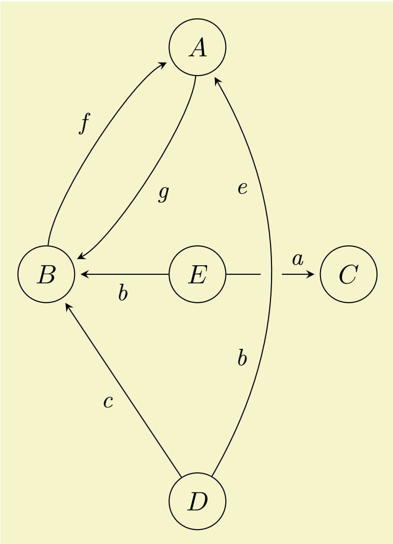

```
\tikz
  \matrix [ampersand replacement=\&]
  {
    \draw (0,0)   circle (4mm); \& \node[rotate=10] {Hello};       \\
    \draw (0.2,0) circle (2mm); \& \fill[red]   (0,0) circle (3mm); \\
  };
```

14.6 Examples

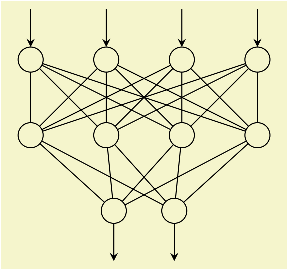
The following examples are adapted from code by Mark Wibrow. The first two redraw pictures from Timothy van Zandt's PSTricks documentation:




```
\begin{tikzpicture}
  \matrix [matrix of math nodes,row sep=1cm]
  {
    |(U)| U &[2mm]                          &[8mm]    \\
            &     |(XZY)| X \times_Z Y  &      |(X)| X \\
            &     |(Y)|   Y             &      |(Z)| Z \\
  };
  \tikzstyle{every node}=[midway,auto,font=\scriptsize]
  \draw [double, dashed] (U)   -- node {$x$} (X);
  \draw                  (X)   -- node {$p$} (X -| XZY.east)
                         (X)   -- node {$f$} (Z)
                               -- node {$g$} (Y)
                               -- node {$q$} (XZY)
                               -- node {$y$} (U);
\end{tikzpicture}
```

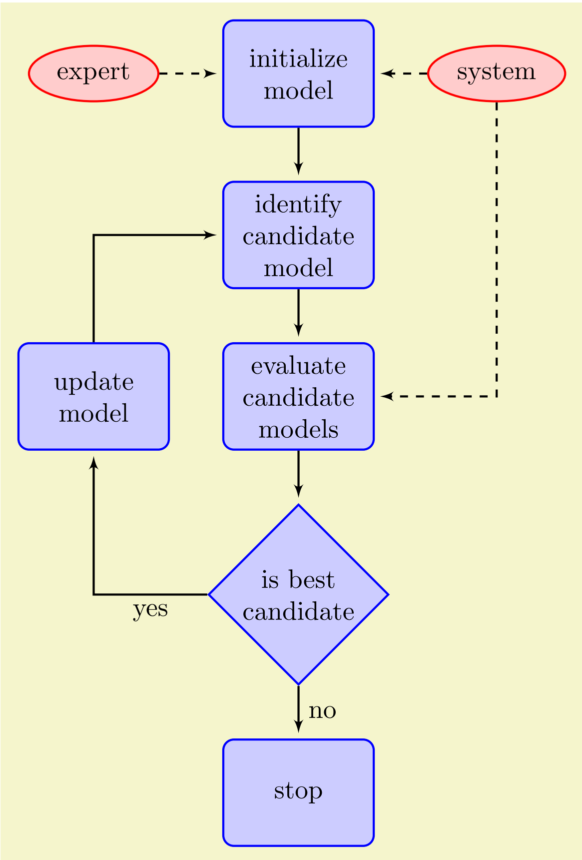



```
\begin{tikzpicture}[>=stealth,->,shorten >=2pt,looseness=.5,auto]
  \matrix [matrix of math nodes,
           column sep={2cm,between origins},
           row sep={3cm,between origins},
           nodes={circle, draw, minimum size=7.5mm}]
  {
            & |(A)| A &         \\
    |(B)| B & |(E)| E & |(C)| C \\
            & |(D)| D           \\
  };
  \tikzstyle{every node}=[font=\small\itshape]
  \draw (A) to [bend left] (B) node [midway]   {g};
  \draw (B) to [bend left] (A) node [midway]   {f};
  \draw (D) --             (B) node [midway]   {c};
  \draw (E) --             (B) node [midway]   {b};
  \draw (E) --             (C) node [near end] {a};
  \draw [-,line width=8pt,draw=graphicbackground]
        (D) to [bend right, looseness=1] (A);
  \draw (D) to [bend right, looseness=1] (A) node [near start] {b} node [near end]   {e};
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \matrix (network)
    [matrix of nodes,%
     nodes in empty cells,
     nodes={outer sep=0pt,circle,minimum size=4pt,draw},
     column sep={1cm,between origins},
     row sep={1cm,between origins}]
  {
                 &                &                 & \\
                 &                &                 & \\
    |[draw=none]| & |[xshift=1mm]| & |[xshift=-1mm]|   \\
  };
  \foreach \a in {1,...,4}{
    \draw (network-3-2) -- (network-2-\a);
    \draw (network-3-3) -- (network-2-\a);
    \draw [-stealth] ([yshift=5mm]network-1-\a.north) -- (network-1-\a);
    \foreach \b in {1,...,4}
      \draw (network-1-\a) -- (network-2-\b);
  }
  \draw [stealth-] ([yshift=-5mm]network-3-2.south) -- (network-3-2);
  \draw [stealth-] ([yshift=-5mm]network-3-3.south) -- (network-3-3);
\end{tikzpicture}
```

The following example is adapted from code written by Kjell Magne Fauske, which is based on the following paper: K. Bossley, M. Brown, and C. Harris, Neurofuzzy identification of an autonomous underwater vehicle, *International Journal of Systems Science*, 1999, 30, 901–913.

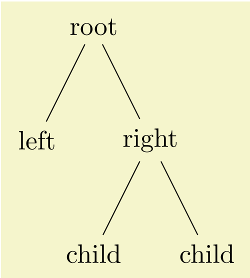


```
\begin{tikzpicture}[auto]
  \tikzstyle{decision} = [diamond, draw=blue, thick, fill=blue!20,
    text width=4.5em, text badly centered, inner sep=1pt]
  \tikzstyle{block} = [rectangle, draw=blue, thick, fill=blue!20,
    text width=5em, text centered, rounded corners, minimum height=4em]
  \tikzstyle{line} = [draw, thick, -latex',shorten >=2pt];
  \tikzstyle{cloud} = [draw=red, thick, ellipse,fill=red!20, minimum height=2em];
  \matrix [column sep=5mm,row sep=7mm]
  {
    % row 1
      \node [cloud] (expert)   {expert}; &
      \node [block] (init)     {initialize model}; &
      \node [cloud] (system)   {system}; \\
    % row 2
      & \node [block] (identify) {identify candidate model}; & \\
    % row 3
      \node [block] (update)   {update model};  &
      \node [block] (evaluate) {evaluate candidate models}; & \\
    % row 4
      & \node [decision] (decide) {is best candidate}; & \\
    % row 5
      & \node [block] (stop)      {stop}; & \\
  };
  \tikzstyle{every path}=[line]
  \path          (init)     -- (identify);
  \path          (identify) -- (evaluate);
  \path          (evaluate) -- (decide);
  \path          (update)   |- (identify);
  \path          (decide)   -| node [near start] {yes} (update);
  \path          (decide)   -- node [midway] {no} (stop);
  \path [dashed] (expert)   -- (init);
  \path [dashed] (system)   -- (init);
  \path [dashed] (system)   |- (evaluate);
\end{tikzpicture}
```

15 Making Trees Grow

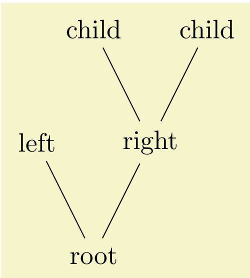
15.1 Introduction to the Child Operation

Trees are a common way of visualizing hierarchical structures. A simple tree looks like this:




```
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

Admittedly, in reality trees are more likely to grow *upward* and not downward as above. You can tell whether the author of a paper is a mathematician or a computer scientist by looking at the direction their trees grow. A computer scientist's trees will grow downward while a mathematician's tree will grow upward. Naturally, the *correct* way is the mathematician's way, which can be specify as follows:




```
\begin{tikzpicture}
  \node {root} [grow'=up]
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

In Ti*k*Z, trees are specified by adding *children* to a node on a path using the `child` operation:

`\path ... child[`⟨*options*⟩`]foreach`⟨*variables*⟩`in{`⟨*values*⟩`}{`⟨*child path*⟩`} ...;`

This operation should directly follow a completed `node` operation or another `child` operation, although it is permissible that the first `child` operation is preceded by options (we will come to that).

When a `node` operation like `node {X}` is followed by `child`, Ti*k*Z starts counting the number of child nodes that follow the original `node {X}`. For this, it scans the input and stores away each `child` and its arguments until it reaches a path operation that is not a `child`. Note that this will fix the character codes of all text inside the child arguments, which means, in essence, that you cannot use verbatim text inside the nodes inside a `child`. Sorry.

Once the children have been collected and counted, Ti*k*Z starts generating the child nodes. For each child of a parent node Ti*k*Z computes an appropriate position where the child is placed. For each child, the coordinate system is transformed so that the origin is at this position. Then the ⟨*child path*⟩ is drawn. Typically, the child path just consists of a `node` specification, which results in a node being drawn at the child's position. Finally, an edge is drawn from the first node in the ⟨*child path*⟩ to the parent node.

The optional `foreach` part (note that there is no backslash before `foreach`) allows you to specify multiple children in a single `child` command. The idea is the following: A `\foreach` statement is (internally) used to iterate over the list of ⟨*values*⟩. For each value in this list, a new `child` is added to the node. The syntax for ⟨*variables*⟩ and for ⟨*values*⟩ is the same as for the `\foreach` statement, see Section 34. For example, when you say

```
node {root} child [red] foreach \name in {1,2} {node {\name}}
```

the effect will be the same as if you had said

```
node {root} child[red] {node {1}} child[ref] {node {2}}
```

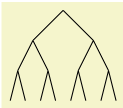
When you write

```
node {root} child[\pos] foreach \name/\pos in {1/left,2/right} {node[\pos] {\name}}
```

the effect will be the same as for

```
node {root} child[left] {node[left] {1}} child[right] {node[right] {2}}
```

You can nest things as in the following example:



```
\begin{tikzpicture}[level distance=4mm]
  \tikzstyle{level 1}=[sibling distance=8mm]
  \tikzstyle{level 2}=[sibling distance=4mm]
  \tikzstyle{level 3}=[sibling distance=2mm]
  \coordinate
    child foreach \x in {0,1}
      {child foreach \y in {0,1}
        {child foreach \z in {0,1}}};
\end{tikzpicture}
```

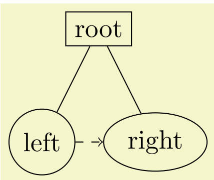
The details and options for this operation are described in the rest of this present section.

15.2 Child Paths and the Child Nodes

For each `child` of a root node, its ⟨*child path*⟩ is inserted at a specific location in the picture (the placement rules are discussed in Section 15.5). The first node in the ⟨*child path*⟩, if it exists, is special and called the *child node*. If there is no first node in the ⟨*child path*⟩, that is, if the ⟨*child path*⟩ is missing (including the curly braces) or if it does not start with `node` or with `coordinate`, then an empty child node of shape `coordinate` is automatically added.

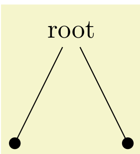
Consider the example `\node {x} child {node {y}} child;`. For the first child, the ⟨*child path*⟩ has the child node `node {y}`. For the second child, no child node is specified and, thus, it is just `coordinate`.

As for any normal node, you can give the child node a name, shift it around, or use options to influence how it is rendered.




```
\begin{tikzpicture}
  \node[rectangle,draw] {root}
    child {node[circle,draw] (left node) {left}}
    child {node[ellipse,draw] (right node) {right}};
  \draw[dashed,->] (left node) -- (right node);
\end{tikzpicture}
```

In many cases, the ⟨*child path*⟩ will just consist of a specification of a child node and, possibly, children of this child node. However, the node specification may be followed by arbitrary other material that will be added to the picture, transformed to the child's coordinate system. For your convenience, a move-to `(0,0)` operation is inserted automatically at the beginning of the path. Here is an example:




```
\begin{tikzpicture}
  \node {root}
    child {[fill] circle (2pt)}
    child {[fill] circle (2pt)};
\end{tikzpicture}
```

At the end of the ⟨*child path*⟩ you may add a special path operation called `edge from parent`. If this operation is not given by yourself somewhere on the path, it will be automatically added at the end. This option causes a connecting edge from the parent node to the child node to be added to the path. By giving options to this operation you can influence how the edge is rendered. Also, nodes following the `edge from parent` operation will be placed on this edge, see Section 15.6 for details.

To sum up:

1. The child path starts with a node specification. If it is not there, it is added automatically.

2. The child path ends with a `edge from parent` operation, possibly followed by nodes to be put on this edge. If the operation is not given at the end, it is added automatically.

15.3 Naming Child Nodes

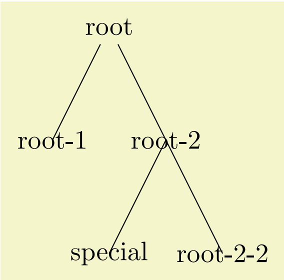
Child nodes can be named like any other node using either the `name` option or the special syntax in which the name of the node is placed in round parentheses between the `node` operation and the node's text.

If you do not assign a name to a child node, Ti*k*Z will automatically assign a name as follows: Assume that the name of the parent node is, say, `parent`. (If you did not assign a name to the parent, Ti*k*Z will do so itself, but that name will not be user-accessible.) The first child of `parent` will be named `parent-1`, the second child is named `parent-2`, and so on.

This naming convention works recursively. If the second child `parent-2` has children, then the first of these children will be called `parent-2-1` and the second `parent-2-2` and so on.

If you assign a name to a child node yourself, no name is generated automatically (the node does not have two names). However, "counting continues," which means that the third child of `parent` is called `parent-3` independently of whether you have assigned names to the first and/or second child of `parent`.

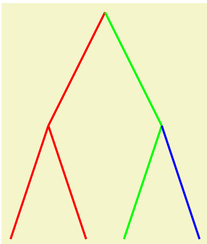
Here is an example:




```
\begin{tikzpicture}
  \node (root) {root}
    child
    child {
      child {coordinate (special)}
      child
    };
  \node at (root-1) {root-1};
  \node at (root-2) {root-2};
  \node at (special) {special};
  \node at (root-2-2) {root-2-2};
\end{tikzpicture}
```

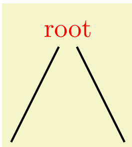
15.4 Specifying Options for Trees and Children

Each `child` may have its own ⟨*options*⟩, which apply to "the whole child," including all of its grandchildren. Here is an example:



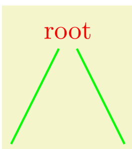
```
\begin{tikzpicture}[thick]
  \tikzstyle{level 2}=[sibling distance=10mm]
  \coordinate
    child[red]   {child child}
    child[green] {child child[blue]};
\end{tikzpicture}
```

The options of the root node have no effect on the children since the options of a node are always "local" to that node. Because of this, the edges in the following tree are black, not red.




```
\begin{tikzpicture}[thick]
  \node [red] {root}
    child
    child;
\end{tikzpicture}
```

This raises the problem of how to set options for *all* children. Naturally, you could always set options for the whole path as in `\path [red] node {root} child child;` but this is bothersome in some situations. Instead, it is easier to give the options *before the first child* as follows:




```
\begin{tikzpicture}[thick]
  \node [red] {root}
    [green] % option applies to all children
    child
    child;
\end{tikzpicture}
```

Here is the set of rules:

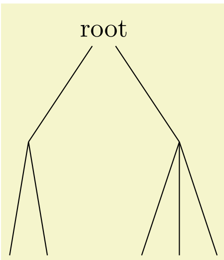
1. Options for the whole tree are given before the root node.
2. Options for the root node are given directly to the `node` operation of the root.
3. Options for all children can be given between the root node and the first child.
4. Options applying to a specific child path are given as options to the `child` operation.

5. Options applying to the node of a child, but not to the whole child path, are given as options to the node command inside the ⟨*child path*⟩.

```
\begin{tikzpicture}
  \path
    [...]             % Options apply to the whole tree
    node[...] {root}  % Options apply to the root node only
      [...]           % Options apply to all children
      child[...]      % Options apply to this child and all its children
      {
        node[...] {}  % Options apply to the child node only
        ...
      }
      child[...]      % Options apply to this child and all its children
    ;
\end{tikzpicture}
```

There are additional styles that influence how children are rendered:

- `style=every child` This style is used at the beginning of each child, as if you had given the options to the `child` operation.
- `style=every child node` This style is used at the beginning of each child node in addition to the `every node` style.
- `style=level` ⟨*number*⟩ This style is used at the beginning of each set of children, where ⟨*number*⟩ is the current level in the current tree. For example, when you say `\node {x} child child;`, then the style `level 1` is used before the first `child`. If this first `child` has children itself, then `level 2` would be used for them.




```
\begin{tikzpicture}
  \tikzstyle{level 1}=[sibling distance=20mm]
  \tikzstyle{level 2}=[sibling distance=5mm]
  \node {root}
    child { child child }
    child { child child child };
\end{tikzpicture}
```

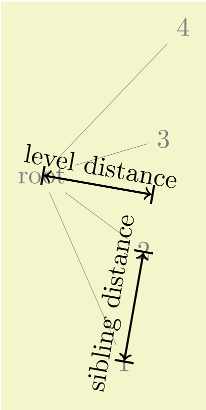
15.5 Placing Child Nodes

Perhaps the most difficult part in drawing a tree is the correct layout of the children. Typically, the children have different sizes and it is not easy to arrange them in such a manner that not too much space is wasted, the children do not overlap, and they are either evenly spaced or their centers are evenly distributed. Calculating good positions is especially difficult since a good position for the first child may depend on the size of the last child.

In Ti*k*Z, a comparatively simple approach is taken to placing the children. In order to compute a child's position, all that is taken into account is the number of the current child in the list of children and the number of children in this list. Thus, if a node has five children, then there is a fixed position for the first child, a position for the second child, and so on. These positions *do not depend on the size of the children* and, hence, children can easily overlap. However, since you can use options to shift individual children a bit, this is not as great a problem as it may seem.

Although the placement of the children only depends on their number in the list of children and the total number of children, everything else about the placement is highly configurable. You can change the distance between children (appropriately called the `sibling distance`) and the distance between levels of the tree. These distances may change from level to level. The direction in which the tree grows can be changed globally and for parts of the tree. You can even specify your own "growth function" to arrange children on a circle or along special lines or curves.

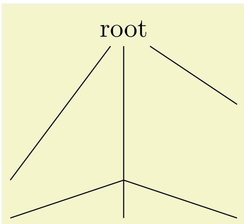
The default growth function works as follows: Assume that we are given a node and five children. These children will be placed on a line with their centers (or, more generally, with their anchors) spaced apart by the current `sibling distance`. The line is orthogonal to the current *direction of growth*, which is set with the `grow` and `grow'` option (the latter option reverses the ordering of the children). The distance from the line to the parent node is given by the `level distance`.



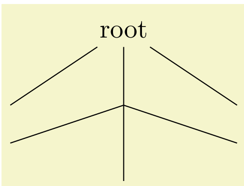

```
\begin{tikzpicture}
  \path [help lines]
    node (root) {root}
    [grow=-10]
    child {node {1}}
    child {node {2}}
    child {node {3}}
    child {node {4}};

  \draw[|<->|,thick] (root-1.center)
    -- node[above,sloped] {sibling distance} (root-2.center);

  \draw[|<->|,thick] (root.center)
    -- node[above,sloped] {level distance} +(-10:\tikzleveldistance);
\end{tikzpicture}
```

Here is a detailed description of the options:

- `level distance`=⟨*distance*⟩ This option allows you to change the distance between different levels of the tree, more precisely, between the parent and the line on which its children are arranged. When given to a single child, this will set the distance for this child only.

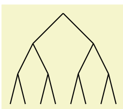



```
\begin{tikzpicture}
  \node {root}
    [level distance=20mm]
    child
    child {
      [level distance=5mm]
      child
      child
      child
    }
    child[level distance=10mm];
\end{tikzpicture}
```

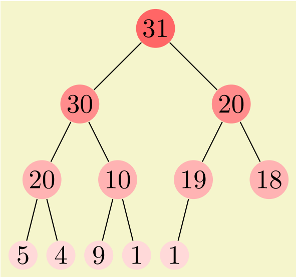



```
\begin{tikzpicture}
  \tikzstyle{level 1}=[level distance=10mm]
  \tikzstyle{level 2}=[level distance=5mm]
  \node {root}
    child
    child {
      child
      child[level distance=10mm]
      child
    }
    child;
\end{tikzpicture}
```

- `sibling distance`=⟨*distance*⟩ This option specifies the distance between the anchors of the children of a parent node.



```
\begin{tikzpicture}[level distance=4mm]
  \tikzstyle{level 1}=[sibling distance=8mm]
  \tikzstyle{level 2}=[sibling distance=4mm]
  \tikzstyle{level 3}=[sibling distance=2mm]
  \coordinate
     child {
       child {child child}
       child {child child}
     }
     child {
       child {child child}
       child {child child}
     };
\end{tikzpicture}
```




```
\begin{tikzpicture}[level distance=10mm]
  \tikzstyle{every node}=[fill=red!60,circle,inner sep=1pt]
  \tikzstyle{level 1}=[sibling distance=20mm,
    set style={{every node}+=[fill=red!45]}]
  \tikzstyle{level 2}=[sibling distance=10mm,
    set style={{every node}+=[fill=red!30]}]
  \tikzstyle{level 3}=[sibling distance=5mm,
    set style={{every node}+=[fill=red!15]}]
  \node {31}
     child {node {30}
       child {node {20}
         child {node {5}}
         child {node {4}}
       }
       child {node {10}
         child {node {9}}
         child {node {1}}
       }
     }
     child {node {20}
       child {node {19}
         child {node {1}}
         child[fill=none] {edge from parent[draw=none]}
       }
       child {node {18}}
     };
\end{tikzpicture}
```

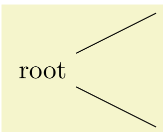
- `grow`=⟨*direction*⟩ This option is used to define the ⟨*direction*⟩ in which the tree will grow. The ⟨*direction*⟩ can either be an angle in degrees or one of the following special text strings: `down`, `up`, `left`, `right`, `north`, `south`, `east`, `west`, `north east`, `north west`, `south east`, and `south west`. All of these have "their obvious meaning," so, say, `south west` is the same as the angle $-135°$.

  As a side effect, this option installs the default growth function.

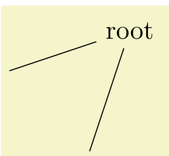
  In addition to setting the direction, this option also has a seemingly strange effect: It sets the sibling distance for the current level to `0pt`, but leaves the sibling distance for later levels unchanged.

  This somewhat strange behaviour has a highly desirable effect: If you give this option before the list of children of a node starts, the "current level" is still the parent level. Each child will be on a later level and, hence, the sibling distance will be as specified originally. This will cause the children to be neatly aligned in a line orthogonal to the given ⟨*direction*⟩. However, if you give this option locally to a single child, then "current level" will be the same as the child's level. The zero sibling distance will then cause the child to be placed exactly at a point at distance `level distance` in the direction ⟨*direction*⟩. However, the children of the child will be placed "normally" on a line orthogonal to the ⟨*direction*⟩.

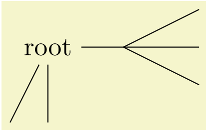
  These placement effects are best demonstrated by some examples:



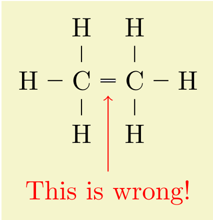

```
\tikz \node {root} [grow=right] child child;
```



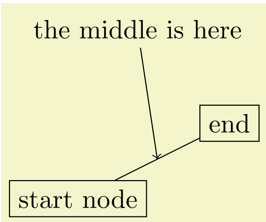

```
\tikz \node {root} [grow=south west] child child;
```




```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
  \node {root}
    [grow=down]
    child
    child
    child[grow=right] {
      child child child
    };
\end{tikzpicture}
```



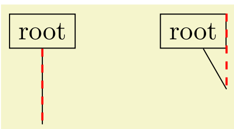

```
\begin{tikzpicture}[level distance=2em]
  \node {C}
    child[grow=up]    {node {H}}
    child[grow=left]  {node {H}}
    child[grow=down]  {node {H}}
    child[grow=right] {node {C}
        child[grow=up]    {node {H}}
        child[grow=right] {node {H}}
        child[grow=down]  {node {H}}
      edge from parent[double]
        coordinate (wrong)
    };
  \draw[<-,red] ([yshift=-2mm]wrong) -- +(0,-1)
    node[below]{This is wrong!};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \node[rectangle,draw] (a) at (0,0) {start node};
  \node[rectangle,draw] (b) at (2,1) {end};

  \draw (a) -- (b)
    node[coordinate,midway] {}
      child[grow=100,<-] {node[above] {the middle is here}};
\end{tikzpicture}
```

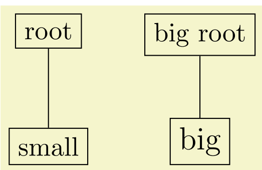
- `grow'`=⟨*direction*⟩ This option has the same effect as `grow`, only the children are arranged in the opposite order.
- `growth parent anchor`=⟨*anchor*⟩ This option allows you to specify which anchor of the parent node is to be used for computing the children's position. For example, when there is only one child and the `level distance` is `2cm`, then the child node will be placed two centimeters below the ⟨*anchor*⟩ of the parent node. "Beinng placed" means that the child node's anchor (which is the anchor specified using the `anchor=` option in the `node` command of the child) is two centimeters below the parent node's ⟨*anchor*⟩. The default value of ⟨*anchor*⟩ is `center`.

  In the following example, the two red lines both have length `1cm`.




```
\begin{tikzpicture}[level distance=1cm]
  \node [rectangle,draw] (a) at (0,0) {root}
  [growth parent anchor=south] child;

  \node [rectangle,draw] (b) at (2,0) {root}
  [growth parent anchor=north east] child;

  \draw [red,thick,dashed] (a.south) -- (a-1);
  \draw [red,thick,dashed] (b.north east) -- (b-1);
\end{tikzpicture}
```

  In the next example, the top and bottom nodes are aligned at the top and the bottom.




```
\begin{tikzpicture}[level distance=2cm,growth parent anchor=north]
  \tikzstyle{every node}=[anchor=north,rectangle,draw]
  \tikzstyle{every child node}=[anchor=south]

  \node at (0,0) {root} child {node {small}};

  \node at (2,0) {big root} child {node {\large big}};
\end{tikzpicture}
```

- growth function=⟨*macro name*⟩ This rather low-level option allows you to set a new growth function. The ⟨*macro name*⟩ must be the name of a macro without parameters. This macro will be called for each child of a node.

  The effect of executing the macro should be the following: It should transform the coordinate system in such a way that the origin becomes the place where the current child should be anchored. When the macro is called, the current coordinate system will be setup such that the anchor of the parent node is in the origin. Thus, in each call, the ⟨*macro name*⟩ must essentially do a shift to the child's origin. When the macro is called, the TEX counter `\tikznumberofchildren` will be set to the total number of children of the parent node and the counter `\tikznumberofcurrentchild` will be set to the number of the current child.

  The macro may, in addition to shifting the coordinate system, also transform the coordinate system further. For example, it could be rotated or scaled.

  Additional growth functions are defined in the library, see Section 33.

15.6 Edges From the Parent Node

Every child node is connected to its parent node via a special kind of edge called the `edge from parent`. This edge is added to the ⟨*child path*⟩ when the following path operation is encountered:

`\path ... edge from parent[`⟨*options*⟩`] ...;`

This path operation can only be used inside ⟨*child paths*⟩ and should be given at the end, possibly followed by node specifications (we will come to that). If a ⟨*child path*⟩ does not contain this operation, it will be added at the end of the ⟨*child path*⟩ automatically.

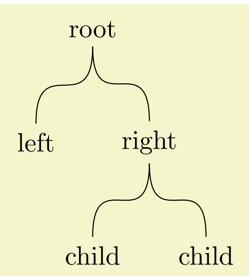
This operation has several effects. The most important is that it inserts the current "edge from parent path" into the child path. The edge from parent path can be set using the following option:

- edge from parent path=⟨*path*⟩ This options allows you to set the edge from parent path to a new path. The default for this path is the following:

```
(\tikzparentnode\tikzparentanchor) -- (\tikzchildnode\tikzchildanchor)
```

  The `\tikzparentnode` is a macro that will expand to the name of the parent node. This works even when you have not assigned a name to the parent node, in this case an internal name is automatically generated. The `\tikzchildnode` is a macro that expands to the name of the child node. The two `...anchor` macros are empty by default. So, what is essentially inserted is just the path segment `(\tikzparentnode) -- (\tikzchildnode)`; which is exactly an edge from the parent to the child.

  You can modify this edge from parent path to achieve all sorts of effects. For example, we could replace the straight line by a curve as follows:

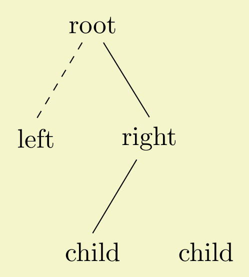



```
\begin{tikzpicture}[edge from parent path=
  {(\tikzparentnode.south) .. controls +(0,-1) and +(0,1)
                           .. (\tikzchildnode.north)}]
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

  Further useful edge from parent paths are defined in the tree library, see Section 33.

  As said before, the anchors in the default edge from parent path are empty. However, you can set them using the following options:

  – child anchor=⟨*anchor*⟩ Specifies the anchor where the edge from parent meets the child node by setting the macro `\tikzchildanchor` to `.`⟨*anchor*⟩.
    If you specify `border` as the ⟨*anchor*⟩, then the macro `\tikzchildanchor` is set to the empty string. The effect of this is that the edge from the parent will meet the child on the border at an automatically calculated position.

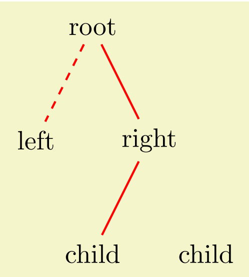



```
\begin{tikzpicture}
  \node {root}
    [child anchor=north]
    child {node {left} edge from parent[dashed]}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

– `parent anchor`=⟨*anchor*⟩ This option works the same way as the `child anchor`, only for the parent.

Besides inserting the edge from parent path, the `edge from parent` operation has another effect: The ⟨*options*⟩ are inserted directly before the edge from parent path and the following style is also installed prior to inserting the path:

- `style=edge from parent` This style is inserted right before the edge from parent path and before the ⟨*options*⟩ are inserted. By default, it just draws the edge from parent, but you can use it to make the edge look different.




```
\begin{tikzpicture}
  \tikzstyle{edge from parent}=[draw,red,thick]
  \node {root}
    child {node {left} edge from parent[dashed]}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

Note: The ⟨*options*⟩ inserted before the edge from parent path is added *apply to the whole child path*. Thus, it is not possible to, say, draw a circle in red as part of the child path and then have an edge to parent in blue. However, as always, the child node is a node and can be drawn in a totally different way.

Finally, the `edge from parent` operation has one more effect: It causes all nodes *following* the operation to be placed on the edge. This is the same effect as if you had added the `pos` option to all these nodes, see also Section 13.7.

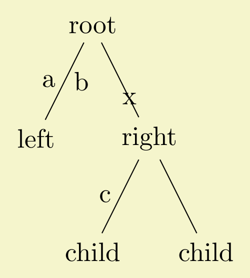
As an example, consider the following code:

```
\node (root) {} child {node (child) {} edge to parent node {label}};
```

The `edge to parent` operation and the following `node` operation will, together, have the same effect as if we had said:

```
(root) -- (child) node [pos=0.5] {label}
```

Here is a more complicated example:




```
\begin{tikzpicture}
  \node {root}
    child {
      node {left}
      edge from parent
        node[left] {a}
        node[right] {b}
    }
    child {
      node {right}
        child {
          node {child}
          edge from parent
            node[left] {c}
        }
        child {node {child}}
      edge from parent
        node[near end] {x}
    };
\end{tikzpicture}
```

16 Plots of Functions

16.1 When Should One Use TikZ for Generating Plots?

There exist many powerful programs that produce plots, examples are GNUPLOT or MATHEMATICA. These programs can produce two different kinds of output: First, they can output a complete plot picture in a certain format (like PDF) that includes all low-level commands necessary for drawing the complete plot (including axes and labels). Second, they can usually also produce "just plain data" in the form of a long list of coordinates. Most of the powerful programs consider it a to be "a bit boring" to just output tabled data and very much prefer to produce fancy pictures. Nevertheless, when coaxed, they can also provide the plain data.

Note that is often not necessary to use TikZ for plots. Programs like GNUPLOT can produce very sophisticated plots and it is usually much easier to simply include these plots as a finished PDF or PostScript graphics.

However, there are a number of reasons why you may wish to invest time and energy into mastering the PGF commands for creating plots:

- Virtually all plots produced by "external programs" use different fonts from the one used in your document.
- Even worse, formulas will look totally different, if they can be rendered at all.
- Line width will usually be too large or too small.
- Scaling effects upon inclusion can create a mismatch between sizes in the plot and sizes in the text.
- The automatic grid generated by most programs is mostly distracting.
- The automatic ticks generated by most programs are cryptic numerics. (Try adding a tick reading "π" at the right point.)
- Most programs make it very easy to create "chart junk" in a most convenient fashion. All show, no content.
- Arrows and plot marks will almost never match the arrows used in the rest of the document.

The above list is not exhaustive, unfortunately.

16.2 The Plot Path Operation

The `plot` path operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

The syntax of the `plot` comes in different versions.

`\path ... --plot`⟨*further arguments*⟩ `...;`

This operation plots the curve through the coordinates specified in the ⟨*further arguments*⟩. The current (sub)path is simply continued, that is, a line-to operation to the first point of the curve is implicitly added. The details of the ⟨*further arguments*⟩ will be explained in a moment.

`\path ... plot`⟨*further arguments*⟩ `...;`

This operation plots the curve through the coordinates specified in the ⟨*further arguments*⟩ by first "moving" to the first coordinate of the curve.

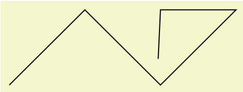
The ⟨*further arguments*⟩ are used in three different ways to specifying the coordinates of the points to be plotted:

1. `--plot[`⟨*local options*⟩`]coordinates{`⟨*coordinate 1*⟩⟨*coordinate 2*⟩...⟨*coordinate n*⟩`}`
2. `--plot[`⟨*local options*⟩`]file{`⟨*filename*⟩`}`
3. `--plot[`⟨*local options*⟩`]`⟨*coordinate expression*⟩
4. `--plot[`⟨*local options*⟩`]function{`⟨*gnuplot formula*⟩`}`

These different ways are explained in the following.

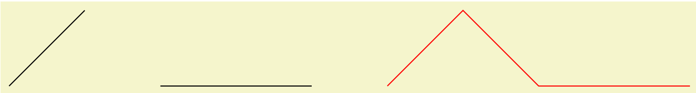
16.3 Plotting Points Given Inline

In the first two cases, the points are given directly in the TEX-file as in the following example:



```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm)};
```

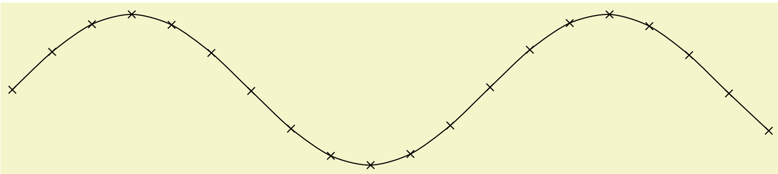
Here is an example showing the difference between `plot` and `--plot`:



```
\begin{tikzpicture}
  \draw (0,0) -- (1,1) plot coordinates {(2,0)  (4,0)};
  \draw[color=red,xshift=5cm]
        (0,0) -- (1,1) -- plot coordinates {(2,0)  (4,0)};
\end{tikzpicture}
```

16.4 Plotting Points Read From an External File

The second way of specifying points is to put them in an external file named ⟨*filename*⟩. Currently, the only file format that TikZ allows is the following: Each line of the ⟨*filename*⟩ should contain one line starting with two numbers, separated by a space. Anything following the two numbers on the line is ignored. Also, lines starting with a `%` or a `#` are ignored as well as empty lines. (This is exactly the format that GNUPLOT produces when you say `set terminal table`.) If necessary, more formats will be supported in the future, but it is usually easy to produce a file containing data in this form.



```
\tikz \draw plot[mark=x,smooth] file {plots/pgfmanual-sine.table};
```

The file `plots/pgfmanual-sine.table` reads:

```
#Curve 0, 20 points
#x y type
0.00000 0.00000  i
0.52632 0.50235  i
1.05263 0.86873  i
1.57895 0.99997  i
...
9.47368 -0.04889  i
10.00000 -0.54402  i
```

It was produced from the following source, using `gnuplot`:

```
set terminal table
set output "../plots/pgfmanual-sine.table"
set format "%.5f"
set samples 20
plot [x=0:10] sin(x)
```

The ⟨*local options*⟩ of the `plot` operation are local to each plot and do not affect other plots "on the same path." For example, `plot[yshift=1cm]` will locally shift the plot 1cm upward. Remember, however, that most options can only be applied to paths as a whole. For example, `plot[red]` does not have the effect of making the plot red. After all, you are trying to "locally" make part of the path red, which is not possible.

16.5 Plotting a Function

When you plot a function, the coordinates of the plot data can be computed by evaluating a mathematical expression. Since PGF comes with a mathematical engine, you can specify this expression and then have TikZ produce the desired coordinates for you, automatically.

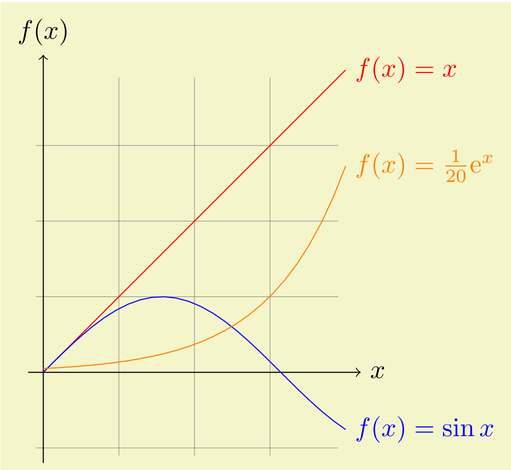
Since this case is quite common when plotting a function, the syntax is easy: Following the `plot` command and its local options, you directly provide a ⟨*coordinate expression*⟩. It looks like a normal coordinate, but inside you may use a special macro, which is `\x` by default, but this can be changed using the `variable` option. The ⟨*coordinate expression*⟩ is then evaluated for different values for `\x` and the resulting coordinates are plotted.

Note that you will often have to put the x- or y-coordinate inside braces, namely whenever you use an expression involving a paranthesis.

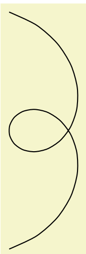
The following options influence how the ⟨*coordinate expression*⟩ is evaluated:

- `variable=`⟨*macro*⟩ sets the macro whose value is set to the different values when ⟨*coordinate expression*⟩ is evaluated.
- `samples=`⟨*number*⟩ sets the number of samples used in the plot. The default is 25.
- `domain=`⟨*start*⟩`:`⟨*end*⟩ sets the domain between which the samples are taken. The default is `-5:5`.
- `samples at=`⟨*sample list*⟩ This option specifies a list of positions for which the variable should be evaluated. For instance, you can say `samples at={1,2,8,9,10}` to have the variable evaluated exactly for values 1, 2, 8, 9, and 10. You can use the `\foreach` syntax, so you can use `...` inside the ⟨*sample list*⟩.

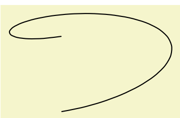
  When this option is used, the `samples` and `domain` option are overruled. The other ways round, setting either `samples` or `domain` will overrule this option.




```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

  \draw[color=red]    plot (\x,\x)             node[right] {$f(x) =x$};
  \draw[color=blue]   plot (\x,{sin(\x r)})    node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot (\x,{0.05*exp(\x)}) node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
\end{tikzpicture}
```



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth,variable=\t]
  plot ({\t*sin(\t r)},{\t*cos(\t r)});
```



```
\tikz \draw[domain=0:360,smooth,variable=\t]
  plot ({sin(\t)},\t/360,{cos(\t)});
```

16.6 Plotting a Function Using Gnuplot

Often, you will want to plot points that are given via a function like $f(x) = x \sin x$. Unfortunately, TEX does not really have enough computational power to generate the points on such a function efficiently (it is a text processing program, after all). However, if you allow it, TEX can try to call external programs that can easily produce the necessary points. Currently, Ti*k*Z knows how to call GNUPLOT.

When Ti*k*Z encounters your operation `plot[id=`⟨*id*⟩`] function{x*sin(x)}` for the first time, it will create a file called ⟨*prefix*⟩⟨*id*⟩`.gnuplot`, where ⟨*prefix*⟩ is `\jobname.` by default, that is, the name of you main `.tex` file. If no ⟨*id*⟩ is given, it will be empty, which is alright, but it is better when each plot has a unique ⟨*id*⟩ for reasons explained in a moment. Next, Ti*k*Z writes some initialization code into this file followed by `plot x*sin(x)`. The initialization code sets up things such that the `plot` operation will write the coordinates into another file called ⟨*prefix*⟩⟨*id*⟩`.table`. Finally, this table file is read as if you had said `plot file{`⟨*prefix*⟩⟨*id*⟩`.table}`.

For the plotting mechanism to work, two conditions must be met:

1. You must have allowed TEX to call external programs. This is often switched off by default since this is a security risk (you might, without knowing, run a TEX file that calls all sorts of "bad" commands). To enable this "calling external programs" a command line option must be given to the TEX program. Usually, it is called something like `shell-escape` or `enable-write18`. For example, for my `pdflatex` the option `--shell-escape` can be given.

2. You must have installed the `gnuplot` program and TEX must find it when compiling your file.

Unfortunately, these conditions will not always be met. Especially if you pass some source to a coauthor and the coauthor does not have GNUPLOT installed, he or she will have trouble compiling your files.

For this reason, Ti*k*Z behaves differently when you compile your graphic for the second time: If upon reaching `plot[id=`⟨*id*⟩`] function{...}` the file ⟨*prefix*⟩⟨*id*⟩`.table` already exists *and* if the ⟨*prefix*⟩⟨*id*⟩`.gnuplot` file contains what Ti*k*Z thinks that it "should" contain, the `.table` file is immediately read without trying to call a `gnuplot` program. This approach has the following advantages:

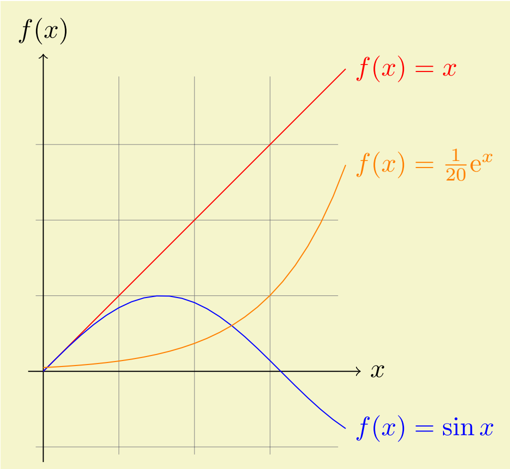
1. If you pass a bundle of your `.tex` file and all `.gnuplot` and `.table` files to someone else, that person can TEX the `.tex` file without having to have `gnuplot` installed.

2. If the `\write18` feature is switched off for security reasons (a good idea), then, upon the first compilation of the `.tex` file, the `.gnuplot` will still be generated, but not the `.table` file. You can then simply call `gnuplot` "by hand" for each `.gnuplot` file, which will produce all necessary `.table` files.

3. If you change the function that you wish to plot or its domain, Ti*k*Z will automatically try to regenerate the `.table` file.

4. If, out of laziness, you do not provide an `id`, the same `.gnuplot` will be used for different plots, but this is not a problem since the `.table` will automatically be regenerated for each plot on-the-fly. *Note: If you intend to share your files with someone else, always use an id, so that the file can by typeset without having* GNUPLOT *installed.* Also, having unique ids for each plot will improve compilation speed since no external programs need to be called, unless it is really necessary.

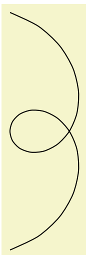
When you use `plot function{`⟨*gnuplot formula*⟩`}`, the ⟨*gnuplot formula*⟩ must be given in the `gnuplot` syntax, whose details are beyond the scope of this manual. Here is the ultra-condensed essence: Use `x` as the variable and use the C-syntax for normal plots, use `t` as the variable for parametric plots. Here are some examples:




```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

  \draw[color=red]    plot[id=x]   function{x}           node[right] {$f(x) =x$};
  \draw[color=blue]   plot[id=sin] function{sin(x)}      node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot[id=exp] function{0.05*exp(x)} node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
\end{tikzpicture}
```

The following options influence the plot:

- `samples`=⟨*number*⟩ sets the number of samples used in the plot. The default is 25.
- `domain`=⟨*start*⟩`:`⟨*end*⟩ sets the domain between which the samples are taken. The default is `-5:5`.
- `parametric`=⟨*true or false*⟩ sets whether the plot is a parametric plot. If true, then `t` must be used instead of `x` as the parameter and two comma-separated functions must be given in the ⟨*gnuplot formula*⟩. An example is the following:



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth]
  plot[parametric,id=parametric-example] function{t*sin(t),t*cos(t)};
```

- `id`=⟨*id*⟩ sets the identifier of the current plot. This should be a unique identifier for each plot (though things will also work if it is not, but not as well, see the explanations above). The ⟨*id*⟩ will be part of a filename, so it should not contain anything fancy like `*` or `$`.
- `prefix`=⟨*prefix*⟩ is put before each plot file name. The default is `\jobname.`, but if you have many plots, it might be better to use, say `plots/` and have all plots placed in a directory. You have to create the directory yourself.
- `raw gnuplot` causes the ⟨*gnuplot formula*⟩ to be passed on to GNUPLOT without setting up the samples or the `plot` operation. Thus, you could write

  ```
  plot[raw gnuplot,id=raw-example] function{set samples 25; plot sin(x)}
  ```

  This can be useful for complicated things that need to be passed to GNUPLOT. However, for really complicated situations you should create a special external generating GNUPLOT file and use the `file`-syntax to include the table "by hand."

The following styles influence the plot:

- `style=every plot` This style is installed in each plot, that is, as if you always said

```
plot[style=every plot,...]
```

This is most useful for globally setting a prefix for all plots by saying:

```
\tikzstyle{every plot}=[prefix=plots/]
```

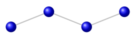
16.7 Placing Marks on the Plot

As we saw already, it is possible to add *marks* to a plot using the `mark` option. When this option is used, a copy of the plot mark is placed on each point of the plot. Note that the marks are placed *after* the whole path has been drawn/filled/shaded. In this respect, they are handled like text nodes.

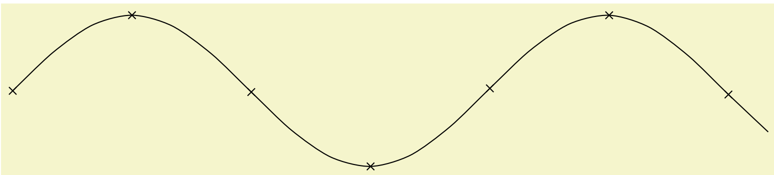
In detail, the following options govern how marks are drawn:

- `mark=`⟨*mark mnemonic*⟩ Sets the mark to a mnemonic that has previously been defined using the `\pgfdeclareplotmark`. By default, `*`, `+`, and `x` are available, which draw a filled circle, a plus, and a cross as marks. Many more marks become available when the library `pgflibraryplotmarks` is loaded. Section 28.3 lists the available plot marks.

  One plot mark is special: the `ball` plot mark is available only it Ti*k*Z. The `ball color` determines the balls's color. Do not use this option with a large number of marks since it will take very long to render in PostScript.

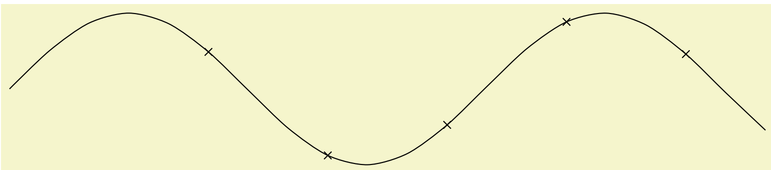
Option	Effect
`mark=ball`	

- `mark repeat=`⟨r⟩ This option tells Ti*k*Z that only every rth mark should be drawn.



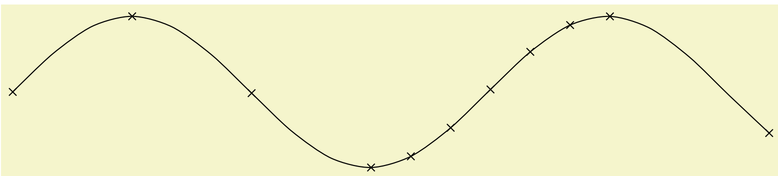
```
\tikz \draw plot[mark=x,mark repeat=3,smooth] file {plots/pgfmanual-sine.table};
```

- `mark phase=`⟨p⟩ This option tells Ti*k*Z that the first mark to be draw should be the pth, followed by the $(p+r)$th, then the $(p+2r)$th, and so on.



```
\tikz \draw plot[mark=x,mark repeat=3,mark phase=6,smooth] file {plots/pgfmanual-sine.table};
```

- `mark indices=`⟨*list*⟩ This option allows you to specify explicitly the indices at which a mark should be placed. Counting starts with 1. You can use the `\foreach` syntax, that is, `...` can be used.

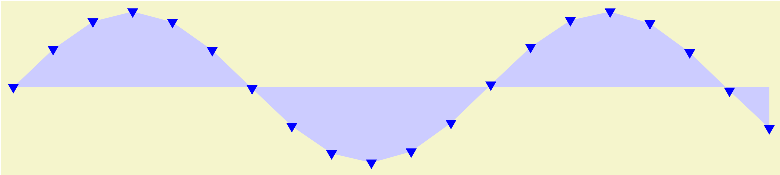


```
\tikz \draw plot[mark=x,mark indices={1,4,...,10,11,12,...,16,20},smooth]
  file {plots/pgfmanual-sine.table};
```

- mark size=⟨*dimension*⟩ Sets the size of the plot marks. For circular plot marks, ⟨*dimension*⟩ is the radius, for other plot marks ⟨*dimension*⟩ should be about half the width and height.

  This option is not really necessary, since you achieve the same effect by specifying `scale=`⟨*factor*⟩ as a local option, where ⟨*factor*⟩ is the quotient of the desired size and the default size. However, using `mark size` is a bit faster and more natural.

- mark options=⟨*options*⟩ These options are applied to marks when they are drawn. For example, you can scale (or otherwise transform) the plot mark or set its color.



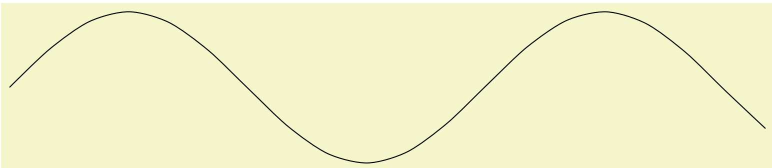
```
\tikz \fill[fill=blue!20]
  plot[mark=triangle*,mark options={color=blue,rotate=180}]
    file{plots/pgfmanual-sine.table} |- (0,0);
```

16.8 Smooth Plots, Sharp Plots, and Comb Plots

There are different things the `plot` operation can do with the points it reads from a file or from the inlined list of points. By default, it will connect these points by straight lines. However, you can also use options to change the behavior of `plot`.

- sharp plot This is the default and causes the points to be connected by straight lines. This option is included only so that you can "switch back" if you "globally" install, say, `smooth`.

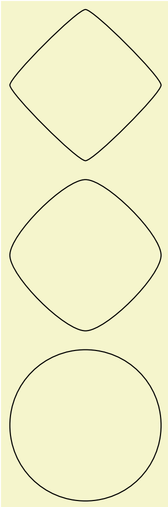
- smooth This option causes the points on the path to be connected using a smooth curve:



```
\tikz\draw plot[smooth] file{plots/pgfmanual-sine.table};
```

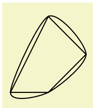
  Note that the smoothing algorithm is not very intelligent. You will get the best results if the bending angles are small, that is, less than about 30° and, even more importantly, if the distances between points are about the same all over the plotting path.

- tension=⟨*value*⟩ This option influences how "tight" the smoothing is. A lower value will result in sharper corners, a higher value in more "round" curves. A value of 1 results in a circle if four points at quarter-positions on a circle are given. The default is 0.55. The "correct" value depends on the details of plot.



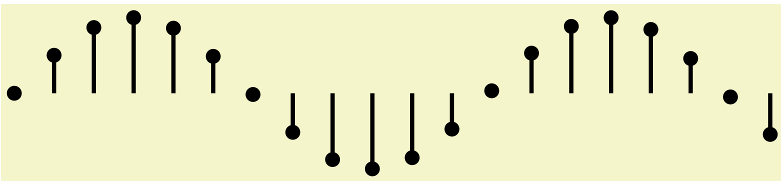
```
\begin{tikzpicture}[smooth cycle]
  \draw                  plot[tension=0.2]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[yshift=-2.25cm] plot[tension=0.5]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[yshift=-4.5cm]  plot[tension=1]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
\end{tikzpicture}
```

- `smooth cycle` This option causes the points on the path to be connected using a closed smooth curve.

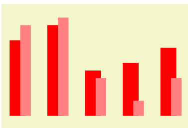


```
\tikz[scale=0.5]
  \draw plot[smooth cycle] coordinates{(0,0) (1,0) (2,1) (1,2)}
      plot               coordinates{(0,0) (1,0) (2,1) (1,2)} -- cycle;
```

- `ycomb` This option causes the `plot` operation to interpret the plotting points differently. Instead of connecting them, for each point of the plot a straight line is added to the path from the x-axis to the point, resulting in a sort of "comb" or "bar diagram."

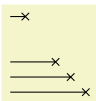


```
\tikz\draw[ultra thick] plot[ycomb,thin,mark=*] file{plots/pgfmanual-sine.table};
```



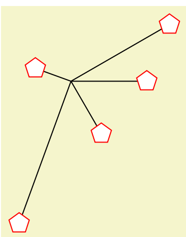
```
\begin{tikzpicture}[ycomb]
  \draw[color=red,line width=6pt]
    plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
  \draw[color=red!50,line width=4pt,xshift=3pt]
    plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

- `xcomb` This option works like `ycomb` except that the bars are horizontal.



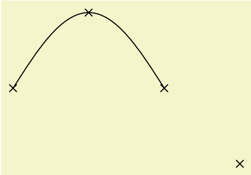
```
\tikz \draw plot[xcomb,mark=x] coordinates{(1,0) (0.8,0.2) (0.6,0.4) (0.2,1)};
```

- `polar comb` This option causes a line from the origin to the point to be added to the path for each plot point.



```
\tikz \draw plot[polar comb,
     mark=pentagon*,mark options={fill=white,draw=red},mark size=4pt]
   coordinates {(0:1cm) (30:1.5cm) (160:.5cm) (250:2cm) (-60:.8cm)};
```

- `only marks` This option causes only marks to be shown; no path segments are added to the actual path. This can be useful for quickly adding some marks to a path.



```
\tikz \draw (0,0) sin (1,1) cos (2,0)
  plot[only marks,mark=x] coordinates{(0,0) (1,1) (2,0) (3,-1)};
```

17 Transformations

PGF has a powerful transformation mechanism that is similar to the transformation capabilities of METAFONT. The present section explains how you can access it in Ti*k*Z.

17.1 The Different Coordinate Systems

It is a long process from a coordinate like, say, $(1,2)$ or $(1\text{cm}, 5\,mathrmpt)$, to the position a point is finally placed on the display or paper. In order to find out where the point should go, it is constantly "transformed," which means that it is mostly shifted around and possibly rotated, slanted, scaled, and otherwise mutilated.

In detail, (at least) the following transformations are applied to a coordinate like $(1,2)$ before a point on the screen is chosen:

1. PGF interprets a coordinate like $(1,2)$ in its xy-coordinate system as "add the current x-vector once and the current y-vector twice to obtain the new point."
2. PGF applies its coordinate transformation matrix to the resulting coordinate. This yields the final position of the point inside the picture.
3. The backend driver (like `dvips` or `pdftex`) adds transformation commands such the coordinate is shifted to the correct position in TeX's page coordinate system.
4. PDF (or PostScript) apply the canvas transformation matrix to the point, which can once more change the position on the page.
5. The viewer application or the printer applies the device transformation matrix to transform the coordinate to its final pixel coordinate on the screen or paper.

In reality, the process is even more involved, but the above should give the idea: A point is constantly transformed by changes of the coordinate system.

In Ti*k*Z, you only have access to the first two coordinate systems: The xy-coordinate system and the coordinate transformation matrix (these will be explained later). PGF also allows you to change the canvas transformation matrix, but you have to use commands of the core layer directly to do so and you "better know what you are doing" when you do this. The moment you start modifying the canvas matrix, PGF immediately looses track of all coordinates and shapes, anchors, and bounding box computations will no longer work.

17.2 The XY- and XYZ-Coordinate Systems

The first and easiest coordinate systems are PGF's xy- and xyz-coordinate systems. The idea is very simple: Whenever you specify a coordinate like `(2,3)` this means $2v_x + 3v_y$, where v_x is the current *x-vector* and v_y is the current *y-vector*. Similarly, the coordinate `(1,2,3)` means $v_x + 2v_y + 3v_z$.

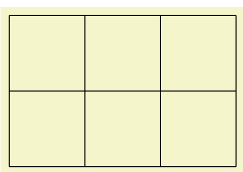
Unlike other packages, PGF does not insist that v_x actually has a y-component of 0, that is, that it is a horizontal vector. Instead, the x-vector can point anywhere you want. Naturally, *normally* you will want the x-vector to point horizontally.

One undesirable effect of this flexibility is that it is not possible to provide mixed coordinates as in $(1, 2\text{pt})$. Life is hard.

To change the x-, y-, and z-vectors, you can use the following options:

- `x=`⟨*dimension*⟩ Sets the x-vector of PGF's xyz-coordinate system to point ⟨*dimension*⟩ to the right, that is, to $(\langle dimension\rangle, 0pt)$. The default is 1cm.

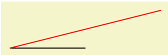

```
\begin{tikzpicture}
  \draw                  (0,0)   -- +(1,0);
  \draw[x=2cm,color=red] (0,0.1) -- +(1,0);
\end{tikzpicture}
```



```
\tikz \draw[x=1.5cm] (0,0) grid (2,2);
```

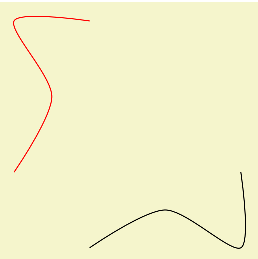
The last example shows that the size of steppings in grids, just like all other dimensions, are not affected by the x-vector. After all, the x-vector is only used to determine the coordinate of the upper right corner of the grid.

- x=⟨*coordinate*⟩ Sets the x-vector of PGF's xyz-coordinate system to the specified ⟨*coordinate*⟩. If ⟨*coordinate*⟩ contains a comma, it must be put in braces.



```
\begin{tikzpicture}
  \draw                          (0,0) -- (1,0);
  \draw[x={(2cm,0.5cm)},color=red] (0,0) -- (1,0);
\end{tikzpicture}
```

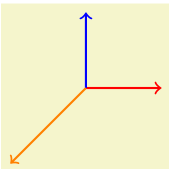
You can use this, for example, to exchange the meaning of the x- and y-coordinate.



```
\begin{tikzpicture}[smooth]
  \draw plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
  \draw[x={(0cm,1cm)},y={(1cm,0cm)},color=red]
        plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
\end{tikzpicture}
```

- y=⟨*value*⟩ Works like the `x=` option, only if ⟨*value*⟩ is a dimension, the resulting vector points to (0, ⟨*value*⟩).

- z=⟨*value*⟩ Works like the `z=` option, but now a dimension is means the point (⟨*value*⟩, ⟨*value*⟩).

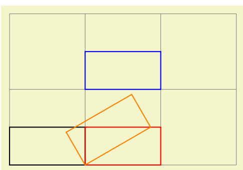


```
\begin{tikzpicture}[z=-1cm,->,thick]
  \draw[color=red] (0,0,0) -- (1,0,0);
  \draw[color=blue] (0,0,0) -- (0,1,0);
  \draw[color=orange] (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

17.3 Coordinate Transformations

PGF and Ti*k*Z allow you to specify *coordinate transformations*. Whenever you specify a coordinate as in `(1,0)` or `(1cm,1pt)` or `(30:2cm)`, this coordinate is first "reduced" to a position of the form "x points to the right and y points upwards." For example, `(1in,5pt)` is reduced to "$72\frac{72}{100}$ points to the right and 5 points upwards" and `(90:100pt)` means "0pt to the right and 100 points upwards."

The next step is to apply the current *coordinate transformation matrix* to the coordinate. For example, the coordinate transformation matrix might currently be set such that it adds a certain constant to the x value. Also, it might be setup such that it, say, exchanges the x and y value. In general, any "standard" transformation like translation, rotation, slanting, or scaling or any combination thereof is possible. (Internally, PGF keeps track of a coordinate transformation matrix very much like the concatenation matrix used by PDF or PostScript.)

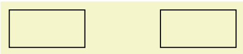


```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw (0,0) rectangle (1,0.5);
  \begin{scope}[xshift=1cm]
    \draw             [red]    (0,0) rectangle (1,0.5);
    \draw[yshift=1cm] [blue]   (0,0) rectangle (1,0.5);
    \draw[rotate=30]  [orange] (0,0) rectangle (1,0.5);
  \end{scope}
\end{tikzpicture}
```

The most important aspect of the coordinate transformation matrix is *that it applies to coordinates only!* In particular, the coordinate transformation has no effect on things like the line width or the dash pattern or the shading angle. In certain cases, it is not immediately clear whether the coordinate transformation matrix *should* apply to a certain dimension. For example, should the coordinate transformation matrix apply to grids? (It does.) And what about the size of arced corners? (It does not.) The general rule is "If there is no 'coordinate' involved, even 'indirectly,' the matrix is not applied." However, sometimes, you simply have to try or look it up in the documentation whether the matrix will be applied.

Setting the matrix cannot be done directly. Rather, all you can do is to "add" another transformation to the current matrix. However, all transformations are local to the current TEX-group. All transformations are added using graphic options, which are described below.

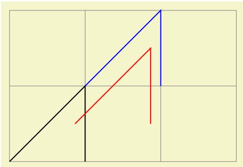
Transformations apply immediately when they are encountered "in the middle of a path" and they apply only to the coordinates on the path following the transformation option.



```
\tikz \draw (0,0) rectangle (1,0.5) [xshift=2cm] (0,0) rectangle (1,0.5);
```

A final word of warning: You should refrain from using "aggressive" transformations like a scaling of a factor of 10000. The reason is that all transformations are done using TEX, which has a fairly low accuracy. Furthermore, in certain situations it is necessary that Ti*k*Z *inverts* the current transformation matrix and this will fail if the transformation matrix is badly conditioned or even singular (if you do not know what singular matrices are, you are blessed).

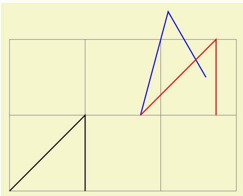
- `shift={`⟨*coordinate*⟩`}` adds the ⟨*coordinate*⟩ to all coordinates.



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                     (0,0) -- (1,1) -- (1,0);
  \draw[shift={(1,1)},blue]    (0,0) -- (1,1) -- (1,0);
  \draw[shift={(30:1cm)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

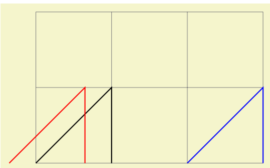
- `shift only` This option does not take any parameter. Its effect is to cancel all current transformations except for the shifting. This means that the origin will remain where it is, but any rotation around the origin or scaling relative to the origin or skewing will no longer have an effect.

  This option is useful in situtations where a complicated transformation is used to "get to a position," but you then wish to draw something "normal" at this position.



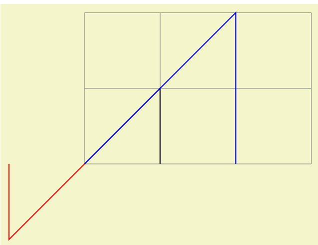
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                                     (0,0) -- (1,1) -- (1,0);
  \draw[rotate=30,xshift=2cm,blue]           (0,0) -- (1,1) -- (1,0);
  \draw[rotate=30,xshift=2cm,shift only,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `xshift=`⟨*dimension*⟩ adds ⟨*dimension*⟩ to the x value of all coordinates.



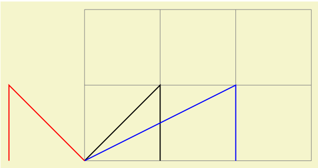
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                   (0,0) -- (1,1) -- (1,0);
  \draw[xshift=2cm,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[xshift=-10pt,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `yshift=`⟨*dimension*⟩ adds ⟨*dimension*⟩ to the y value of all coordinates.
- `scale=`⟨*factor*⟩ multiplies all coordinates by the given ⟨*factor*⟩. The ⟨*factor*⟩ should not be excessively large in absolute terms or very near to zero.



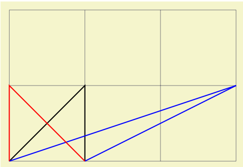
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                 (0,0) -- (1,1) -- (1,0);
  \draw[scale=2,blue]   (0,0) -- (1,1) -- (1,0);
  \draw[scale=-1,red]   (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `xscale`=⟨*factor*⟩ multiplies only the x-value of all coordinates by the given ⟨*factor*⟩.



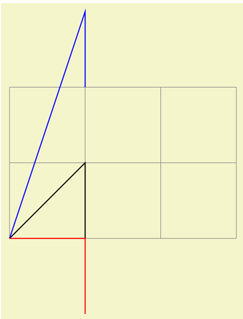
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                   (0,0) -- (1,1) -- (1,0);
  \draw[xscale=2,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[xscale=-1,red]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `yscale`=⟨*factor*⟩ multiplies only the y-value of all coordinates by ⟨*factor*⟩.
- `xslant`=⟨*factor*⟩ slants the coordinate horizontally by the given ⟨*factor*⟩:



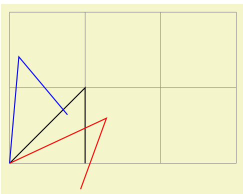
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                   (0,0) -- (1,1) -- (1,0);
  \draw[xslant=2,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[xslant=-1,red]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `yslant`=⟨*factor*⟩ slants the coordinate vertically by the given ⟨*factor*⟩:



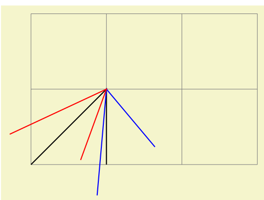
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                   (0,0) -- (1,1) -- (1,0);
  \draw[yslant=2,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[yslant=-1,red]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `rotate`=⟨*degree*⟩ rotates the coordinate system by ⟨*degree*⟩:



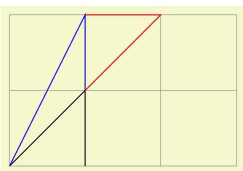
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                    (0,0) -- (1,1) -- (1,0);
  \draw[rotate=40,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[rotate=-20,red]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `rotate around`=`{`⟨*degree*⟩`:`⟨*coordinate*⟩`}` rotates the coordinate system by ⟨*degree*⟩ around the point ⟨*coordinate*⟩.



```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                                   (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={40:(1,1)},blue]  (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={-20:(1,1)},red]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `cm`=`{`⟨*a*⟩`,`⟨*b*⟩`,`⟨*c*⟩`,`⟨*d*⟩`,`⟨*coordinate*⟩`}` applies the following transformation to all coordinates: Let (x, y) be the coordinate to be transformed and let ⟨*coordinate*⟩ specify the point (t_x, t_y). Then the new coordinate is given by $\left(\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right)\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) + \left(\begin{smallmatrix} t_x \\ t_y \end{smallmatrix}\right)$. Usually, you do not use this option directly.



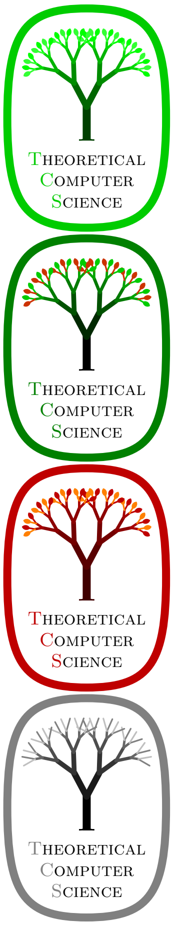
```
\begin{tikzpicture}
  \draw[style=help lines] (0,0) grid (3,2);
  \draw                                (0,0) -- (1,1) -- (1,0);
  \draw[cm={1,1,0,1,(0,0)},blue]     (0,0) -- (1,1) -- (1,0);
  \draw[cm={0,1,1,0,(1cm,1cm)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

- `reset cm` completely resets the coordinate transformation matrix to the identity matrix. This will destroy not only the transformations applied in the current scope, but also all transformations inherited from surrounding scopes. Do not use this option, unless you really, really know what you are doing.

Part IV
Libraries

by Till Tantau

In this part the library packages are documented. They provide additional predefined graphic objects like new arrow heads or new plot marks. These are not loaded by default since many users will not need them.




```
\tikzstyle{level 1}=
  [level distance=4.5mm,   trunk,        line width=1ex ,sibling angle=60]
\tikzstyle{level 2}=
  [level distance=3.5mm, trunk!80!leaf a,line width=.8ex,sibling angle=56]
\tikzstyle{level 3}=
  [level distance=2.75mm,trunk!60!leaf a,line width=.6ex,sibling angle=52]
\tikzstyle{level 4}=
  [level distance=2mm,   trunk!40!leaf a,line width=.4ex,sibling angle=48]
\tikzstyle{level 5}=
  [level distance=1mm,   trunk!20!leaf a,line width=.3ex,sibling angle=44]
\tikzstyle{level 6}=
  [level distance=1.75mm,leaf a,         line width=.2ex,sibling angle=40]

\pgfarrowsdeclare{leaf}{leaf}
  {\pgfarrowsleftextend{-2pt} \pgfarrowsrightextend{1pt}}
{
  \pgfpathmoveto{\pgfpoint{-2pt}{0pt}}
  \pgfpatharc{150}{30}{1.8pt}
  \pgfpatharc{-30}{-150}{1.8pt}
  \pgfusepathqfill
}

\newcommand{\tcslogo}[5]
{
  \colorlet{border}{#1}
  \colorlet{trunk}{#2}
  \colorlet{leaf a}{#3}
  \colorlet{leaf b}{#4}
  \begin{tikzpicture}
    \scriptsize\scshape
    \draw[border,line width=1ex,yshift=.3cm,
          yscale=1.45,xscale=1.05,looseness=1.42]
      (1,0) to [out=90, in=0]    (0,1)  to [out=180,in=90]  (-1,0)
            to [out=-90,in=-180] (0,-1) to [out=0,  in=-90] (1,0) -- cycle;

    \coordinate (root) [grow cyclic,rotate=90]
    child {
      child [cap=round] foreach \a in {0,1} {
        child foreach \b in {0,1} {
          child foreach \c in {0,1} {
            child foreach \d in {0,1} {
              child foreach \leafcolor in {leaf a,leaf b}
                { edge from parent [color=\leafcolor,-#5] }
        } } }
      } edge from parent [shorten >=-1pt,serif cm-,cap=butt]
    };

    \node [text centered,text width=2cm,below] at (0pt,-.5ex)
    { \textcolor{border}{T}heoretical \\ \textcolor{border}{C}omputer \\
      \textcolor{border}{S}cience };
  \end{tikzpicture}
}
\begin{minipage}{3cm}
  \tcslogo{green!80!black}{green!25!black}{green}{green!80}{leaf}\\
  \tcslogo{green!50!black}{black}{green!80!black}{red!80!green}{leaf}\\
  \tcslogo{red!75!black}{red!25!black}{red!75!black}{orange}{leaf}\\
  \tcslogo{black!50}{black}{black!50}{black!25}{}
\end{minipage}
```

18 Arrow Tip Library

```
\usepgflibrary{arrows}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[arrows]  % ConTeXt and pure pgf
\usetikzlibrary{arrows} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[arrows] % ConTeXt when using TikZ
```

The package defines additional arrow tips, which are described below. See page 322 for the arrows tips that are defined by default. Note that neither the standard packages nor this package defines an arrow name containing > or <. These are left for the user to defined as he or she sees fit.

18.1 Triangular Arrow Tips

`latex'`	yields thick and thin
`latex' reversed`	yields thick and thin
`stealth'`	yields thick and thin
`stealth' reversed`	yields thick and thin
`triangle 90`	yields thick and thin
`triangle 90 reversed`	yields thick and thin
`triangle 60`	yields thick and thin
`triangle 60 reversed`	yields thick and thin
`triangle 45`	yields thick and thin
`triangle 45 reversed`	yields thick and thin
`open triangle 90`	yields thick and thin
`open triangle 90 reversed`	yields thick and thin
`open triangle 60`	yields thick and thin
`open triangle 60 reversed`	yields thick and thin
`open triangle 45`	yields thick and thin
`open triangle 45 reversed`	yields thick and thin

18.2 Barbed Arrow Tips

`angle 90`	yields thick and thin
`angle 90 reversed`	yields thick and thin
`angle 60`	yields thick and thin
`angle 60 reversed`	yields thick and thin
`angle 45`	yields thick and thin
`angle 45 reversed`	yields thick and thin
`hooks`	yields thick and thin
`hooks reversed`	yields thick and thin

18.3 Bracket-Like Arrow Tips

`[-]`	yields thick and thin
`]-[`	yields thick and thin
`(-)`	yields thick and thin
`)-(`	yields thick and thin

18.4 Circle and Diamond Arrow Tips

`o`	yields thick and thin
`*`	yields thick and thin
`diamond`	yields thick and thin
`open diamond`	yields thick and thin

18.5 Serif-Like Arrow Tips

`serif cm`	yields thick and thin

18.6 Partial Arrow Tips

`left to`	yields thick and thin
`left to reversed`	yields thick and thin
`right to`	yields thick and thin
`right to reversed`	yields thick and thin
`left hook`	yields thick and thin
`left hook reversed`	yields thick and thin
`right hook`	yields thick and thin
`right hook reversed`	yields thick and thin

18.7 Line Caps

`round cap`	yields for line width 1ex
`butt cap`	yields for line width 1ex
`triangle 90 cap`	yields for line width 1ex
`triangle 90 cap reversed`	yields for line width 1ex
`fast cap`	yields for line width 1ex
`fast cap reversed`	yields for line width 1ex

19 Automata Drawing Library

```
\usetikzlibrary{automata} % LaTeX and plain TeX
\usetikzlibrary[automata] % ConTeXt
```

This packages provides shapes and styles for drawing finite state automata and Turing machines.

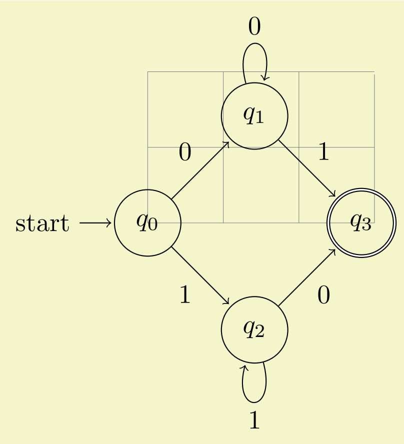
19.1 Drawing Automata

The automata drawing library is intended to make it easy to draw finite automata and Turing machines. It does not cover every situation imaginable, but most finite automata and Turing machines found in text books can be drawn in a nice and convenient fashion using this library.

To draw an automaton, proceed as follows:

1. For each state of the automaton, there should be one node with the option `state`.
2. To place the states, you can either use absolute positions or relative positions, using options like `above of` or `right of`.
3. Give a unique name to each state node.
4. Accepting and initial states are indicated by adding the options `accepting` and `initial`, respectively, to the state nodes.
5. Once the states are fixed, the edges can be added. For this, the `edge` operation is most useful. It is, however, also possible to add edges after each node has been placed.
6. For loops, use the `edge [loop]` operation.

Let us now see how this works for a real example. Let us consider a nondeterminsitic four state automaton that checks whether an contains the sequence 0^*1 or the sequence 1^*0.




```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,auto]
  \draw[help lines] (0,0) grid (3,2);

  \node[state,initial]  (q_0)                      {$q_0$};
  \node[state]          (q_1) [above right of=q_0] {$q_1$};
  \node[state]          (q_2) [below right of=q_0] {$q_2$};
  \node[state,accepting](q_3) [below right of=q_1] {$q_3$};

  \path[->] (q_0) edge              node        {0} (q_1)
                  edge              node [swap] {1} (q_2)
            (q_1) edge              node        {1} (q_3)
                  edge [loop above] node        {0} ()
            (q_2) edge              node [swap] {0} (q_3)
                  edge [loop below] node        {1} ();
\end{tikzpicture}
```

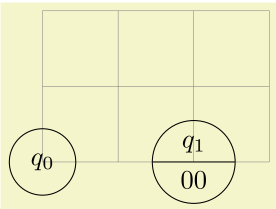
19.2 States With and Without Output

The `state` style actually just "selects" a default underlying style. Thus, you can define multiple new complicated state style and then simply set the `state` style to your given style to get the desired kind of styles.

By default, the following state styles are defined:

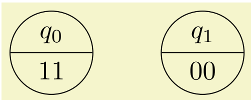
- `style=state without output` This node style causes nodes to be drawn circles. Also, this style calls `every state`.

- `style=state with output` This node style causes nodes to be drawn as split circles, that is, using the `circle split` shape. In the upper part of the shape you have the name of the style, in the lower part the output is placed. To specify the output, use the command `\nodepart{lower}` inside the node. This style also calls `every state`.



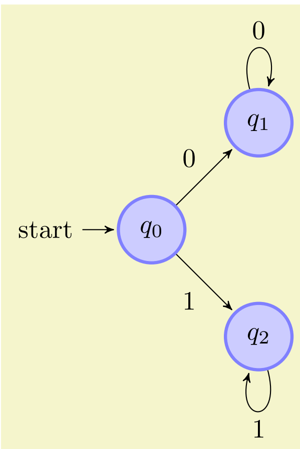

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \node[state without output] {$q_0$};

  \node[state with output] at (2,0) {$q_1$ \nodepart{lower} $00$};
\end{tikzpicture}
```

- `style=state` This style is set to `state without output` by default. You should redefine it to something else, if you wish to use states of a different nature.




```
\begin{tikzpicture}
  \tikzstyle{state}=[state with output]
  \node[state]          {$q_0$ \nodepart{lower} $11$};
  \node[state] at (2,0) {$q_1$ \nodepart{lower} $00$};
\end{tikzpicture}
```

- `style=every state` This style is used by `state with output` and also by `state without output`. By default, it does nothing, but you can use it to make your state look more fancy:




```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,>=stealth’]
  \tikzstyle{every state}=[draw=blue!50,very thick,fill=blue!20]

  \node[state,initial]  (q_0)                        {$q_0$};
  \node[state]          (q_1) [above right of=q_0] {$q_1$};
  \node[state]          (q_2) [below right of=q_0] {$q_2$};

  \path[->] (q_0) edge               node [above left]  {0} (q_1)
                  edge               node [below left]  {1} (q_2)
            (q_1) edge [loop above] node                {0} ()
            (q_2) edge [loop below] node                {1} ();
\end{tikzpicture}
```

19.3 Initial and Accepting States

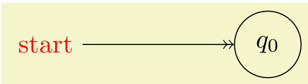
The styles `initial` and `accepting` are similar to the `state` style as they also just select an "underlying" style, which installs the actual settings for initial and accepting states.

For initial states, there is only one predefined way of drawing them, so the two-stage mechanism is not really necessary, but, perhaps, I will find another way of drawing them in the literature some time.

Let us start with the initial states.

- `style=initial` This style simply selects `initial by arrow`.

- `style=`initial by arrow This style causes an arrow and, possibly, some text to be added to the node. The arrow points from the text to the node. The node text and the direction and the distance can be set using the following options:
  - initial text=⟨*text*⟩ sets the text to be used. Use an empty text to suppress all text. The default is `start`.
  - initial where=⟨*direction*⟩ set the place where the text should be shown. Allowed values are `above`, `below`, `left`, and `right`.
  - intial distance=⟨*distance*⟩ is the length of the arrow leading from the text to the state node.
- `style=`every initial by arrow This style is executed at the beginning of every path that contains the arrow and the text. You can use it to, say, make the text red or whatever.

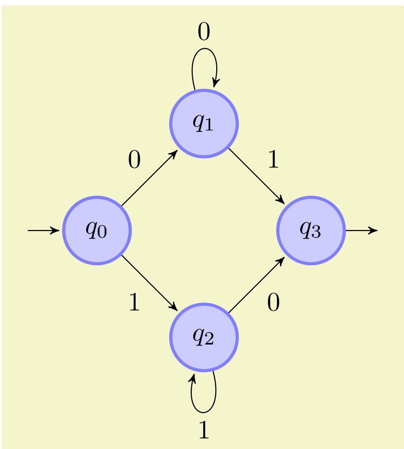



```
\begin{tikzpicture}
  \tikzstyle{every initial by arrow}=[text=red,->>]
  \node[state,initial,initial distance=2cm] {$q_0$};
\end{tikzpicture}
```

- `style=`initial above is a shorthand for `initial by arrow,initial where=above`
- `style=`initial below works similarly to the previous option.
- `style=`initial left works similarly to the previous option.
- `style=`initial right works similarly to the previous option.

For the accepting states, the sitation is similar: There is also an `accepting` style that selects the way accepting states are rendered. However, there are now two options: First, `accepting by arrow`, which works the same way as `initial by arrow`, only with the direction of arrow reversed, and `accepting by double`, where accepting states get a double line around them.

- `style=`accepting This style selects `accepting by double` by default. You can replace this by the style `accepting by arrow` to get accepting states with an arrow leaving them.
- `style=`accepting by double Specifies that the node should get a double line around it.
- `style=`accepting by arrow This style causes an arrow and, possibly, some text to be added to the node. The arrow points to the text from the node.
  - accepting text=⟨*text*⟩ sets the text to be used. Use an empty text to suppress all text. This is the default.
  - accepting where=⟨*direction*⟩ set the place where the text should be shown. Allowed values are `above`, `below`, `left`, and `right`.
  - intial distance=⟨*distance*⟩ is the length of the arrow.

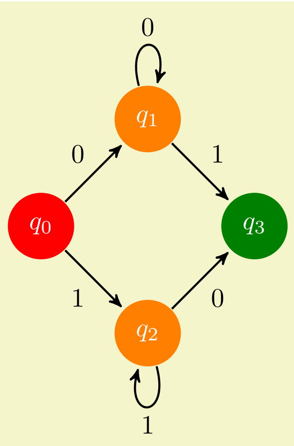


```
\begin{tikzpicture}
  [shorten >=1pt,node distance=2cm,>=stealth',initial text=]

  \tikzstyle{every state}=[draw=blue!50,very thick,fill=blue!20]
  \tikzstyle{accepting}=[accepting by arrow]

  \node[state,initial]  (q_0)                      {$q_0$};
  \node[state]          (q_1) [above right of=q_0] {$q_1$};
  \node[state]          (q_2) [below right of=q_0] {$q_2$};
  \node[state,accepting](q_3) [below right of=q_1] {$q_3$};

  \path[->] (q_0) edge              node [above left]  {0} (q_1)
                  edge              node [below left]  {1} (q_2)
            (q_1) edge              node [above right] {1} (q_3)
                  edge [loop above] node               {0} ()
            (q_2) edge              node [below right] {0} (q_3)
                  edge [loop below] node               {1} ();
\end{tikzpicture}
```

- `style=`every accepting by arrow This style is executed at the beginning of every path that contains the arrow and the text.
- `style=`accepting above is a shorthand for `accepting by arrow,accepting where=above`
- `style=`accepting below works similarly to the previous option.
- `style=`accepting left works similarly to the previous option.
- `style=`accepting right works similarly to the previous option.

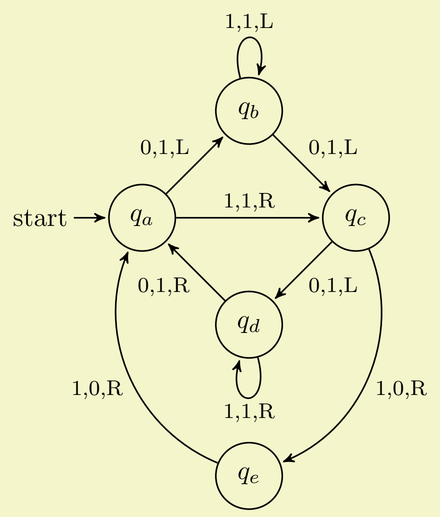
19.4 Examples

In the following example, we once more typeset the automaton presented in the previous sections. This time, we use the following rule for accepting/initial state: Initial states are red, accepting states are green, and normal states are orange. Then, we must find a path from a red state to a green state.




```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,>=stealth',thick]
  \tikzstyle{every state}=[fill,draw=none,orange,text=white]
  \tikzstyle{accepting}=[green!50!black,text=white]
  \tikzstyle{initial}=  [red,text=white]

  \node[state,initial]  (q_0)                      {$q_0$};
  \node[state]          (q_1) [above right of=q_0] {$q_1$};
  \node[state]          (q_2) [below right of=q_0] {$q_2$};
  \node[state,accepting](q_3) [below right of=q_1] {$q_3$};

  \path[->] (q_0) edge              node [above left]  {0} (q_1)
                  edge              node [below left]  {1} (q_2)
            (q_1) edge              node [above right] {1} (q_3)
                  edge [loop above] node               {0} ()
            (q_2) edge              node [below right] {0} (q_3)
                  edge [loop below] node               {1} ();
\end{tikzpicture}
```

The next example is the current candidate for the five-state busiest beaver:




```
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,%
                    auto,node distance=2cm,semithick,
                    inner sep=2pt,bend angle=45]
  \node[initial,state] (A)                    {$q_a$};
  \node[state]         (B) [above right of=A] {$q_b$};
  \node[state]         (D) [below right of=A] {$q_d$};
  \node[state]         (C) [below right of=B] {$q_c$};
  \node[state]         (E) [below of=D]       {$q_e$};

  \tikzstyle{every node}=[font=\footnotesize]

  \path (A) edge              node {0,1,L} (B)
            edge              node {1,1,R} (C)
        (B) edge [loop above] node {1,1,L} (B)
            edge              node {0,1,L} (C)
        (C) edge              node {0,1,L} (D)
            edge [bend left]  node {1,0,R} (E)
        (D) edge [loop below] node {1,1,R} (D)
            edge              node {0,1,R} (A)
        (E) edge [bend left]  node {1,0,R} (A);
\end{tikzpicture}
```

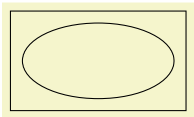
20 Background Library

```
\usetikzlibrary{backgrounds} % LATEX and plain TEX
\usetikzlibrary[backgrounds] % ConTEXt
```

This library defines "backgrounds" for pictures. This does not refer to background pictures, but rather to frames drawn around and behind pictures. For example, this package allows you to just add the `framed` option to a picture to get a rectangular box around your picture or `gridded` to put a grid behind your picture.

When this package is loaded, the following styles become available:

- `style=show background rectangle` This style causes a rectangle to be drawn behind your graphic. This style option must be given to the `{tikzpicture}` environment or to the `\tikz` command.



```
\begin{tikzpicture}[show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

The size of the background rectangle is determined as follows: We start with the bounding box of the picture. Then, a certain separator distance is added on the sides. This distance can be different for the x- and y-directions and can be set using the following options:

  – `inner frame xsep=`⟨*dimension*⟩ Sets the additional horizontal separator distance for the background rectangle. The default is `1ex`.

  – `inner frame ysep=`⟨*dimension*⟩ Same for the vertical separator distance.

  – `inner frame sep=`⟨*dimension*⟩ sets the horizontal and vertical separator distances simultaneously.

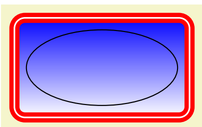
The following two styles make setting the inner separator a bit easier to remember:

  – `style=tight background` Sets the inner frame separator to 0pt. The background rectangle will have the size of the bounding box.

  – `style=loose background` Sets the inner frame separator to 2ex.

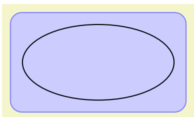
You can influence how the background rectangle is rendered by setting the following style:

  – `style=background rectangle` This style dictates how the background rectangle is drawn or filled. By default this style is set to `draw`, which causes the path of the background rectangle to be drawn in the usual way. Setting this style to, say, `fill=blue!20` causes a light blue background to be added to the picture. You can also use more fancy settings as shown in the following example:



```
\tikzstyle{background rectangle}=
  [double,ultra thick,draw=red,top color=blue,rounded corners]
\begin{tikzpicture}[show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

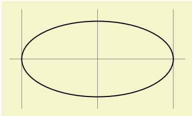
Naturally, no one in their right mind would use the above, but here is a nice background:



```
\tikzstyle{background rectangle}=
  [draw=blue!50,fill=blue!20,rounded corners=1ex]
\begin{tikzpicture}[show background rectangle]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

- `style=framed` This is a shorthand for `show background rectangle`.

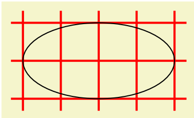
- `style=show background grid` This style behaves similarly to the `show background rectangle` style, but it will not use a rectangle path, but a grid. The lower left and upper right corner of the grid is computed in the same way as for the background rectangle:



```
\begin{tikzpicture}[show background grid]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

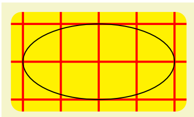
You can influence the background grid by setting the following style:

– `style=background grid` This style dictates how the background grid path is drawn. The default is `draw,help lines`.



```
\tikzstyle{background grid}=[thick,draw=red,step=.5cm]
\begin{tikzpicture}[show background grid]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

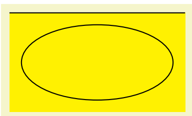
This option can be combined with the `framed` option (use the `framed` option first):



```
\tikzstyle{background grid}=[thick,draw=red,step=.5cm]
\tikzstyle{background rectangle}=[rounded corners,fill=yellow]
\begin{tikzpicture}[framed,gridded]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

– `style=gridded` This is a shorthand for `show background grid`.

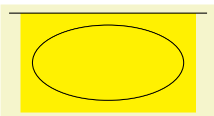
- `style=show background top` This style causes a single line to be drawn at the top of the background rectangle. Normally, the line coincides exactly with the top line of the background rectangle:



```
\tikzstyle{background rectangle}=[fill=yellow]
\begin{tikzpicture}[framed,show background top]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

The following option allows you to lengthen (or shorten) the line:

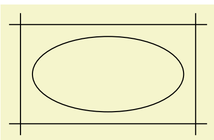
– `outer frame xsep=`⟨*dimension*⟩ The ⟨*dimension*⟩ is added at the left and right side of the line.



```
\tikzstyle{background rectangle}=[fill=yellow]
\begin{tikzpicture}
    [framed,show background top,outer frame xsep=1ex]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

– `outer frame ysep=`⟨*dimension*⟩ This option does not apply to the top line, but to the left and right lines, see below.

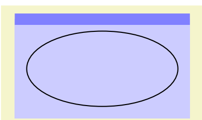
– `outer frame sep=`⟨*dimension*⟩ Sets both the x- and y-separation.



```
\tikzstyle{background rectangle}=[fill=blue!20]
\begin{tikzpicture}
  [outer frame sep=1ex,%
   show background top,%
   show background bottom,%
   show background left,%
   show background right]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

You can influence how the line is drawn grid by setting the following style:

– `style=background top` Default is `draw`.



```
\tikzstyle{background rectangle}=[fill=blue!20]
\tikzstyle{background top}=[draw=blue!50,line width=1ex]
\begin{tikzpicture}[framed,show background top]
  \draw (0,0) ellipse (10mm and 5mm);
\end{tikzpicture}
```

- `style=show background bottom` works like the style for the top line.
- `style=show background left` works like the style for the top line.
- `style=show background right` works like the style for the top line.

21 Calendar Library

`\usetikzlibrary{calendar} % LaTeX and plain TeX`
`\usetikzlibrary[calendar] % ConTeXt`

The library defines the `\calendar` command, which can be used to typeset calendars. The command relies on the `\pgfcalendar` command from the `pgfcalendar` package, which is loaded automatically.

The `\calendar` command is quite configurable, allowing you to produce all kinds of different calendars.

21.1 Calendar Command

The core command for creating calendars in Ti*k*Z is the `\calendar` command. It is available only inside `{tikzpicture}` environments (similar to, say, the `\draw` command).

`\calendar`⟨*calendar specification*⟩`;`

The syntax for this command is similar to commands like `\node` or `\matrix`. However, it has its complete own parser and only those commands described in the following will be recognized, nothing else. Note, furthermore, that a ⟨*calendar specification*⟩ is not a path specification, indeed, no path is created for the calendar.

The specification syntax. The ⟨*calendar specification*⟩ must be a sequence of elements, each of which has one of the following structures:

- `[`⟨*options*⟩`]`
  You provide ⟨*options*⟩ in square brackets as in `[red,draw=none]`. These ⟨*options*⟩ can be any Ti*k*Z option and they apply to the whole calendar. You can provide this element multiple times, the effect accumulates.
- `(`⟨*name*⟩`)`
  This has the same effect as saying `[name=`⟨*name*⟩`]`. The effect of providing a ⟨*name*⟩ is explained later. Note alreadys that *a calendar is not a node* and the ⟨*name*⟩ is *not the name of a node*.
- `at (`⟨*coordinate*⟩`)`
  This has the same effect as saying `[at=(`⟨*coordinate*⟩`)]`.
- `if (`⟨*date condition*⟩`)` ⟨*options or commands*⟩`else`⟨*else options or commands*⟩
  The effect of such an `if` is explained later.

At the beginning of every calendar, the following style is used:

- `style=every calendar` This style is empty be default.

The date range. The overall effect of the `\calendar` command is to execute code for each day of a range of dates. This range of dates is set using the following option:

- `dates=`⟨*start date*⟩ `to` ⟨*end date*⟩ This option specifies the date range. Both the start and end date are specified as described on page 250. In short: You can provide ISO-format type dates like `2006-01-02`, you can replace the day of month by `last` to refer to the last day of a month (so `2006-02-last` is the same as `2006-02-28`), and you can add a plus sign followed by a number to specify an offset (so `2006-01-01+-1` is the same as `2005-12-31`).

It will be useful to fix two pieces of terminology for the following descriptions: The `\calendar` command iterates over the dates in the range. The *current date* refers to the current date the command is processing as it iterates over the dates. For each current date code is executed, which will be called the *current date code*. The current date code consists of different parts, to be detailed later.

The central part of the current date code is the execution of the code `\tikzdaycode`. By default, this code simply produces a node whose text is set to the day of month. This means that unless further action is taken, all days of a calendar will be put on top of each other! To avoid this, you must modify the current date code to shift days around appropriately. Predefined arrangements like `day list downward` or `week list` do this for you, but you can define arrangements yourself. Since defining an arrangement is a bit tricky, it is explained only later on. For the time being, let us use a predefined arrangement to produce our first calendar:

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list];
```

Changing the spacing. In the above calendar, the spacing between the days is determined by the numerous options. Most arrangement do not use all of these options, but only those that apply naturally.

- `day xshift`=⟨*dimension*⟩ specifies the horizontal shift between days. This is not the gap between days, but the shift between the anchors of their nodes. The default is `3.5ex`.

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,day xshift=3ex];
```

- `day yshift`=⟨*dimension*⟩ specifies the vertical shift between days. Again, this is the shift between the anchors of their nodes. The default is `3ex`.

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,day yshift=2ex];
```

- `month xshift`=⟨*dimension*⟩ specifies an additional horizontal shift between different months.
- `month yshift`=⟨*dimension*⟩ specifies an additional vertical shift between different months.

1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31 1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29

```
\tikz \calendar[dates=2000-01-01 to 2000-02-last,week list,
                month yshift=0pt];
```

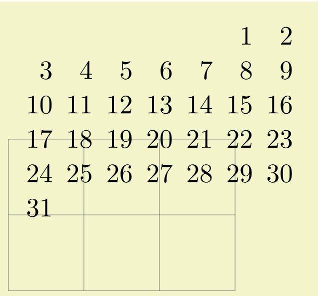
1 2
3 4 5 6 7 8 9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

1 2 3 4 5 6
7 8 9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29

```
\tikz \calendar[dates=2000-01-01 to 2000-02-last,week list,
                month yshift=1cm];
```

Changing the position of the calendar. The calendar is placed in such a way that, normally, the anchor of the first day label is at the origin. This can be changed by using the `at` option. When you say `at={(1,1)}`, this anchor of the first day will lie at coordinate $(1, 1)$.

In general, arrangements will not always place the anchor of the first day at the origin. Sometimes, additional spacing rules get in the way. There are different ways of addressing this problem: First, you can just ignore it. Since calendars are often placed in their own `{tikzpicture}` and since their size if computed automatically, the exact position of the origin often does not matter at all. Second, you can put the calendar inside a node as in `...node {\tikz \calendar...}`. This allows you to position the node in the normal ways using the node's anchors. Third, you can be very clever and use a single-cell matrix. The advantage is that a matrix allows you to provide any anchor of any node inside the matrix as an anchor for the whole matrix. For example, the following calendar is placed in such a way the center of 2000-01-20 lies on the position $(2, 2)$:

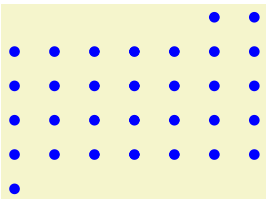



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \matrix [anchor=cal-2000-01-20.center] at (2,2)
  { \calendar(cal)[dates=2000-01-01 to 2000-01-31,week list]; \\};
\end{tikzpicture}
```

Unfortunately, the matrix-base positions, which is the cleanest way, isnot as portable as the other approaches (it currently does not work with the SVG backend for instance).

Changing the appearance of days. As mentioned before, each day in the above calendar is produced by an execution of the `\tikzdaycode`. Each time this code is executed, the coordinate system will have been setup appropriately to place the day of the month correctly. You can change both the code and its appearance using the following options.

- `day code`=⟨*code*⟩ This option allows you to change the code that is executed for each day. The default is to create a node with an appropriate name, but you can change this:



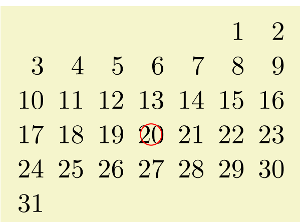
```
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                day code={\fill[blue] (0,0) circle (2pt);}];
```

The default code is the following:

```
\node[name=\pgfcalendarsuggestedname,every day]{\tikzdaytext};
```

The first part causes the day nodes to be accessible via the following names: If ⟨*name*⟩ is the name given to the calendar via a `name=` option or via the specification element `(`⟨*name*⟩`)`, then `\pgfcalendarsuggestedname` will expand to ⟨*name*⟩-⟨*date*⟩, where ⟨*date*⟩ is the date of the day that is currently being processed in ISO format .

For example, if January 1, 2006 is being processed and the calendar has been named `mycal`, then the node containg the `1` for this date will be names `mycal-2006-01-01`. You can later reference this node.




```
\begin{tikzpicture}
  \calendar (mycal) [dates=2000-01-01 to 2000-01-31,week list];

  \draw[red] (mycal-2000-01-20) circle (4pt);
\end{tikzpicture}
```

- `day text=`⟨*text*⟩ This option changes the setting of the `\tikzdaytext`. By default, this macro simply yields the current day of month, but you can change it arbitrarily. Here is a silly example:

					x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x						

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                day text=x];
```

More useful examples are based on using the `\%` command. This command is redefined inside a `\pgfcalendar` to mean the same as `\pgfcalendarshorthand`. (The original meaning of `\%` is lost inside the calendar, you need to save if before the calendar if you really need it.)

The `\%` inserts the current day/month/year/day of week in a certain format into the text. The first letter following the `\%` selects the type (permisslbe values are `d`, `m`, `y`, `w`), the second letter specifies how the value should be displayed (`-` means numerically, `=` means numerically with leading space, `0` means numerically with leading zeros, `t` means textual, and `.` means textual, abbreviated). For example `\%d0` gives the day with a leading zero (for more details see the description of `\pgfcalendarshorthand` on page 255).

Let us redefine the `day text` so that it yields the day with a leading zero:

					01	02
03	04	05	06	07	08	09
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                day text=\%d0];
```

- `style=every day` This style is executed by the default node code for each day. The default setting is

```
anchor=base east
```

The `every day` style is useful for changing the way days look. For example, let us make all days red:

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\tikzstyle{every day}+=[red]
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list];
```

Changing the appearance of month and year labels. In addition to the days of a calendar, labels for the months and even years (for really long calendars) can be added. These labels are only added once per month or year and this is not done by default. Rather, special styles starting with `month label` place these labels and make them visible:

January

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

February

	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

```
\tikz \calendar[dates=2000-01-01 to 2000-02-last,week list,
                month label above centered];
```

The following options change the appearance of the month and year label:

- `month code`=⟨*code*⟩ This option allows you to specify what the macro `\tikzmonthcode` should expand to.

  By default, the `\tikzmonthcode` it is set to

  ```
  \node[every month]{\tikzmonthtext};
  ```

  Note that this node is not named by default.
- `month text`=⟨*text*⟩ This option allows you to change the macro `\tikzmonthtext`. By default, the month text is a long textual presentation of the current month being typeset.

January 2000

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                month label above centered,
                month text=\textcolor{red}{\%mt} \%y-];
```

- `style=every month` This style, which is empty by default, can be used to change the appearance of month labels.
- `year code`=⟨*code*⟩ Works like `month code`, only for years.
- `year text`=⟨*text*⟩ Works like `month text`, only for years.
- `style=every year` Works like `every month`, only for years.

Date ifs. Much of the power of the `\calendar` command comes from the use of conditionals. There are two equivalent way of specifying such a conditional. First, you can add the text `if (`⟨*conditions*⟩`)` ⟨*code or options*⟩ to your ⟨*calendar specification*⟩, possibly followed by `else`⟨*else code or options*⟩. You can have multiple such conditionals (but you cannot nest them in this simple manner). The second way is to use the following option:

- `if=(`⟨*coditions*⟩`)`⟨*code or options*⟩`else`⟨*else code or options*⟩ This option has the same effect as giving a corresponding if in the ⟨*calendar specification*⟩. The option is mostly useful for use in the `every calendar` style, where you cannot provide if conditionals otherwise.

Now, regardless of how you specify a conditional, it has the following effect (individually and independently for each date in the calendar):

1. It is checked whether the current date is one of the possibilities listed in ⟨*coditions*⟩. An example of such a condition is `Sunday`. Thus, when you write `if (Saturday,Sunday) {foo}`, then `foo` will be executed for every day in the calendar that is a Saturday *or* a Sunday.

The command `\ifdate` and, thereby, `\pgfcalendarifdate` are used to evaluate the ⟨*conditions*⟩, see page 251 for a complete list of possible tests. The most useful tests are: Tests like `Monday` and so on, `workday` for the days Monday to Friday, `weekend` for Saturday and Sunday, `equals` for testing whether the current date equals a given date, `at least` and `at least` for comparing the current date with a given date.

2. If the date passes the check, the ⟨*code or options*⟩ is evaluated in a manner to be described in a moment; if the date fails, the ⟨*else code or options*⟩ is evaluated, if present.

   The ⟨*code or options*⟩ can either be some code. This is indicated by surrounding the code with curly braces. It can also be a list of Ti*k*Z options. This is indicated by surrounding the options with square brackets. For example in the date test `if (Sunday) {\draw...} else {\fill...}` there are two pieces of code involved. By comparison, `if (Sunday) [red] else [green]` involves two options.

   If ⟨*code or options*⟩ is code, it is simply executed (for the current day). If it is a list of options, these options are passed to a scope surrounding the current date.

Let us now have a look at some examples. First, we use a conditional to make all Sundays red.

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-31,week list]
    if (Sunday) [red];
```

Next, let us do something on a specific date:

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-31,week list]
    if (Sunday)             [red]
    if (equals=2000-01-20) {\draw (0,0) circle (8pt);};
```

You might wonder why the circle seems to be "off" the date. Actually, it is centered on the date, it is just that the date label uses the `base east` anchor, which shifts the label up and right. To overcome this problem we can change the anchor:

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\tikzstyle{every day}=[anchor=mid]
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-31,week list]
    if (Sunday)             [red]
    if (equals=2000-01-20) {\draw (0,0) circle (8pt);};
```

However, the single day dates are now no longer aligned correctly. For this, we can change the day text to `\%d=`, which adds a space at the beginning of single day text.

In the following, more technical information is covered. Most readers may wish to skip it.

The current date code. As mentioned earlier, for each date in the calendar the current date code is executed. It is the job of this code to shift around date nodes, to render the date nodes, to draw the month labels and to do all other stuff that is necessary to draw a calendar.

The current daet code consists of the following parts, in this order:

1. The before-scope code.
2. A scope is opened.
3. The at-begin-scope code.

4. All date-ifs from the ⟨*calendar specification*⟩ are executed.
5. The at-end-scope code.
6. The scope is closed.
7. The after-scope code.

All of the codes mentioned above can be changed using appropriate options, see below. In case you wonder why so many are needed, the reason is that the current date code as a whole is not surrounded by a scope or TeX group. This means that code executed in the before-scope code and in the after-scope code has an effect on all following days. For example, if the after-scope code modifies the transformation matrix by shifting everything downward, all following days will be shifted downward. If each day does this, you get a list of days, one below the other.

However, you do not always want code to have an effect on everything that follows. For instance, if a day has the date-if `if (Sunday) [red]`, we only want this Sunday to red, not all following days also. Similarly, sometimes it is easier to compute the position of a day relative to a fixed origin and we do not want any modifications of the transformation matrix to have an effect outside the scope.

By cleverly adjusting the different codes, all sorts of different day arrangements are possible.

- `execute before day scope=`⟨*code*⟩ This code is executed before everything else for each date. Multiple calls of this option have an accumulative effect. Thus, if you use this option twice, the code from the first use is used first for each day, followed by the code given the second time.
- `execute at begin day scope=`⟨*code*⟩ This code is execute before everything else inside the scope of the current date. Again, the effect is accumulative.
- `execute at end day scope=`⟨*code*⟩ This code is executed just before the day scope is closed. The effect is also accumulative, however, in reverse order. This is useful to pair, say, `\scope` and `\endscope` commands in at-begin- and at-end-code.
- `execute after day scope=`⟨*code*⟩ This is executed at the very end of the current date, outside the scope. The accumulation is also in reverse.

In the rest of the following subsections we have a look at how the different scope codes can be used to create different calendar arrangements.

21.1.1 Creating a Simple List of Days

We start with a list the days of the calendar, one day below the other. For this, we simply shift the coordinate system downward at the end of the code for each day. This shift must be *outside* the day scope as we want day shifts to accumulate. Thus, we use the following code:


```
\tikz
  \calendar [dates=2000-01-01 to 2000-01-08,
             execute after day scope=
               {\pgftransformyshift{-1em}}];
```

Clearly, we can use this approach to create day lists going up, down, right, left, or even diagonally.

21.1.2 Adding a Month Label

We now want to add a month label to the left of the beginning of each month. The idea is to do two things:

1. We add code that is executed only on the first of each month.
2. The code is executed before the actual day is rendered. This ensures that options applying to the days do not affect the month rendering.

We have two options where we should add the month code: Either we add it at the beginning of the day scope or before. Either will work fine, but it might be safer to put the code inside the scope to ensure that settings to not inadventerdly “leak outside.”

January 1
2
3
4
5
6
7
8

```
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-08,
     execute after day scope={\pgftransformyshift{-1em}},
     execute at begin day scope=
       {\ifdate{day of month=1}{\tikzmonthcode}{}},
     set style={{every month}+=[anchor=base east,xshift=-2em]}];
```

In the above code we used the `\ifdate{`⟨*condition*⟩`}{`⟨*then code*⟩`}{`⟨*else code*⟩`}` command, which is described on page 253 in detail and which has much the same effect as `if (`⟨*condition*⟩`){`⟨*then code*⟩`} else {`⟨*else code*⟩`}`, but works in normal code.

21.1.3 Creating a Week List Arrangement

Let us now address a more complicated arrangement: A week list. In this arrangement there is line for each week. The horizontal placement of the days is thus that all Mondays lie below each other, likewise for all Tuesdays, and so on.

In order to typeset this arrangement, we can use the following approach: The origin of the coordinate system rests at the anchor for the Monday of each week. That means that at the end of each week the origin is moved downward one line. On all other days, the origin at the end of the day code is the same as at the beginning. To position each day correctly, we use code inside and at the beginning of the day scope to horizontally shift the day according to its day of week.

1 2
3 4 5 6 7 8 9
10111213141516
17181920

```
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-20,
     % each day is shifted right according to the day of week
     execute at begin day scope=
       {\pgftransformxshift{\pgfcalendarcurrentweekday em}},
     % after each week, the origin is shifted downward:
     execute after day scope=
       {\ifdate{Sunday}{\pgftransformyshift{-1em}}{}}];
```

21.1.4 Creating a Month List Arrangement

For another example, let us create an arrangment that contains one line for each month. This is easy enough to do as for weeks, unless we add the following requirement: Again, we want all days in a column to have the same day of week. Since months start on different days of week, this means that each row has to have an individual offset.

One possible way is to use the following approach: After each month (or at the beginning of each month) we advance the vertical position of the offset by one line. For horizontal placement, inside the day scope we locally shift the day by its day of month. Furthermore, we must additionally shift the day to ensure that the first day of the month lies on the correct day of week column. For this, we remember this day of week the first time we see it.

1 2 3 4 5 6 7 8 910111213141516171819202122232425262728293031
1 2 3 4 5 6 7 8 91011121314151617181920212223242526272829

```
\newcount\mycount
\tikz
  \calendar
    [dates=2000-01-01 to 2000-02-last,
     execute before day scope=
     {
       \ifdate{day of month=1} {
         % Remember the weekday of first day of month
         \mycount=\pgfcalendarcurrentweekday
         % Shift downward
         \pgftransformyshift{-1em}
       }{}
     },
     execute at begin day scope=
     {
       % each day is shifted right according to the day of month
       \pgftransformxshift{\pgfcalendarcurrentday em}
       % and additionally according to the weekday of the first
       \pgftransformxshift{\the\mycount em}
     }];
```

21.2 Arrangements

An *arrangement* specifies how the days of calendar are arranged on the page. The calendar library defines a number of predefined arrangements.

We start with arrangements in which the days are listed in a long line.

- `style=day list downward` This style causes the days of a month to be typeset one below the other. The shift between days is given by `day yshift`. Between month an additional shift of `month yshift` is added.

28
29
30
31

1
2
3

```
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list downward,month yshift=1em];
```

- `style=day list upward` works as above, only the list grows upward instead of downward.

3
2
1

31
30
29
28

```
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list upward,month yshift=1em];
```

- `style=day list right` This style also works as before, but the list of days grows to the right. Instead of `day yshift` and `month yshift`, the values of `day xshift` and `month xshift` are used.

28 29 30 31   1 2 3

```
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list right,month xshift=1em];
```

- `style=day list left` as above, but the list grows left.

The next arrangement lists days weekwise.

- `style=week list` This style creates one row for each week in the range. The value of `day xshift` is used for the distance between days in each week row, the value of `day yshift` is used for the distance between rows. In both cases, "distance" refers to the distance between the anchors of the nodes of the days (or, more generally, the distance between the origins of the little pictures created for each day).

  The days inside each week are shifted such that Monday is always at the first position (to change this, you need to copy and then modify the code appropriately). If the date range does not start on a Monday, the first line will not start in the first column, but rather in the column appropriate for the first date in the range.

  At the beginning of each month (except for the first month in the range) an additional vertical space of `month yshift` is added. If this is set to `0pt` you get a continuous list of days.

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

```
\tikz
  \calendar [dates=2000-01-01 to 2000-02-last,week list];
```

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29					

```
\tikz
  \calendar [dates=2000-01-01 to 2000-02-last,week list,
             month yshift=0pt];
```

The following arrangement gives a very compact view of a whole year.

- `style=month list` In this arrangement there is a row for each month. As for the `week list`, the `day xshift` is used for the horizontal distance. For the vertical shft, `month yshift` is used.

  In each row, all days of the month are listed alongside each other. However, it is once more ensured that days in each column lie on the same day of week. Thus, the very first column contains only Mondays. If a month does not start with a Monday, its days are shifted to the right such that the days lie on the correct columns.

January						1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
February		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29							
March			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31				
April						1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30		
May	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31						
June				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30				
July						1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
August		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31					
September					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30			
October							1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
November			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30					
December					1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31		

```
\sffamily\scriptsize
\tikz
  \calendar [dates=2000-01-01 to 2000-12-31,
             month list,month label left,month yshift=1.25em]
            if (Sunday) [black!50];
```

21.3 Month Labels

For many calendars you may wish to add a labels to each month. We have already covered how month nodes are created and rendered in the description of the `\calendar` command: use `month text`, `every month`, and also `month code` (if necessary) to change the appearance of the month labels.

What we have not yet covered is where these labels are placed. By default, they are not placed at all as there is no good "default position" for them. Instead, you can use one of the following options to specify a position for the labels:

- `style=month label left` Places the month label to the left of the first day of the month. (For `week list` and `month list` where a month does not start on a Monday, the position is chosen "as if" the month had started on a Monday – which is usually exactly what you want.)

  28
29
30
31
February 1
2
3

  ```
  \tikz
    \calendar [dates=2000-01-28 to 2000-02-03,
               day list downward,month yshift=1em,
               month label left];
  ```

- `style=month label left vertical` This style works like the above style, only the label is rotated counterclockwise by 90 degrees.

  28
29
30
31
1
2
3
February

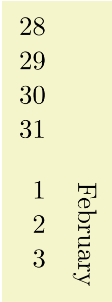
  ```
  \tikz
    \calendar [dates=2000-01-28 to 2000-02-03,
               day list downward,month yshift=1em,
               month label left vertical];
  ```

- `style=month label right` This style places the month label to the right of the row in which the first day of the month lies. This means that for a day list the label is to the right of the first day, for a week list it is to the right of the first week, and for a month list it is to the right of the whole month.

  28
29
30
31
1 February
2
3

  ```
  \tikz
    \calendar [dates=2000-01-28 to 2000-02-03,
               day list downward,month yshift=1em,
               month label right];
  ```

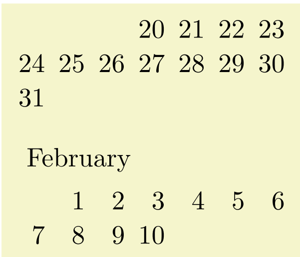
- `style=month label right vertical` as above, only the label is rotated clockwise by 90 degrees.




```
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list downward,month yshift=1em,
             month label right vertical];
```

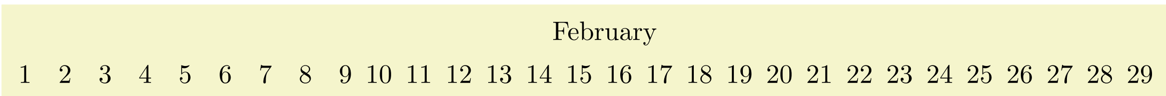
- `style=month label above left` This style places the month label above of the row of the first day, flushed left to the leftmost column. The amount by which the label is raised is fixed to `1.25em`; use the `yshift` option with the month node to modify this.


```
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list right,month xshift=1em,
             month label above left];
```

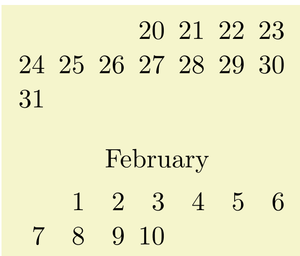



```
\tikz
  \calendar [dates=2000-01-20 to 2000-02-10,
             week list,month label above left];
```

- `style=month label above centered` works as above, only the label is centered above the row containing the first day.




```
\tikz
  \calendar [dates=2000-02-01 to 2000-02-last,
             day list right,month label above centered];
```




```
\tikz
  \calendar [dates=2000-01-20 to 2000-02-10,
             week list,month label above centered];
```

- `style=month label above right` works as above, but flushed right

20 21 22 23
24 25 26 27 28 29 30
31

February

1 2 3 4 5 6
7 8 9 10

```
\tikz
  \calendar [dates=2000-01-20 to 2000-02-10,
             week list,month label above right];
```

- `style=month label below left` Works like `month label above left`, only the label is placed below the row. This placement is not really useful with the `week list` arrangement, but rather with the `day list right` or `month list` arrangement.

  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
  February

  ```
  \tikz
    \calendar [dates=2000-02-01 to 2000-02-last,
               day list right,month label below left];
  ```

- `style=month label below centered` works like `month label above centered`, only below.

  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
  February

  ```
  \tikz
    \calendar [dates=2000-02-01 to 2000-02-last,
               day list right,month label below centered];
  ```

21.4 Examples

In the following, some example calendars are shown that come either from real applications or are just nice to look at.

Let us start with a year-2100-countdown, in which we cross out dates as we approach the big celebration. For this, we set the shape to `strike out` for these dates.

December 2099

~~1~~ ~~2~~ ~~3~~ ~~4~~ 5 6
~~7~~ ~~8~~ ~~9~~ ~~10~~ ~~11~~ 12 13
~~14~~ ~~15~~ ~~16~~ ~~17~~ ~~18~~ 19 20
~~21~~ ~~22~~ ~~23~~ ~~24~~ ~~25~~ 26 27
~~28~~ ~~29~~ 30 31

January 2100

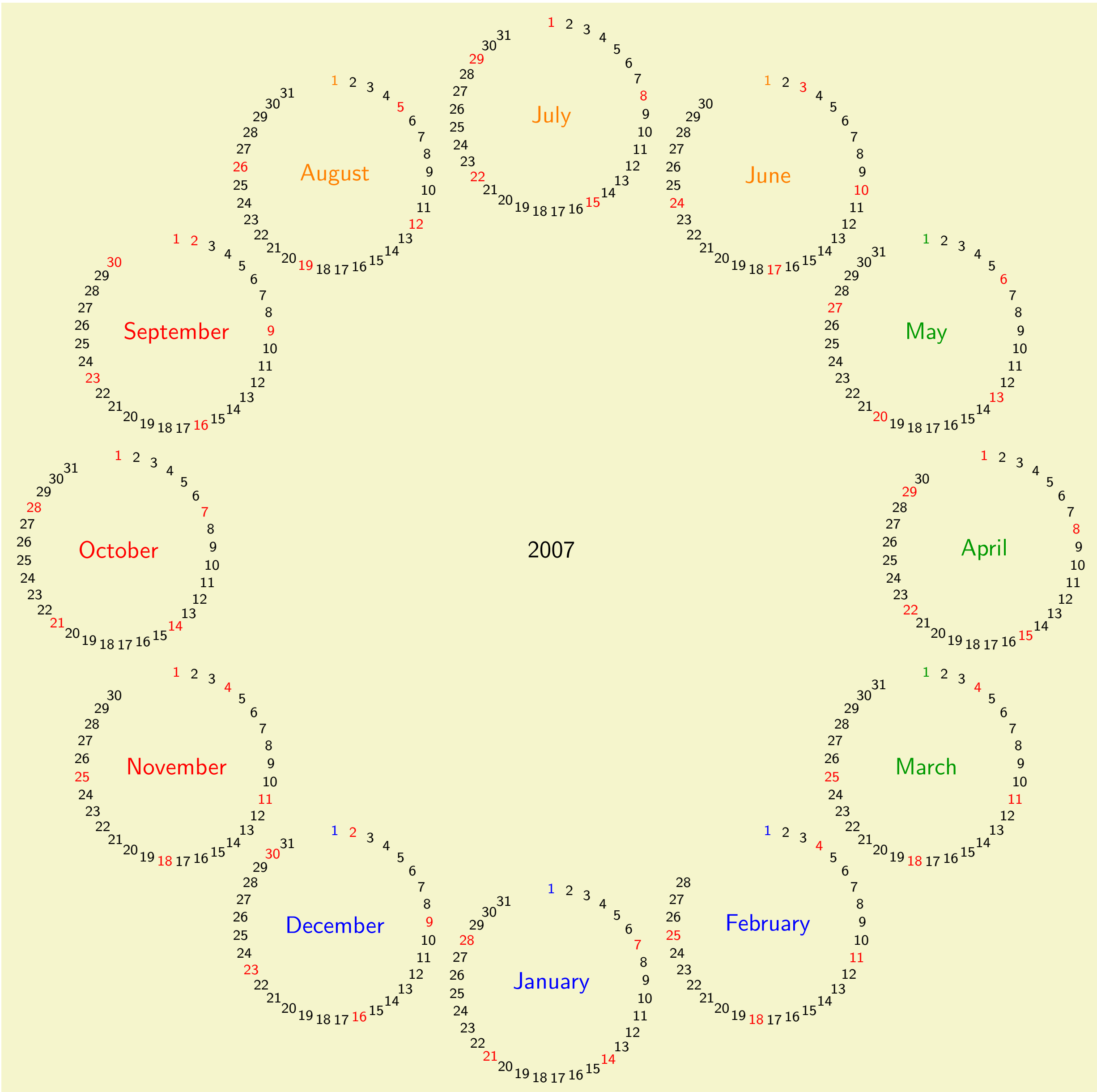
1 2 3
4 5 6 7 8 9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31

```
\begin{tikzpicture}
  \calendar
  [
    dates=2099-12-01 to 2100-01-last,
    week list,inner sep=2pt,month label above centered,
    month text=\%mt \%y0
  ]
  if (at most=2099-12-29) [nodes={strike out,draw}]
  if (weekend)            [black!50,nodes={draw=none}]
  ;
\end{tikzpicture}
```

The next calendar shows a deadline, which is 10 days in the future from the current date. The last three days before the deadline are in red, because we really should be done by then. All days on which we can no longer work on the project are crossed out.


```
\begin{tikzpicture}
  \calendar
  [
    dates=\year-\month-\day+-25 to \year-\month-\day+25,
    week list,inner sep=2pt,month label above centered,
    month text=\textit{\%mt \%y0}
  ]
  if (at least=\year-\month-\day) {}
    else [nodes={strike out,draw}]
  if (at most=\year-\month-\day+7)
    [green!50!black]
  if (between=\year-\month-\day+8 and \year-\month-\day+10)
    [red]
  if (Sunday)
    [gray,nodes={draw=none}]
  ;
\end{tikzpicture}
```

The following example is a futuristic calendar that is all about circles:



```
\sffamily

\colorlet{winter}{blue}
\colorlet{spring}{green!60!black}
\colorlet{summer}{orange}
\colorlet{fall}{red}

% A counter, since TikZ is not clever enough (yet) to handle
% arbitrary angle systems.
\newcount\mycount

\begin{tikzpicture}[transform shape]
  \tikzstyle{every day}=[anchor=mid,font=\fontsize{6}{6}\selectfont]
  \node{\normalsize\the\year};
  \foreach \month/\monthcolor in
    {1/winter,2/winter,3/spring,4/spring,5/spring,6/summer,
     7/summer,8/summer,9/fall,10/fall,11/fall,12/winter}
  {
    % Computer angle:
    \mycount=\month
    \advance\mycount by -1
    \multiply\mycount by 30
    \advance\mycount by -90

    % The actual calendar
    \calendar at (\the\mycount:6.4cm)
    [
      dates=\the\year-\month-01 to \the\year-\month-last,
    ]
    if (day of month=1) {\color{\monthcolor}\tikzmonthcode}
    if (Sunday) [red]
    if (all)
    {
      % Again, compute angle
      \mycount=1
      \advance\mycount by -\pgfcalendarcurrentday
      \multiply\mycount by 11
      \advance\mycount by 90
      \pgftransformshift{\pgfpointpolar{\mycount}{1.4cm}}
    };
  }
\end{tikzpicture}
```

Next, lets us have a whole year in a tight column:

01	1	2	3	4	5	6	7
	8	9	10	11	12	13	14
	15	16	17	18	19	20	21
	22	23	24	25	26	27	28
02	29	30	31	1	2	3	4
	5	6	7	8	9	10	11
	12	13	14	15	16	17	18
	19	20	21	22	23	24	25
03	26	27	28	1	2	3	4
	5	6	7	8	9	10	11
	12	13	14	15	16	17	18
	19	20	21	22	23	24	25
04	26	27	28	29	30	31	1
	2	3	4	5	6	7	8
	9	10	11	12	13	14	15
	16	17	18	19	20	21	22
	23	24	25	26	27	28	29
05	30	1	2	3	4	5	6
	7	8	9	10	11	12	13
	14	15	16	17	18	19	20
	21	22	23	24	25	26	27
06	28	29	30	31	1	2	3
	4	5	6	7	8	9	10
	11	12	13	14	15	16	17
	18	19	20	21	22	23	24
07	25	26	27	28	29	30	1
	2	3	4	5	6	7	8
	9	10	11	12	13	14	15
	16	17	18	19	20	21	22
	23	24	25	26	27	28	29
08	30	31	1	2	3	4	5
	6	7	8	9	10	11	12
	13	14	15	16	17	18	19
	20	21	22	23	24	25	26
09	27	28	29	30	31	1	2
	3	4	5	6	7	8	9
	10	11	12	13	14	15	16
	17	18	19	20	21	22	23
	24	25	26	27	28	29	30
10	1	2	3	4	5	6	7
	8	9	10	11	12	13	14
	15	16	17	18	19	20	21
	22	23	24	25	26	27	28
11	29	30	31	1	2	3	4
	5	6	7	8	9	10	11
	12	13	14	15	16	17	18
	19	20	21	22	23	24	25
12	26	27	28	29	30	1	2
	3	4	5	6	7	8	9
	10	11	12	13	14	15	16
	17	18	19	20	21	22	23
	24	25	26	27	28	29	30
	31						

```
\begin{tikzpicture}
  \small\sffamily
  \colorlet{darkgreen}{green!50!black}
  \calendar[dates=\year-01-01 to \year-12-31,week list,
            month label left,month yshift=0pt,
            month text=\textcolor{darkgreen}{\%m0}]
           if (Sunday) [black!50];
\end{tikzpicture}
```

22 Entity-Relationship Diagram Drawing Library

```
\usetikzlibrary{er} % LaTeX and plain TeX
\usetikzlibrary[er] % ConTeXt
```

This packages provides styles for drawing entity-relationship diagrams.

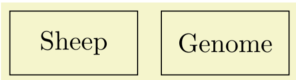
This library is intended to help you in creating E/R-diagrams. It defines only very little new styles, but using these style `entity` instead of saying `rectangle,draw` makes the code more expressive.

22.1 Entities

The package defines a simple style for drawing entities:

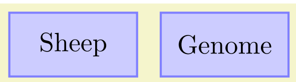
- `style=entity` This style is to be used with nodes that represent entity types. It causes the node's shape to be set to a rectangle that is drawn and whose minimum size and width are set to sensible values.

  Note that this style is called `entity` despite the fact that it is to be used for nodes representing entity *types* (the difference between an entity and an entity type is the same as the difference between an object and a class in object-oriented programming). If this bothers you, feel free to define a style `entity type` instead.




```
\begin{tikzpicture}[node distance=2cm]
  \node[entity] (sheep)                     {Sheep};
  \node[entity] (genome) [right of=sheep] {Genome};
\end{tikzpicture}
```

- `style=every entity` This stype is envoked by the style `entity`. To change the appearance of entities, you can change this style.

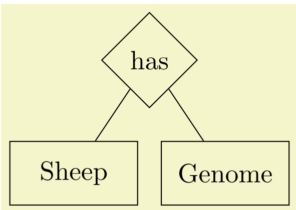



```
\begin{tikzpicture}[node distance=2cm]
  \tikzstyle{every entity}=[draw=blue!50,fill=blue!20,thick]
  \node[entity] (sheep)                     {Sheep};
  \node[entity] (genome) [right of=sheep] {Genome};
\end{tikzpicture}
```

22.2 Relationships

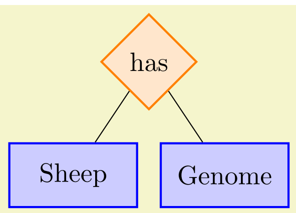
Relationships are drawn using styles that are very similar to the styles for entities.

- `style=relationship` This style works like `entity`, only it is to be used for relationships. Again, `relationships` are actually relationship types.




```
\begin{tikzpicture}
  \node[entity] (sheep)  at (0,0)   {Sheep};
  \node[entity] (genome) at (2,0)   {Genome};
  \node[relationship]    at (1,1.5) {has}
    edge (sheep)
    edge (genome);
\end{tikzpicture}
```

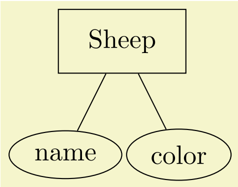
- `style=every relationship` works like `every entity`



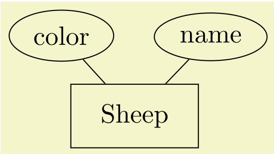

```
\begin{tikzpicture}
  \tikzstyle{every entity}=[fill=blue!20,draw=blue,thick]
  \tikzstyle{every relationship}=[fill=orange!20,draw=orange,thick,aspect=1.5]
  \node[entity] (sheep)  at (0,0)   {Sheep};
  \node[entity] (genome) at (2,0)   {Genome};
  \node[relationship]    at (1,1.5) {has}
    edge (sheep)
    edge (genome);
\end{tikzpicture}
```

22.3 Attributes

- `style=attribuate` This style is used to indicate that a node is an attribute. To connect an attribute to its entity, you can use, for example, the `child` command or the `pin` option.



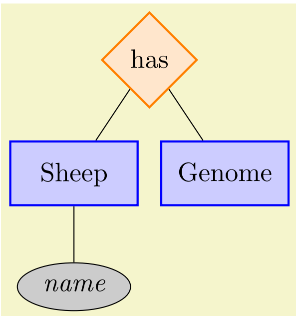

```
\begin{tikzpicture}
  \node[entity] (sheep)  {Sheep}
    child {node[attribute] {name}}
    child {node[attribute] {color}};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \tikzstyle{every pin edge}=[draw]
  \node[entity,pin={[attribute]60:name},pin={[attribute]120:color}] {Sheep};
\end{tikzpicture}
```

- `style=key attribute` This style is intended for key attributes. By default, the will cause the attribute to be typeset in italics. Typcially, underlining is used instead, but that looks ugly and it is difficult to implement in TEX.

- `style=every attribute` This style is used with every (key) attribute.




```
\begin{tikzpicture}[text depth=1pt]
  \tikzstyle{every attribute}=[fill=black!20,draw=black]
  \tikzstyle{every entity}=[fill=blue!20,draw=blue,thick]
  \tikzstyle{every relationship}=[fill=orange!20,draw=orange,thick,aspect=1.5]
  \node[entity] (sheep)  at (0,0)   {Sheep}
    child {node  [key attribute] {name}};
  \node[entity] (genome) at (2,0)   {Genome};
  \node[relationship]    at (1,1.5) {has}
    edge (sheep)
    edge (genome);
\end{tikzpicture}
```

23 Paper Folding Diagrams Library

`\usetikzlibrary{folding} % LaTeX and plain TeX`
`\usetikzlibrary[folding] % ConTeXt`

This library defines commands for creating paper folding diagrams. Currently, it just contains a single command for creating a single diagram, but that one is really useful for creating calendars for your (real) desktop.

`\tikzfoldingdodecahedron[⟨options⟩];`

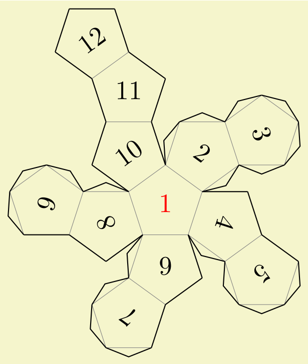
This command draws a folding diagram for a dodecahedron. The syntax is intended to remind of the `\path` command, but (currently) you must specify the ⟨*options*⟩ and nothing else may be specified between the command name and the closing semicolon.

The following ⟨*options*⟩ apply:

- `folding line length=`⟨*dimension*⟩ sets the length of the base line for folding. For the dodecahedron this is the length of all the sides of the pentagons.
- `face 1=`⟨*code*⟩ The ⟨*code*⟩ is executed for the first face of the dodecahedron. When it is executed, the coordinate system will have been shifted and rotated such that it lies at the middle of the first face of the dodecahedron.
- `face 2=`⟨*code*⟩ Same as `face 1`, but for the second face.
- `face 3=`⟨*code*⟩ Same as `face 1`, but for the third face.

  There are further similar options, ending with the following:
- `face 12=`⟨*code*⟩ Same as `face 1`, but for the last face.

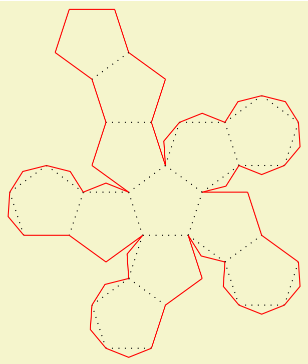
Here is a simple example:




```
\begin{tikzpicture}[transform shape]
  \tikzfoldingdodecahedron
  [folding line length=6mm,
  face 1={ \node[red] {1};},
  face 2={ \node      {2};},
  face 3={ \node      {3};},
  face 4={ \node      {4};},
  face 5={ \node      {5};},
  face 6={ \node      {6};},
  face 7={ \node      {7};},
  face 8={ \node      {8};},
  face 9={ \node      {9};},
  face 10={\node      {10};},
  face 11={\node      {11};},
  face 12={\node      {12};}];
\end{tikzpicture}
```

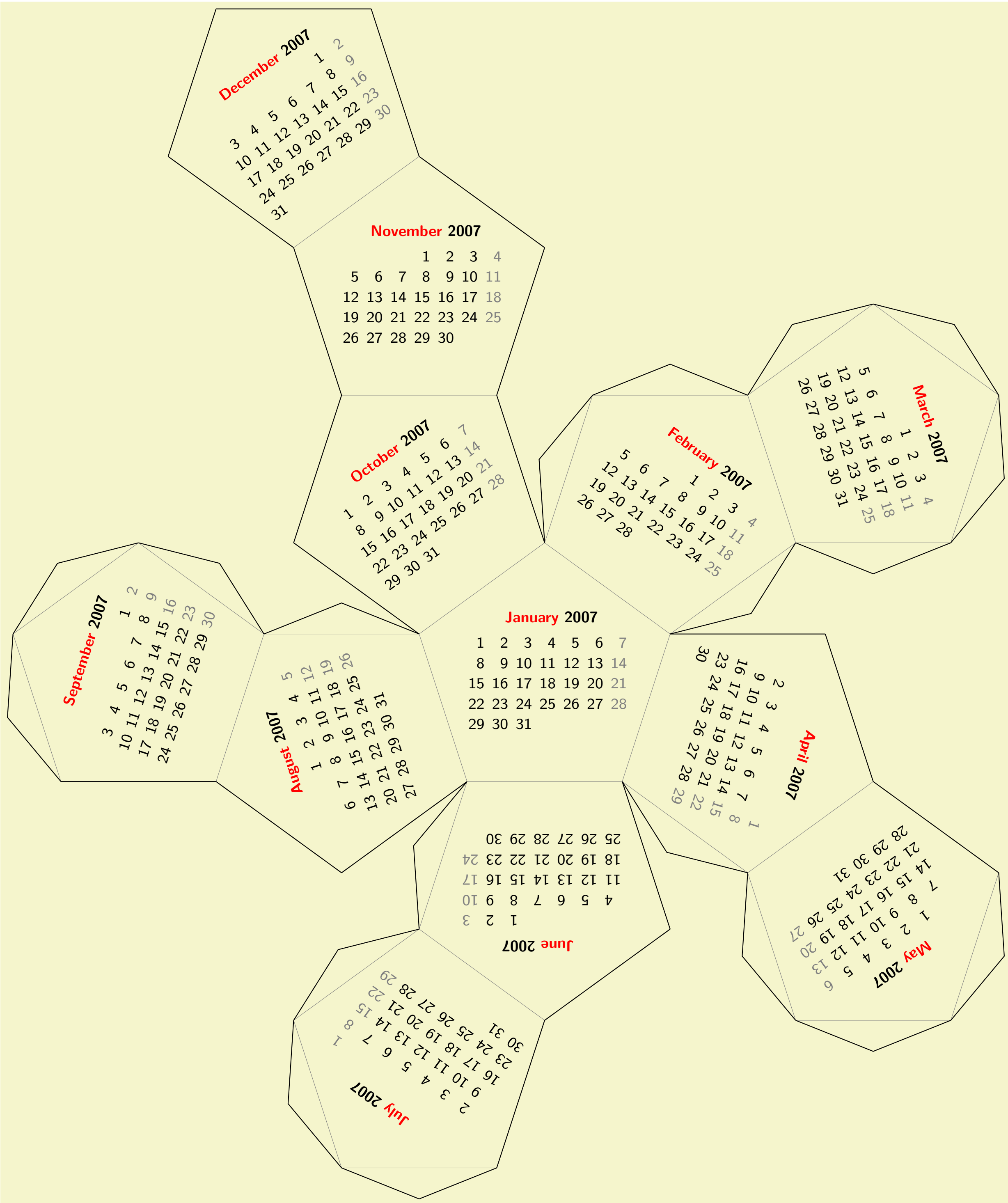
The appearance of the cut and folding lines can be influenced using the following styles:

- `style=every cut` Executed for every line that should be cut using scissors. Empty by default.
- `style=every fold` Executed for every line that should be folded. Equal to `help lines` by default.



```
\begin{tikzpicture}
  \tikzstyle{every cut}=[red]
  \tikzstyle{every fold}=[dotted]
  \tikzfoldingdodecahedron[folding line length=6mm];
\end{tikzpicture}
```

Here is a big example that produces a diagram for a calendar:



```
\sffamily\scriptsize
\begin{tikzpicture}[transform shape]
  \tikzstyle{every calendar}=
  [
    at={(-8ex,4ex)},
    week list,
    month label above centered,
    month text=\bfseries\textcolor{red}{\%mt} \%y0,
    if={(Sunday) [black!50]}
  ]
  \tikzfoldingdodecahedron
  [
    folding line length=2.5cm,
    face 1={ \calendar [dates=\the\year-01-01 to \the\year-01-last];},
    face 2={ \calendar [dates=\the\year-02-01 to \the\year-02-last];},
    face 3={ \calendar [dates=\the\year-03-01 to \the\year-03-last];},
    face 4={ \calendar [dates=\the\year-04-01 to \the\year-04-last];},
    face 5={ \calendar [dates=\the\year-05-01 to \the\year-05-last];},
    face 6={ \calendar [dates=\the\year-06-01 to \the\year-06-last];},
    face 7={ \calendar [dates=\the\year-07-01 to \the\year-07-last];},
    face 8={ \calendar [dates=\the\year-08-01 to \the\year-08-last];},
    face 9={ \calendar [dates=\the\year-09-01 to \the\year-09-last];},
    face 10={\calendar [dates=\the\year-10-01 to \the\year-10-last];},
    face 11={\calendar [dates=\the\year-11-01 to \the\year-11-last];},
    face 12={\calendar [dates=\the\year-12-01 to \the\year-12-last];}
  ];
\end{tikzpicture}
```

24 Matrix Library

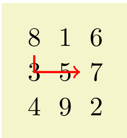
`\usetikzlibrary{matrix} % LaTeX and plain TeX`
`\usetikzlibrary[matrix] % ConTeXt`

This library packages defines additional styles and options for creating matrices.

24.1 Matrices of Nodes

A *matrix of nodes* is a Ti*k*Z matrix in which each cell contains a node. In this case it is bothersome having to write `\node{` at the beginning of each cell and `};` at the end of each cell. The following style simplifies typesetting such matrices.

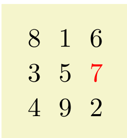
- `style=matrix of nodes` Conceptually, this style adds `\node{` at the beginning and `};` at the end of each cell and sets the `anchor` of the node to `base`. Furthermore, it adds the option `name` option to each node, where the name is set to ⟨*matrix name*⟩-⟨*row number*⟩-⟨*column number*⟩. For example, if the matrix has the name `my matrix`, then the node in the upper left cell will get the name `my matrix-1-1`.




```
\begin{tikzpicture}
  \matrix (magic) [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };

  \draw[thick,red,->] (magic-1-1) |- (magic-2-3);
\end{tikzpicture}
```

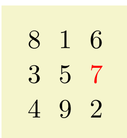
You may wish to add options to certain nodes in the matrix. This can be achieved in three ways.

1. You can modify, say, the `row 2 column 5` option to pass special options to this particular cell.




```
\begin{tikzpicture}
  \tikzstyle{row 2 column 3}=[red]
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

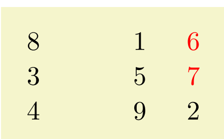
2. At the beginning of a cell, you can use a special syntax. If a cell starts with a vertical bar, then everything between this bar and the next bar is passed on to the `node` command.




```
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 & 1 &         6 \\
    3 & 5 & |[red]| 7 \\
    4 & 9 &         2 \\
  };
\end{tikzpicture}
```

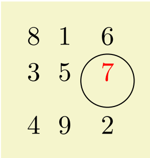
You can also use an option like `|[red] (seven)|` to give a different name to the node.

Note that the `&` character also takes an optional argument, which is an extra column skip.



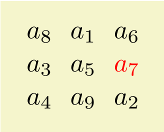

```
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 &[1cm] 1 &[3mm] |[red]| 6 \\
    3 &      5 &      |[red]| 7 \\
    4 &      9 &              2 \\
  };
\end{tikzpicture}
```

3. If your cell starts with a `\path` command or any command that expands to `\path`, which includes `\draw`, `\node`, `\fill` and other, the `\node{` startup code and the `};` code are suppressed. This means that for this particular cell you can provide a totally different contents.



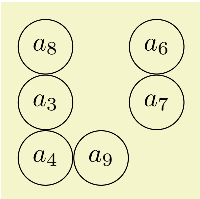

```
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & \node[red]{7}; \draw(0,0) circle(10pt);\\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

- `style=matrix of math nodes` This style is almost the same as the previous style, only `$` is added at the beginning and at the end of each node, so math mode will be switched on in all nodes.

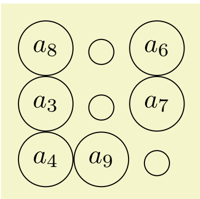


```
\begin{tikzpicture}
  \matrix [matrix of math nodes]
  {
    a_8 & a_1 &          a_6 \\
    a_3 & a_5 & |[red]| a_7 \\
    a_4 & a_9 &          a_2 \\
  };
\end{tikzpicture}
```

- `nodes in empty cells=`⟨*true or false*⟩ When set to `true`, a node (with an empty contents) is put in empty cells. Normally, empty cells are just, well, empty. The style can be used together with both a `matrix of nodes` and a `matrix of math nodes`.



```
\begin{tikzpicture}
  \matrix [matrix of math nodes,nodes={circle,draw}]
  {
    a_8 &     & a_6 \\
    a_3 &     & a_7 \\
    a_4 & a_9 &     \\
  };
\end{tikzpicture}
```

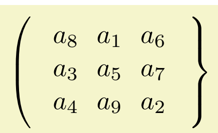


```
\begin{tikzpicture}
  \matrix [matrix of math nodes,nodes={circle,draw},nodes in empty cells]
  {
    a_8 &     & a_6 \\
    a_3 &     & a_7 \\
    a_4 & a_9 &     \\
  };
\end{tikzpicture}
```

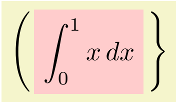
24.2 Delimiters

Delimiters are parantheses or braces to the left and right of a formula or a matrix. The matrix library offers options for adding such delimiters to a matrix. However, delimiters can actually be added to any node that has the standard anchors `north`, `south`, `north west` and so on. In particular, you can add delimiters to any `rectangle` box. They are implemented by "measuring the height" of the node and then adding a delimiter of the correct size to the left or right using some after node magic.

- `left delimiter=`⟨*delimiter*⟩ This option can be given to a any node that has the standard anchors `north`, `south` and so on. The ⟨*delimiter*⟩ can be any delimiter that is acceptable to TEX's `\left` command.

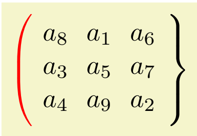


```
\begin{tikzpicture}
  \matrix [matrix of math nodes,left delimiter=(,right delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```



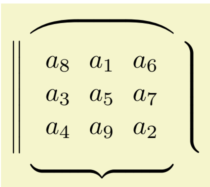
```
\begin{tikzpicture}
  \node [fill=red!20,left delimiter=(,right delimiter=\}]
    {$\displaystyle\int_0^1 x\,dx$};
\end{tikzpicture}
```

- `style=every delimiter` This style is executed for every delimiter. You can use it to shift or color delimiters or do whatever.
- `style=every left delimiter` This style is additionally executed for every left delimiter.



```
\begin{tikzpicture}
  \tikzstyle{every left delimiter}=[red,xshift=1ex]
  \tikzstyle{every right delimiter}=[xshift=-1ex]
  \matrix [matrix of math nodes,left delimiter=(,right delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```

- `right delimiter=`⟨*delimiter*⟩ Works as above.
- `style=every right delimiter` Works as above.
- `above delimiter=`⟨*delimiter*⟩ This option allows you to add a delimiter above the node. It is implementing by rotating a left delimiter.



```
\begin{tikzpicture}
  \matrix [matrix of math nodes,%
           left delimiter=\|,right delimiter=\rmoustache,%
           above delimiter=(,below delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```

- `style=every above delimiter` Works as above.
- `below delimiter=`⟨*delimiter*⟩ works as above.
- `style=every below delimiter` Works as above.

25 Mindmap Drawing Library

\usetikzlibrary{mindmap} % LaTeX and plain TeX
\usetikzlibrary[mindmap] % ConTeXt

This packages provides styles for drawing mindmap diagrams.

25.1 Overview

This library is intended to make the creation of mindmaps easier. A *mindmap* is a graphical representation of a concept together with related concepts and annotations. Mindmaps are, essentially, trees, possibly with a few extra edges added, but they are usually drawn in a special way: The root concept is placed in the middle of the page and is drawn as a huge circle, ellipse, or cloud. The related concepts then "leave" this root concept in branch-like tendrils.

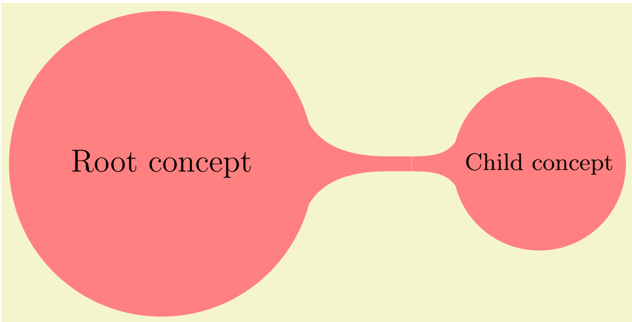
The mindmap library of Ti*k*Z produces mindmaps that look a bit different from the standard mindmaps: While the big root concept is still a circle, related concepts are also depicted as (smaller) circles. The related concepts are linked to the root concept via organic-looking connections. The overall effect is visually rather pleasing, but readers may not immediately think of a mindmap when they see a picture created with this library.

Although it is not strictly necessary, you will usually create mindmaps using Ti*k*Z's tree mechanism and some of the styles and macros of the package work best when used inside trees. However, it is still possible and sometimes necessary to treat parts of a mindmap as a graph with arbitrary edges and this is also possible.

25.2 The Mindmap Style

Every mindmap should be put in a scope or a picture where the `mindmap` style is used. This style installs some internal settings.

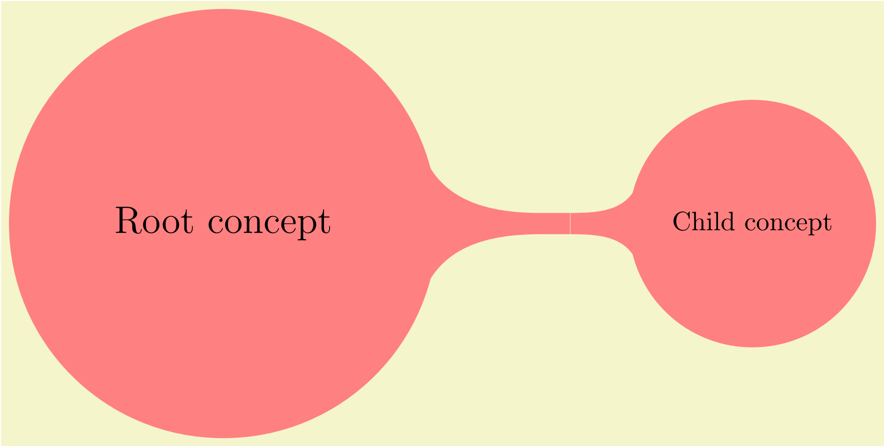
- `style=mindmap` Use this style with all pictures or at least scopes that contain a mindmap. It installs a whole bunch of settings that are useful for drawing mindmaps.




```
\tikz[mindmap,concept color=red!50]
  \node [concept] {Root concept}
    child[grow=right] {node[concept] {Child concept}};
```

The sizes of concepts are predefined in such a way that a medium-size mindmap will fit on an A4 page (more or less).

- `style=every mindmap` This style is included by the `mindmap` style. Change this style to add special settings to your mindmaps.




```
\tikz[large mindmap,concept color=red!50]
  \node [concept] {Root concept}
    child[grow=right] {node[concept] {Child concept}};
```

- `style=large mindmap` This style includes the `mindmap` style, but additionally changes the default size of concepts and of distances so that a medium-sized mindmap will fit on an A3 page (A3 pages are twice as large as A4 pages).
- `style=huge mindmap` This style causes conepts to be even bigger and it is best used with A2 paper and above.

25.3 Concepts Nodes

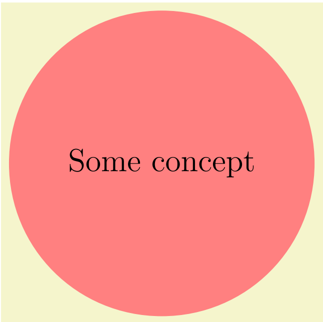
The basic entities of mindmaps are called *concepts* in Ti*k*Z. A concept is a node of style `concept` and it must be circular for some of the connection macros to work.

25.3.1 Isolated Concepts

The following styles influence how isolated concepts are rendered:

- `style=concept` This style should be used with all nodes that are concepts, although some styles like `extra concept` install this style automatically.

  Bascially, this style makes the concept node circular and installs a uniform color called `concept color`, see below. Additionally, the style `every concept` is called.

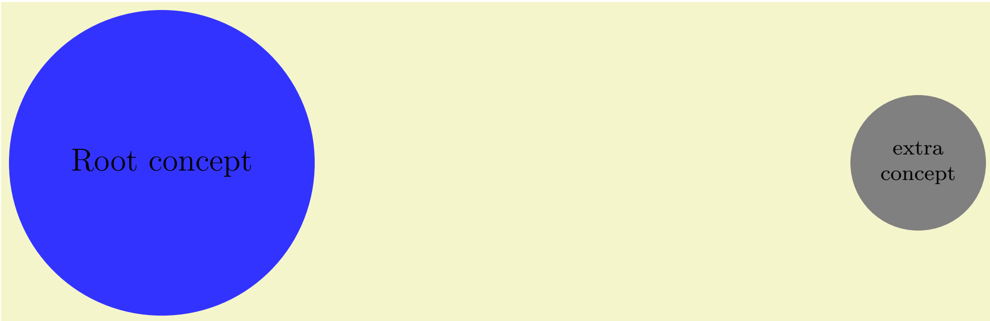



```
\tikz[mindmap,concept color=red!50] \node [concept] {Some concept};
```

- `style=every concept` In order to change the appearance of concept nodes, you should change this style. Note, however, that the color of a concept should be uniform for some of the connection bar stuff to work, so you should not change the color or the draw/fill state of concepts using this option. It is mostly useful for changing the text color and font.
- `concept color=`⟨*color*⟩ This option tells Ti*k*Z which color should be used for filling and stroking concepts. The difference between this option and just setting `every concept` to the desired color is that this option allows Ti*k*Z to keep track of the colors used for concepts. This is important when

you *change* the color between two connected concepts. In this case, TikZ can automatically create a shading that provides a smooth transition between the old and the new concept color; we will come back to this in the next section.

- `style=extra concept` This style is intended for concepts that are not part of the "mindmap tree," but stand beside it. Typically, they will have a subdued color are be smaller. In order to have these concepts appear in a uniform way and in order to indicate in the code that these concepts are extra, you can use this style.



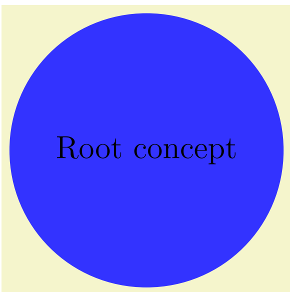

```
\begin{tikzpicture}[mindmap,concept color=blue!80]
  \node [concept]                {Root concept};
  \node [extra concept] at (10,0) {extra concept};
\end{tikzpicture}
```

- `style=every extra concept` Change this style to change the appearance of extra concepts.

25.3.2 Concepts in Trees

As pointed out earlier, TikZ assumes that your mindmap is build using the `child` facilities of TikZ. There are numerous options that influence how concepts are rendered at the different levels of a tree.

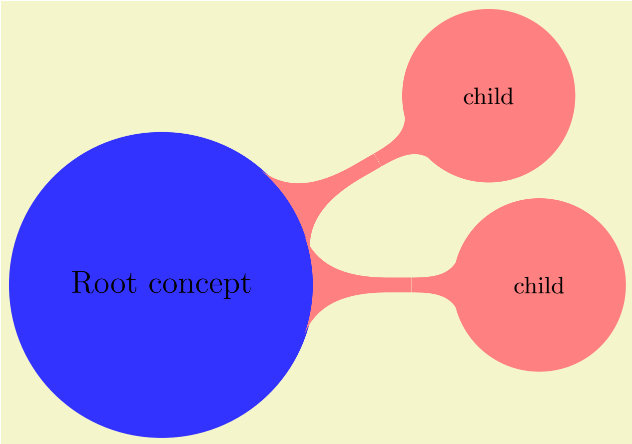
- `style=root concept` This style is used for the roots of mindmap trees. More precisely, this style is included by the `mindmap` style. Thus, by adding something to this, you can change how the root of a mindmap will be rendered.




```
\tikzstyle{root concept}+=[concept color=blue!80,minimum size=3.5cm]
\tikz[mindmap] \node [concept] {Root concept};
```

Note that styles like `large mindmap` redefine these styles, so you should add something to this style only inside the picture.

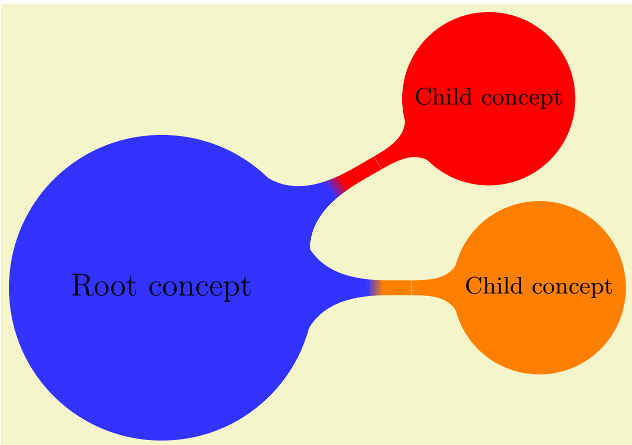
- `style=level 1 concept` The `mindmap` style adds this style to the `level 1` style. This means that the first level children of a mindmap tree will use this style.




```
\tikzstyle{root concept}+=[concept color=blue!80]
\tikzstyle{level 1 concept}+=[concept color=red!50]
\tikz[mindmap]
  \node [concept] {Root concept}
    child[grow=30] {node[concept] {child}}
    child[grow=0 ] {node[concept] {child}};
```

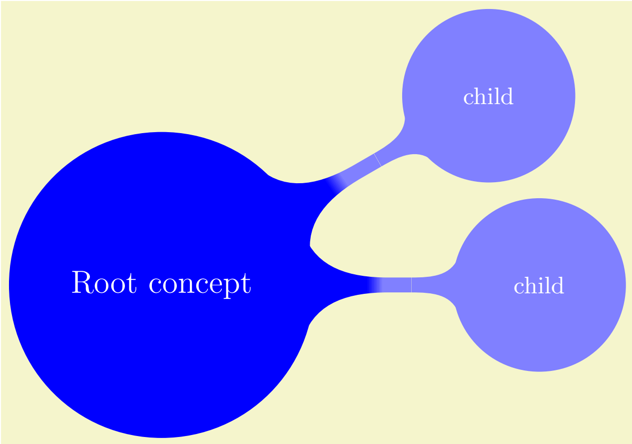
- `style=level 2 concept` works like `level 1 concept`, only for second level children.
- `style=level 3 concept` works like `level 1 concept`.
- `style=level 4 concept` works like `level 1 concept`. Note that there are not fifth and higher level styles, you need to modify `level 5` directly in such cases.
- `concept color=`⟨*color*⟩ We saw already that this option is used to change the color of concepts. We now have a look at its effect when used on child nodes of a concept. Normally, this option simply changes the color of the children. However, when the option is given as an option to the `child` operation (and not to the `node` operation and also not as an option to all children via the `level 1` style), Ti*k*Z will smoothly change the concept color from the parent's color to the color of the child concept.

  Here is an example:




```
\tikz[mindmap,concept color=blue!80]
  \node [concept] {Root concept}
    child[concept color=red,grow=30] {node[concept] {Child concept}}
    child[concept color=orange,grow=0]  {node[concept] {Child concept}};
```

In order to have all children of a certain level have a different concept color, a tiny bit of magic is needed:




```
\tikzstyle{root concept}+=[concept color=blue]
\tikzstyle{level 1 concept}+=[set style={{every child}=[concept color=blue!50]}]
\tikz[mindmap,text=white]
  \node [concept] {Root concept}
    child[grow=30] {node[concept] {child}}
    child[grow=0 ] {node[concept] {child}};
```

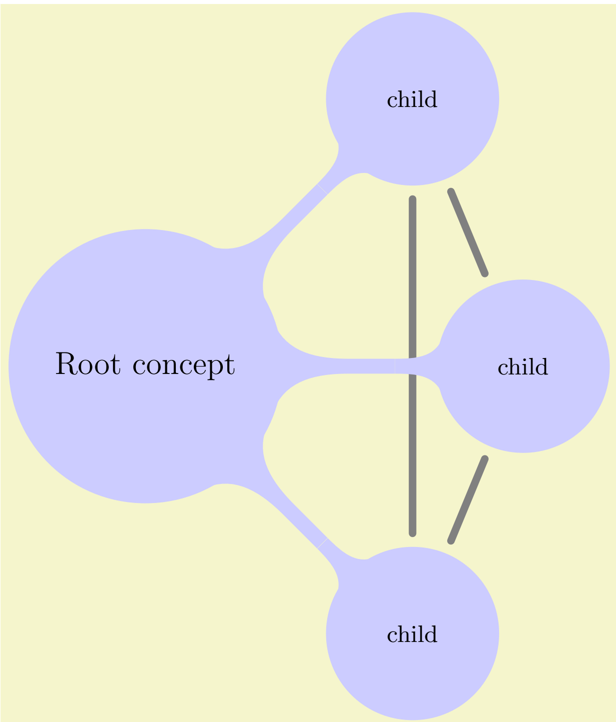
25.4 Connecting Concepts

25.4.1 Simple Connections

The easiest way to connect two concepts is to draw a line between them. In order to give such lines a consistent appearance, it is recommendable to use the following style when drawing such lines:

- `style=concept connection` This style can be used for lines between two concepts. Feel free to redefine this style.

A problem arises when you need to connect concepts after the main mindmap has been drawn. In this case you will want the connection lines to lie *behind* the main mindmap. However, you can draw the lines only after the coordinates of the concepts have been determined. In this case you should place the connecting lines on a background layer as in the following example:



```
\tikzstyle{root concept}+=[concept color=blue!20,minimum size=2cm]
\tikzstyle{level 1 concept}+=[sibling angle=45]
\begin{tikzpicture}[mindmap]
  \node [concept] {Root concept}
    [clockwise from=45]
    child { node[concept] (c1) {child}}
    child { node[concept] (c2) {child}}
    child { node[concept] (c3) {child}};
  \begin{pgfonlayer}{background}
    \draw [concept connection]  (c1) edge (c2)
                                     edge (c3)
                                (c2) edge (c3);
  \end{pgfonlayer}
\end{tikzpicture}
```

25.4.2 The Circle Connection Bar Snake

Instead of a simple line between two concepts, you can also add a bar between the two nodes that has slightly organic ends. These bars are also used by default as the edges from parents in the mindmap tree.

For the drawing of the bars a special snake is used, which is defined in the mindmap library:

Snake `circle connection bar`

This snake can be used to connect two circles. The start of the snake should lie on the border of the first circle, the end should lie on the border of the second circle. The following two macros should be initialized with the sizes of the circles:

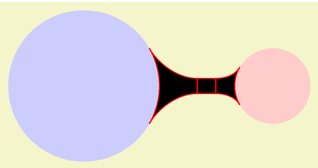
- `\pgfsnakecirclestartradius`
- `\pgfsnakecircleendradius`

Furthermore, the following two macros influence the snake:

- `\pgfsnakesegmentamplitude`
- `\pgfsnakesegmentangle`

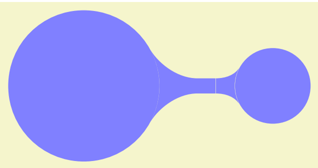
The snake itself will be a path that starts on the border of the first circle at the specified angle relative to the line connecting the centers of the circles. The path then changes into a rectangle whose thickness is given by the amplitude. Finally, the path ends with the same angles on the second circle.

Here is an example that should make this clearer:



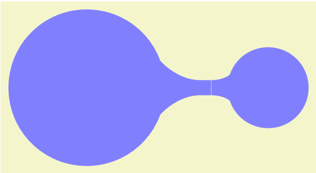
```
\begin{tikzpicture}
  \fill[blue!20] (0,0) circle   (1cm);
  \fill[red!20]  (2.5,0) circle (.5cm);

  \def\pgfsnakecirclestartradius{1cm}
  \def\pgfsnakecircleendradius{.5cm}
  \def\pgfsnakesegmentamplitude{2mm}
  \def\pgfsnakesegmentangle{30}

  \filldraw [draw=red,fill=black,snake=circle connection bar] (1,0) -- (2,0);
\end{tikzpicture}
```

As can be seen, the snake consists of three parts and is not really useful for drawing. However, if you fill the snake only and if you use the same color as for the circles, the result is better.



```
\begin{tikzpicture}[blue!50]
  \fill (0,0) circle   (1cm);
  \fill  (2.5,0) circle (.5cm);

  \def\pgfsnakecirclestartradius{1cm}
  \def\pgfsnakecircleendradius{.5cm}
  \def\pgfsnakesegmentamplitude{2mm}
  \def\pgfsnakesegmentangle{30}

  \fill [snake=circle connection bar] (1,0) -- (2,0);
\end{tikzpicture}
```

In the above example you may notice the small white line between the circles and the snake. This is due to rounding errors. Unfortunately, for larger distances, there errors can accumulate quite strongly, especially since Ti*k*Z and TEX are not very good at computing square roots. For this reason, it is a good idea to make the circles slightly larger to cover up such problems. When using nodes of shape `circle`, you can just add the `draw` option with a `line width` or one or two points (for very large distances you may need line width up to 4pt).



```
\begin{tikzpicture}[blue!50]
  \fill (0,0) circle   (1cm+1pt);
  \fill  (2.4,0) circle (.5cm+1pt);

  \def\pgfsnakecirclestartradius{1cm}
  \def\pgfsnakecircleendradius{.5cm}
  \def\pgfsnakesegmentamplitude{2mm}
  \def\pgfsnakesegmentangle{30}

  \fill [snake=circle connection bar] (1,0) -- (1.9,0);
\end{tikzpicture}
```

Note the slightly strange `outer sep=0pt`. This is needed so that the snakes being on the border of the filled circle, not on the border of the stroked circle (which is slightly larger and this slightly larger size is exactly what we wish to use to cover up the rounding errors).

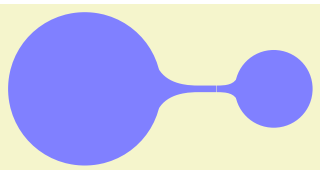
25.4.3 The Circle Connection Bar To-Path

The circle connection bar snake is a bit complicated to use. Especially specifying the radii is quite bothersome (the amplitude and the angle can be set once and for all). For this reason, the mindmap library defines a special to-path, that performs the necessary computations for you.

- `style=circle connection bar` This style installs a rather involved to-path. Unlike normal to-paths, this path requires that the start and the target of the to-path are named nodes of shape `circle` – if this is not the case, this path will produce errors.

  Assuming that the start and the target are circles, the to-path will first compute the radii of these circles (by measuring the distance from the `center` anchor to some anchor on the border) and will set the `\pgfsnakecirlce...` macros accordingly. Next, the `fill` option is set to the `concept color` while `draw=none` is set. The snake is set to `circle conncetion bar`. Finally, the following style is included:

  – `style=every circle connection bar` Redefine this sytle to change the appearance of circle connection bar to-paths.

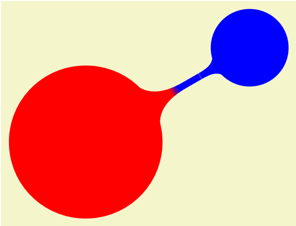


```
\begin{tikzpicture}[concept color=blue!50,blue!50,outer sep=0pt]
  \node (n1) at (0,0)   [circle,minimum size=2cm,fill,draw,thick] {};
  \node (n2) at (2.5,0) [circle,minimum size=1cm,fill,draw,thick] {};

  \path (n1) to[circle connection bar] (n2);
\end{tikzpicture}
```

  Note that it is not a good idea to have more than one `to` operation together this the option `circle connection bar` in a single `\path`. Use the `edge` operation, instead, for creating multiple connections and this operation creates a new scope for each edge.

In a mindmap we sometimes want colors to change from one concept color to another. Then, the connection bar should, ideally, consist of a smooth transition between these two colors. Getting this right using shadings is a bit tricky if you try this "by hand," so the mindmap library provides a special option for facilitating this procedure.

- `circle connection bar switch color=from (`⟨*first color*⟩`) to (`⟨*second color*⟩`)` This style works similarly to the `circle connection bar`. The only difference is that instead of filling the path with a single color a shading is used.



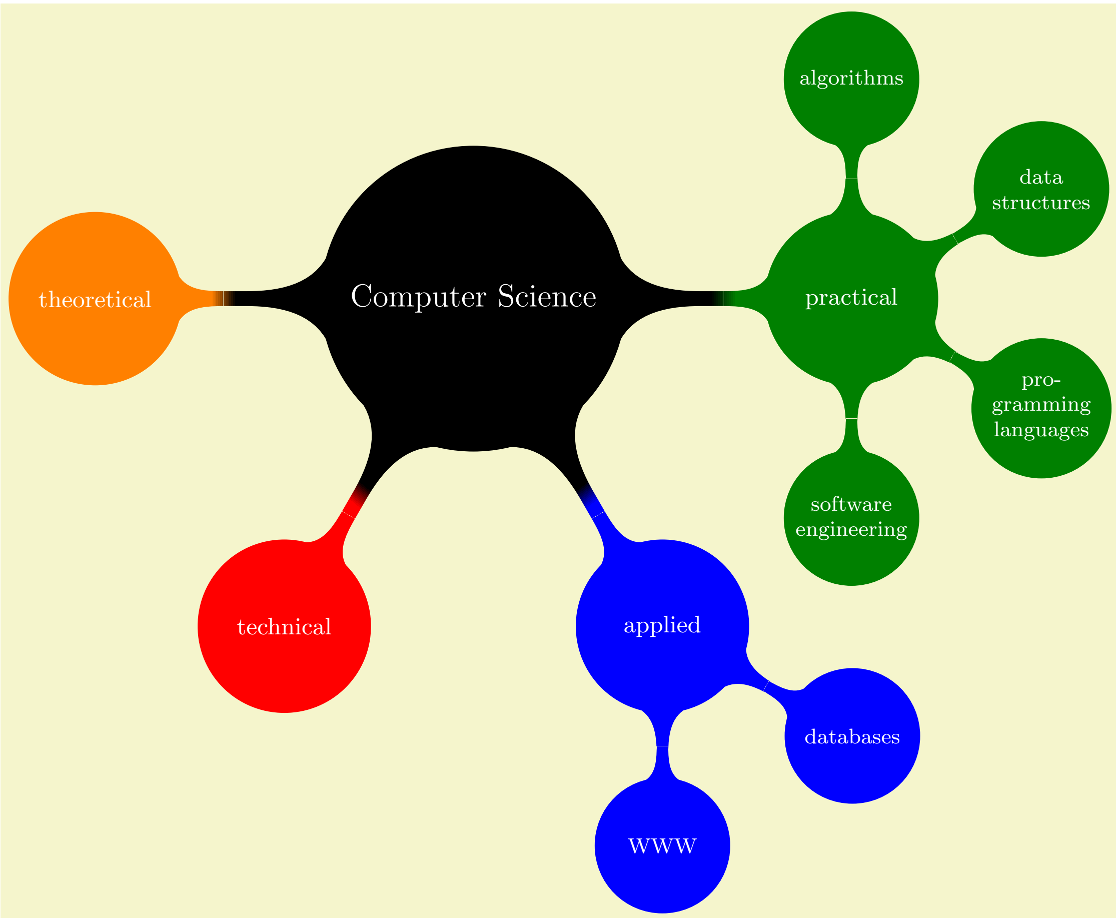
```
\begin{tikzpicture}[outer sep=0pt]
  \node (n1) at (0,0)    [circle,minimum size=2cm,fill,draw,thick,red] {};
  \node (n2) at (30:2.5) [circle,minimum size=1cm,fill,draw,thick,blue] {};

  \path (n1) to[circle connection bar switch color=from (red) to (blue)] (n2);
\end{tikzpicture}
```

25.4.4 Tree Edges

Most of the time, concepts in a mindmap are connected automatically when the mindmap is build as a tree. The reason is that the `mindmap` installs a `circle connection bar` path as the edge from parent path. Also, the `mindmap` option takes care of things like setting the correct `draw` and `outer sep` settings and some other stuff.

In detail, the `mindmap` option sets the `edge from parent path` to a path that uses the to-path `circle connection bar` to connect the parent node and the child node. The `concept color` option (locally) changes this by using `circle connection bar switch color` instead with the from-color set to the old (parent's) concept color and the to-color set to the new (child's) concept color. This menas that when you provide the `concept color` option to a `child` command, the color will change from the parent's concept color to the specified color.

Here is an example of a tree build in this way:

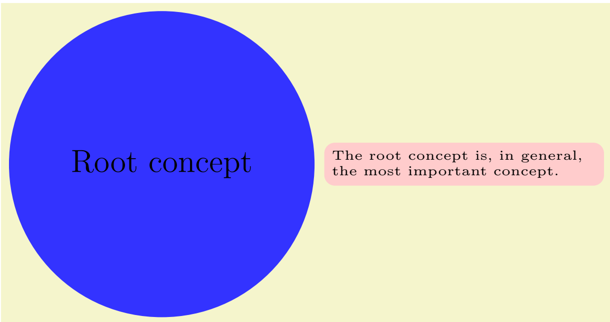


```
\begin{tikzpicture}
  \path[mindmap,concept color=black,text=white]
    node[concept] {Computer Science}
    [clockwise from=0]
    child[concept color=green!50!black] {
      node[concept] {practical}
      [clockwise from=90]
      child { node[concept] {algorithms} }
      child { node[concept] {data structures} }
      child { node[concept] {pro\-gramming languages} }
      child { node[concept] {software engineer\-ing} }
    }
    child[concept color=blue] {
      node[concept] {applied}
      [clockwise from=-30]
      child { node[concept] {databases} }
      child { node[concept] {WWW} }
    }
    child[concept color=red] { node[concept] {technical} }
    child[concept color=orange] { node[concept] {theoretical} };
\end{tikzpicture}
```

25.5 Adding Annotations

An *annotation* is some text outside a mindmap that, unlike an extra concept, simply explains something in the mindmap. The following style is mainly intended to help readers of the code see that a node in an annotation node.

- `style=annotation` This style indicates that a node is an annotation node. It includes the style `every annotation`, which allows you to change this style in a convenient fashion.

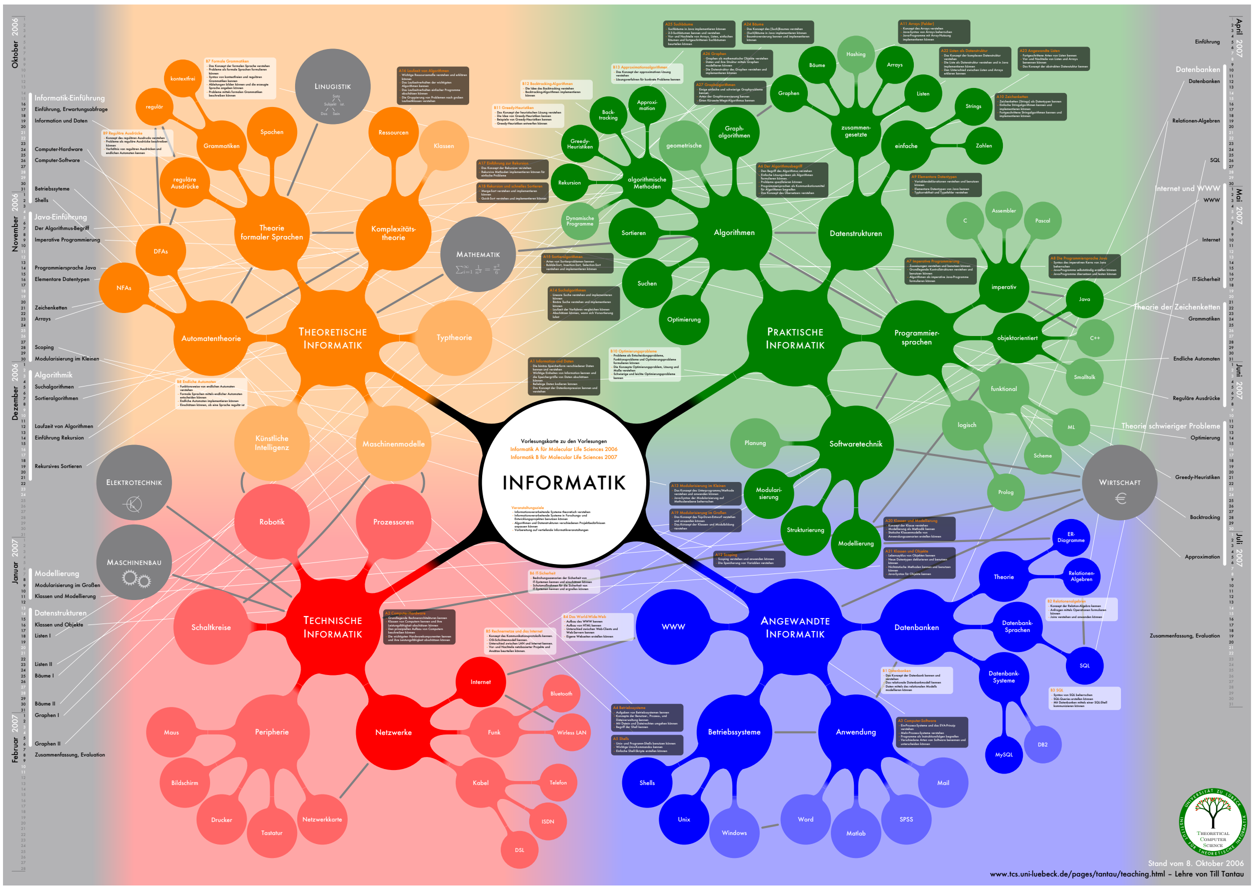


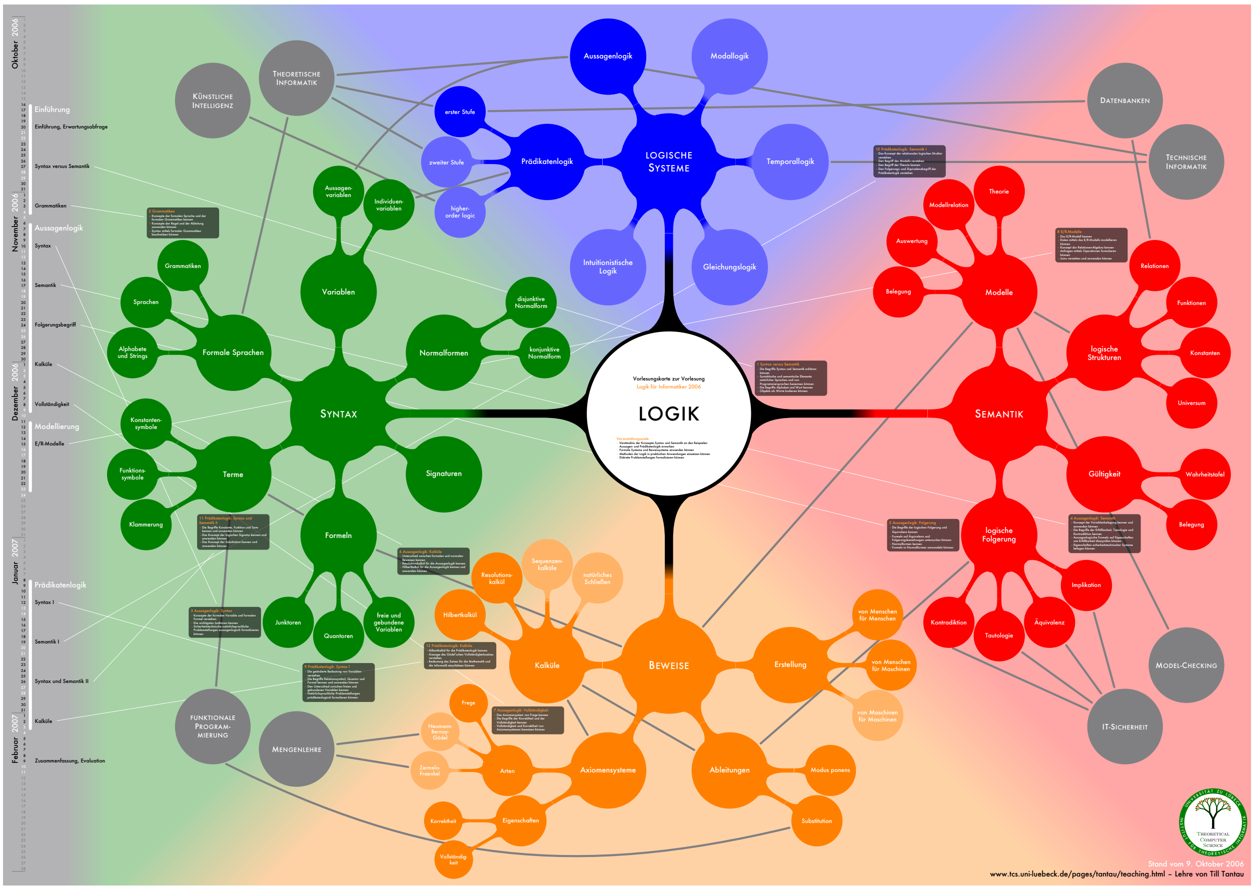

```
\tikzstyle{every annotation}=[fill=red!20]
\begin{tikzpicture}[mindmap,concept color=blue!80]
  \node [concept] (root)  {Root concept};

  \node [annotation,right] at (root.east)
  {The root concept is, in general, the most important concept.};
\end{tikzpicture}
```

- `style=every annotation` This style is included by `annotation`.

25.6 Examples

The following pictures show examples of more complicated mindmaps that have been created using the mindmap library. Note that these mindmaps should be printed on A1 paper.

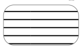
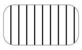
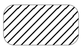
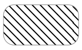
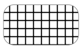
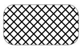
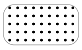
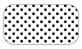
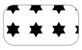
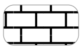

Vorlesungskarte zu den Vorlesungen
Informatik A für Molecular Life Sciences 2006
Informatik B für Molecular Life Sciences 2007
INFORMATIK
Theoretische Informatik
Praktische Informatik
Technische Informatik
Angewandte Informatik
Linguistik
Mathematik
Elektrotechnik
Maschinenbau
Wirtschaft
Theorie formaler Sprachen
Komplexitätstheorie
Automatentheorie
Typtheorie
Künstliche Intelligenz
Maschinenmodelle
Grammatiken
Spachen
kontextfrei
regulär
reguläre Ausdrücke
DFAs
NFAs
Ressourcen
Klassen
Algorithmen
Datenstrukturen
Programmiersprachen
Softwaretechnik
algorithmische Methoden
Graphalgorithmen
geometrische
Approximation
Backtracking
Greedy-Heuristiken
Rekursion
Dynamische Programme
Sortieren
Suchen
Optimierung
Graphen
Hashing
Bäume
Arrays
Listen
zusammengesetzte
einfache
Strings
Zahlen
imperativ
objektorientiert
funktional
logisch
C
Assembler
Pascal
Java
C++
Smalltalk
ML
Scheme
Prolog
Planung
Modularisierung
Strukturierung
Modellierung
Robotik
Prozessoren
Schaltkreise
Peripherie
Netzwerke
Internet
Bluetooth
Funk
Wireless LAN
Kabel
Telefon
ISDN
DSL
Maus
Bildschirm
Drucker
Tastatur
Netzwerkkarte
WWW
Datenbanken
Betriebssysteme
Anwendung
Theorie
ER-Diagramme
Relationen-Algebren
Datenbank-Sprachen
Datenbank-Systeme
SQL
MySQL
DB2
Shells
Unix
Windows
Word
Matlab
SPSS
Mail
Stand vom 8. Oktober 2006
www.tcs.uni-luebeck.de/pages/tantau/teaching.html – Lehre von Till Tantau


Vorlesungskarte zur Vorlesung
Logik für Informatiker 2006
LOGIK
Syntax
Semantik
Beweise
Logische Systeme
Künstliche Intelligenz
Theoretische Informatik
Datenbanken
Technische Informatik
Model-Checking
IT-Sicherheit
Funktionale Programmierung
Mengenlehre
Aussagenlogik
Modallogik
Prädikatenlogik
Temporallogik
Intuitionistische Logik
Gleichungslogik
erster Stufe
zweiter Stufe
higher-order logic
Aussagenvariablen
Individuenvariablen
Variablen
Grammatiken
Sprachen
Alphabete und Strings
Formale Sprachen
Normalformen
disjunktive Normalform
konjunktive Normalform
Konstantensymbole
Funktionssymbole
Klammerung
Terme
Signaturen
Formeln
Junktoren
Quantoren
freie und gebundene Variablen
Modellrelation
Theorie
Auswertung
Belegung
Modelle
Relationen
Funktionen
logische Strukturen
Konstanten
Universum
Gültigkeit
Wahrheitstafel
logische Folgerung
Implikation
Kontradiktion
Tautologie
Äquivalenz
Resolutionskalkül
Sequenzenkalkül
natürliches Schließen
Hilbertkalkül
Kalküle
Erstellung
von Menschen für Menschen
von Menschen für Maschinen
von Maschinen für Maschinen
Frege
Axiomensysteme
Ableitungen
Modus ponens
Arten
Korrektheit
Eigenschaften
Vollständigkeit
Substitution
Stand vom 9. Oktober 2006
www.tcs.uni-luebeck.de/pages/tantau/teaching.html – Lehre von Till Tantau

26 Pattern Library

```
\usepgflibrary{patterns}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[patterns]  % ConTeXt and pure pgf
\usetikzlibrary{patterns} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[patterns] % ConTeXt when using TikZ
```

The package defines patterns for filling areas.

Pattern name	*Example*
`horizontal lines`	
`vertical lines`	
`north east lines`	
`north west lines`	
`grid`	
`crosshatch`	
`dots`	
`crosshatch dots`	
`fivepointed stars`	
`sixpointed stars`	
`bricks`	

27 Petri-Net Drawing Library

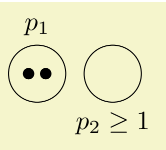
```
\usetikzlibrary{petri} % LaTeX and plain TeX
\usetikzlibrary[petri] % ConTeXt
```

This packages provides shapes and styles for drawing Petri nets.

27.1 Places

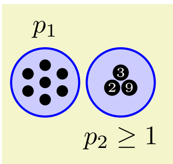
The package defines a style for drawing places of Petri nets.

- `style=place` This style indicates that a node is a place of a Petri net. Usually, the text of the node should be empty since places do not contain any text. You should use the `label` option to add text outside the node like its name or its capacity. You should use the `tokens` options, explained in Section 27.3, to add tokens inside the place.




```
\begin{tikzpicture}
  \node[place,label=above:$p_1$,tokens=2]        (p1) {};
  \node[place,label=below:$p_2\ge1$,right of=p1] (p2) {};
\end{tikzpicture}
```

- `style=every place` This stype is envoked by the style `place`. To change the appearance of places, you can change this style.



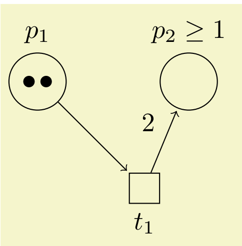

```
\tikzstyle{every place}=[draw=blue,fill=blue!20,thick,minimum size=9mm]
\begin{tikzpicture}
  \node[place,tokens=7,label=above:$p_1$]  (p1) {};
  \node[place,structured tokens={3,2,9},
        label=below:$p_2\ge1$,right of=p1] (p2) {};
\end{tikzpicture}
```

27.2 Transitions

Transitions are also nodes. They should be drawn using the following style:

- `style=transition` This style indicates that a node is a transition. As for places, the text of a transition should be empty and the `label` option should be used for adding labels.

  To connect a transition to places, you can use the `edge` command as in the following example:




```
\begin{tikzpicture}[node distance=2cm]
  \node[place,tokens=2,label=above:$p_1$]        (p1) {};
  \node[place,label=above:$p_2\ge1$,right of=p1] (p2) {};

  \node[transition,below right of=p1,label=below:$t_1$] {}
    edge[pre]                  (p1)
    edge[post] node[auto] {2} (p2);
\end{tikzpicture}
```

- `style=every transition` This style is envoked by the style `transition`.

- `style=pre` This style can be used with paths leading *from* a transition *to* a place to indicate that the place is in the pre-set of the transition. By default, this style is `<-,shorten <=1pt`, but feel free to redefine it.

- `style=post` This style is also used with paths leading *from* a transition *to* a place, but this time the place is in the post-set of the transition. Again, feel free to redefine it.

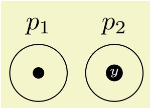
- `style=pre and post` This style is to be used to indicate that a place is both in the pre- and post-set of a transition.

27.3 Tokens

Interestingly, the most complicated aspect of drawing Petri nets in Ti*k*Z turns out to be the placement of tokens.

Let us start with a single token. They are also nodes and there is a simple style for typesetting them.

- `style=token` This style indicates that a node is a token. By default, this causes the node to be a small black circle. Unlike places and transitions, it *does* make sense to provide text for the token node. Such text will be typeset in a tiny font and in white on black (naturally, you can easily change this by setting the style `every token`).

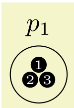



```
\begin{tikzpicture}
  \node[place,label=above:$p_1$]            (p1) {};
  \node[token] at (p1) {};

  \node[place,label=above:$p_2$,right of=p1] (p2) {};
  \node[token] at (p2) {$y$};
\end{tikzpicture}
```

In the above example, it is bothersome that we need an extra command for the token node. Worse, when we have *two* tokens on a node, it is difficult to place both nodes inside the node without overlap.

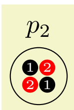
The Petri net library offers a solution to this problem: The `children are tokens` style.

- `style=children are tokens` The idea behind this style is to use trees mechanism for placing tokens. Every token lying on a place is treated as a child of the node. Normally this would have the effect that the tokens are placed below the place and they would be connected to the place by an edge. The `children are tokens` style, however, redefines the growth function of trees such that it places the children next to each other inside (or, rather, on top) of the place node. Additionally, the edge from the parent node is not drawn.



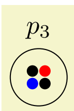

```
\begin{tikzpicture}
  \node[place,label=above:$p_1$] {}
  [children are tokens]
  child {node [token] {1}}
  child {node [token] {2}}
  child {node [token] {3}};
\end{tikzpicture}
```

In detail, what happens is the following: Tree growth functions tell Ti*k*Z where it should place the children of nodes. These functions get passed the number of children that a node has an the number of the child that should be placed. The special tree growth function for tokens has a special mapping for each possible number of children up to nine children. This mapping decides for each child where it should be placed on top of the place. For example, a single child is placed directly on top of the place. Two children are placed next to each other, separated by the `token distance`. Three children are placed in a triangle whose side lengths are `token distance`; and so on up to nine tokens. If you wish to place more than nice tokens on a place, you will have to write your own placement code.




```
\begin{tikzpicture}
  \node[place,label=above:$p_2$] {}
  [children are tokens]
  child {node [token] {1}}
  child {node [token,fill=red] {2}}
  child {node [token,fill=red] {2}}
  child {node [token] {1}};
\end{tikzpicture}
```

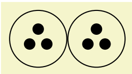
- `token distance=`⟨*distance*⟩ This specifies the distance between the centers of the tokens in the arrangements of the option `children are tokens`.




```
\begin{tikzpicture}
  \node[place,label=above:$p_3$] {}
  [children are tokens,token distance=1.1ex]
  child {node [token] {}}
  child {node [token,red] {}}
  child {node [token,blue] {}}
  child {node [token] {}};
\end{tikzpicture}
```

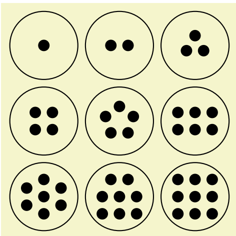
The `children are tokens` options gives you a lot of flexibility, but it is a bit cumbersome to use. For this reason there are some options that help in standard situations. They all use `children are tokens` internally, so any change to, say, the `every tokens` style will affect how these options depict tokens.

- `tokens`=⟨*number*⟩ This option is given to a `place` node, not to a `token` node. The effect of this option is to add ⟨*number*⟩ many child nodes to the place, each having the style `token`. Thus, the following two pieces of codes have the same effect:



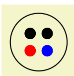
```
\tikz
  \node[place] {}
  [children are tokens]
  child {node [token] {}}
  child {node [token] {}}
  child {node [token] {}};
\tikz
  \node[place,tokens=3] {};
```

It is legal to say `tokens=0`, no tokens are drawn in this case. This option does not handle ten or more tokens correctly. If you need this many tokens, you will have to program your own code.



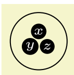
```
\begin{tikzpicture}
  \tikzstyle{every place}=[minimum size=9mm]

  \foreach \x/\y/\tokennumber in {0/2/1,1/2/2,2/2/3,
                                  0/1/4,1/1/5,2/1/6,
                                  0/0/7,1/0/8,2/0/9}
    \node [place,tokens=\tokennumber] at (\x,\y) {};
\end{tikzpicture}
```

- `colored tokens`=⟨*color list*⟩ This option, which must also be given when a place node is being created, gets a list of colors as parameter. It will then add as many tokens to the place are in this list, each colored with the corresponding color in the list.

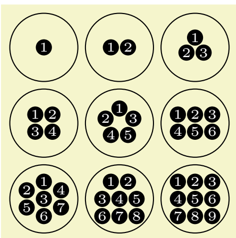


```
\tikz  \node[place,colored tokens={black,black,red,blue}] {};
```

- `structured tokens`=⟨*token texts*⟩ This option, which must again be passed to a place, gets a list texts for tokens. For each text, another token will be added to the place.



```
\tikz  \node[place,structured tokens={$x$,$y$,$z$}] {};
```



```
\begin{tikzpicture}
  \tikzstyle{every place}=[minimum size=9mm]

  \foreach \x/\y/\tokennumber in {0/2/1,1/2/2,2/2/3,
                                  0/1/4,1/1/5,2/1/6,
                                  0/0/7,1/0/8,2/0/9}
    \node [place,structured tokens={1,...,\tokennumber}] at (\x,\y) {};
\end{tikzpicture}
```

If you use lots of structured tokens, consider redefining the `every token` style so that the tokens are larger.

28 Plot Handler Library

```
\usepgflibrary{plothandlers}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[plothandlers]  % ConTeXt and pure pgf
\usetikzlibrary{plothandlers} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[plothandlers] % ConTeXt when using TikZ
```

This library packages defines additional plot handlers, see Section 56.3 for an introduction to plot handlers. The additional handlers are described in the following.

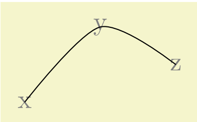
This library is loaded automatically by Ti*k*Z.

28.1 Curve Plot Handlers

`\pgfplothandlercurveto`

This handler will issue a `\pgfpathcurveto` command for each point of the plot, *except* possibly for the first. As for the line-to handler, what happens with the first point can be specified using `\pgfsetmovetofirstplotpoint` or `\pgfsetlinetofirstplotpoint`.

Obviously, the `\pgfpathcurveto` command needs, in addition to the points on the path, some control points. These are generated automatically using a somewhat "dumb" algorithm: Suppose you have three points x, y, and z on the curve such that y is between x and z:

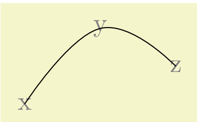



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

In order to determine the control points of the curve at the point y, the handler computes the vector $z - x$ and scales it by the tension factor (see below). Let us call the resulting vector s. Then $y + s$ and $y - s$ will be the control points around y. The first control point at the beginning of the curve will be the beginning itself, once more; likewise the last control point is the end itself.

`\pgfsetplottension{`⟨*value*⟩`}`

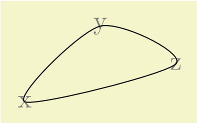
Sets the factor used by the curve plot handlers to determine the distance of the control points from the points they control. The higher the curvature of the curve points, the higher this value should be. A value of 1 will cause four points at quarter positions of a circle to be connected using a circle. The default is 0.5.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfsetplottension{0.75}
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerclosedcurve`

This handler works like the curve-to plot handler, only it will add a new part to the current path that is a closed curve through the plot points.



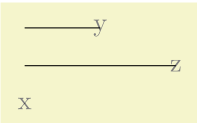

```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerclosedcurve
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

28.2 Comb Plot Handlers

There are three "comb" plot handlers. There name stems from the fact that the plots they produce look like "combs" (more or less).

`\pgfplothandlerxcomb`

This handler converts each point in the plot stream into a line from the y-axis to the point's coordinate, resulting in a "horizontal comb."

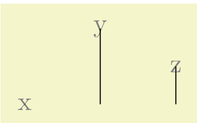



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerxcomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerycomb`

This handler converts each point in the plot stream into a line from the x-axis to the point's coordinate, resulting in a "vertical comb."

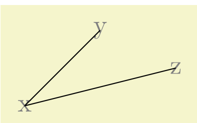
This handler is useful for creating "bar diagrams."




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerycomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplothandlerpolarcomb`

This handler converts each point in the plot stream into a line from the origin to the point's coordinate.



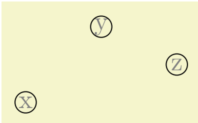

```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerpolarcomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

28.3 Mark Plot Handler

`\pgfplothandlermark{`⟨*mark code*⟩`}`

This command will execute the ⟨*mark code*⟩ for some points of the plot, but each time the coordinate transformation matrix will be setup such that the origin is at the position of the point to be plotted. This way, if the ⟨*mark code*⟩ draws a little circle around the origin, little circles will be drawn at some point of the plot.

By default, a mark is drawn at all points of the plot. However, two parameters r and p influence this. First, only every rth mark is drawn. Second, the first mark drawn is the pth. These parameters can be influenced using the commands below.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

Typically, the ⟨*code*⟩ will be `\pgfuseplotmark{`⟨*plot mark name*⟩`}`, where ⟨*plot mark name*⟩ is the name of a predefined plot mark.

`\pgfsetplotmarkrepeat{`⟨*repeat*⟩`}`

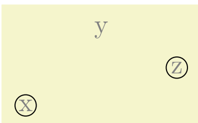
Sets the r parameter to ⟨*repeat*⟩, that is, only every rth mark will be drawn.

`\pgfsetplotmarkphase{`⟨*phase*⟩`}`

Sets the p parameter to ⟨*phase*⟩, that is, the first mark to be drawn is the pth, followed by the $(p+r)$th, then the $(p+2r)$th, and so on.

`\pgfplothandlermarklisted{`⟨*mark code*⟩`}{`⟨*index list*⟩`}`

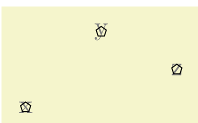
This command works similar to the previous one. However, marks will only be placed at those indices in the given ⟨*index list*⟩. The syntax for the list is the same as for the `\foreach` statement. For example, if you provide the list `1,3,...,25`, a mark will be placed only at every second point. Similarly, `1,2,4,8,16,32` yields marks only at those points that are powers of two.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermarklisted
    {\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
    {1,3}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfuseplotmark{`⟨*plot mark name*⟩`}`

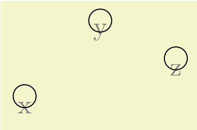
Draws the given ⟨*plot mark name*⟩ at the origin. The ⟨*plot mark name*⟩ must previously have been declared using `\pgfdeclareplotmark`.



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfuseplotmark{pentagon}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfdeclareplotmark{`⟨*plot mark name*⟩`}{`⟨*code*⟩`}`

Declares a plot mark for later used with the `\pgfuseplotmark` command.

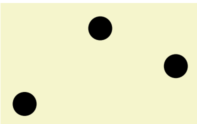



```
\pgfdeclareplotmark{my plot mark}
  {\pgfpathcircle{\pgfpoint{0cm}{1ex}}{1ex}\pgfusepathqstroke}
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfuseplotmark{my plot mark}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfsetplotmarksize{`⟨*dimension*⟩`}`

This command sets the TEX dimension `\pgfplotmarksize` to ⟨*dimension*⟩. This dimension is a "recommendation" for plot mark code at which size the plot mark should be drawn; plot mark code may choose to ignore this ⟨*dimension*⟩ altogether. For circles, ⟨*dimension*⟩ should be the radius, for other shapes it should be about half the width/height.

The predefined plot marks all take this dimension into account.



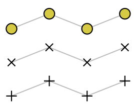
```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfsetplotmarksize{1ex}
  \pgfplothandlermark{\pgfuseplotmark{*}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfplotmarksize`

A TEX dimension that is a "recommendation" for the size of plot marks.

The following plot marks are predefined (the filling color has been set to yellow):

`\pgfuseplotmark{*}`
`\pgfuseplotmark{x}`
`\pgfuseplotmark{+}`



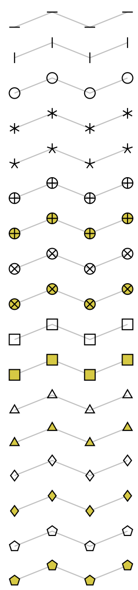
29 Plot Mark Library

```
\usepgflibrary{plotmarks}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[plotmarks]  % ConTeXt and pure pgf
\usetikzlibrary{plotmarks} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[plotmarks] % ConTeXt when using TikZ
```

This library defines a number of plot marks.

This library defines the following plot marks in addition to `*`, `x`, and `+` (the filling color has been set to a dark yellow):

```
\pgfuseplotmark{-}
\pgfuseplotmark{|}
\pgfuseplotmark{o}
\pgfuseplotmark{asterisk}
\pgfuseplotmark{star}
\pgfuseplotmark{oplus}
\pgfuseplotmark{oplus*}
\pgfuseplotmark{otimes}
\pgfuseplotmark{otimes*}
\pgfuseplotmark{square}
\pgfuseplotmark{square*}
\pgfuseplotmark{triangle}
\pgfuseplotmark{triangle*}
\pgfuseplotmark{diamond}
\pgfuseplotmark{diamond*}
\pgfuseplotmark{pentagon}
\pgfuseplotmark{pentagon*}
```



30 Shape Library

In addition to the standard shapes `rectangle`, `circle` and `coordinate`, there exist a number of additional shapes defined in different shape libraries. In the present section, these shapes are described.

```
\usepgflibrary{shapes}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes]  % ConTeXt and pure pgf
\usetikzlibrary{shapes} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes] % ConTeXt when using TikZ
```

This library packages just includes all of the libraries defined in the following. Note that it includes only those libraries starting with `shapes.`, more special-purpose libraries are described in dedicated sections.

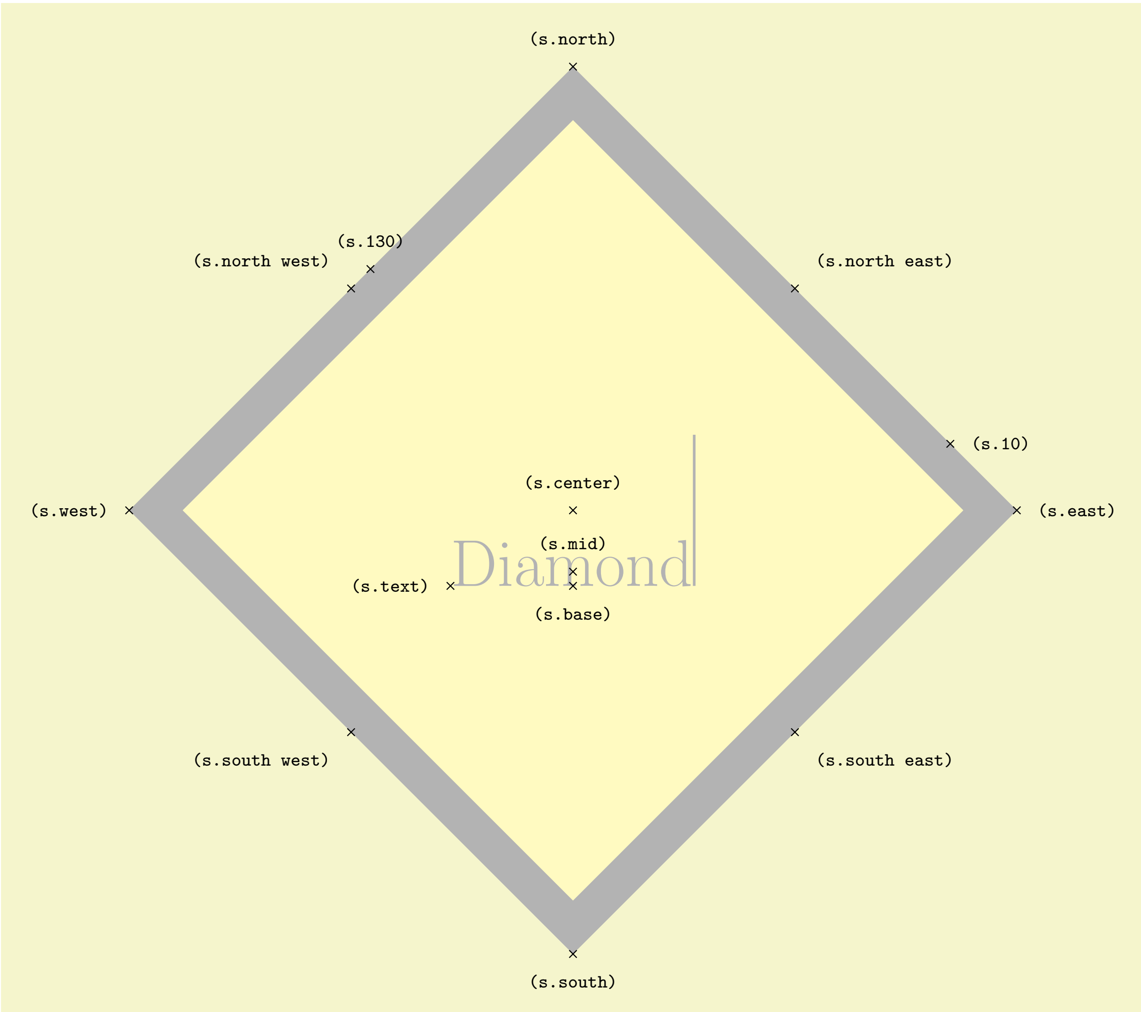
30.1 Geometric Shapes

```
\usepgflibrary{shapes.geometric}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.geometric]  % ConTeXt and pure pgf
\usetikzlibrary{shapes.geometric} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.geometric] % ConTeXt when using TikZ
```

This library defines different shapes that correspond to basic geometric objects like ellipses or polygons.

Shape `diamond`

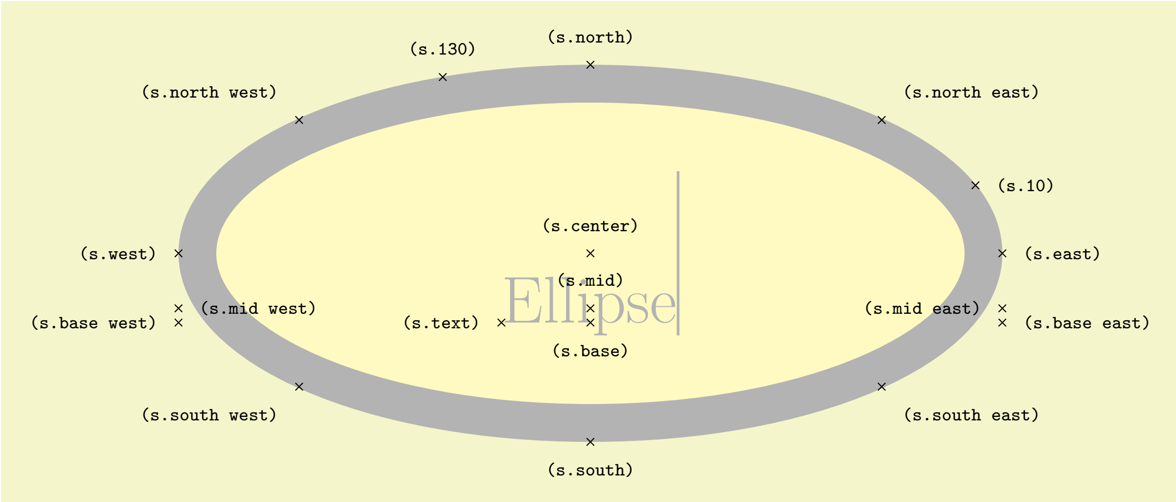
This shape is a diamond tightly fitting the text box. The ratio between width and height is 1 by default, but can be changed by setting the shape aspect ratio (using the `aspect` option of Ti*k*Z). The following figure shows the anchors this shape defines; the anchors `10` and `130` are example of border anchors.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=diamond,style=shape example] {Diamond\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid/above,
     base/below,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape ellipse

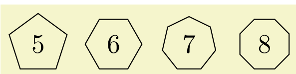
This shape is an ellipse tightly fitting the text box, if no inner separation is given. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.




```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=ellipse,style=shape example] {Ellipse\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

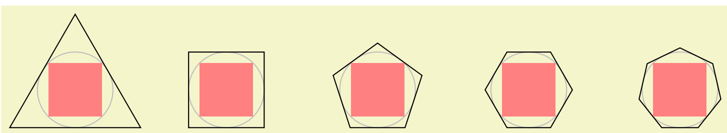
Shape regular polygon

This shape is a regular polygon, which, by default, is drawn so that a side (rather than a corner) is always at the bottom.



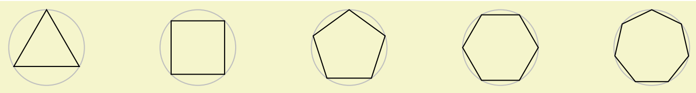

```
\begin{tikzpicture}
   \foreach \i in {5,...,8}
     \node[regular polygon, regular polygon sides=\i, draw] at (\i,0)  {\i};
\end{tikzpicture}
```

Two points should be remembered regarding the dimensions of the polygon border. Firstly, the border is constructed using the incircle, that is, the circle that touches every side of the polygon border. The radius of the incircle is calculated to tightly fit the node contents.



```
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[gray!50] (\a*2,0)  circle(0.5cm);
    \node[regular polygon, regular polygon sides=\a, inner sep=0cm, draw] at (\a*2,0)
      {\tikz\fill[red!50] rectangle(0.707cm,0.707cm);};
   }
\end{tikzpicture}
```

Secondly, if the node is enlarged to any specified minimum size, width or height, this is interpreted as the diameter of the the circumcircle, that is, the circle that passes through all the corners of the polygon border.



```
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[gray!50] (\a*2,0)  circle(0.5cm);
    \node[regular polygon, regular polygon sides=\a, minimum size=1cm, draw] at (\a*2,0) {};
   }
\end{tikzpicture}
```

There are PGF commands and Ti*k*Z options to set the number of sides for the polygon, and the rotation of the polygon border. The PGF commands are as follows:

`\pgfsetpolygonsides`{⟨*integer*⟩}

Set the number of sides for the polygon.

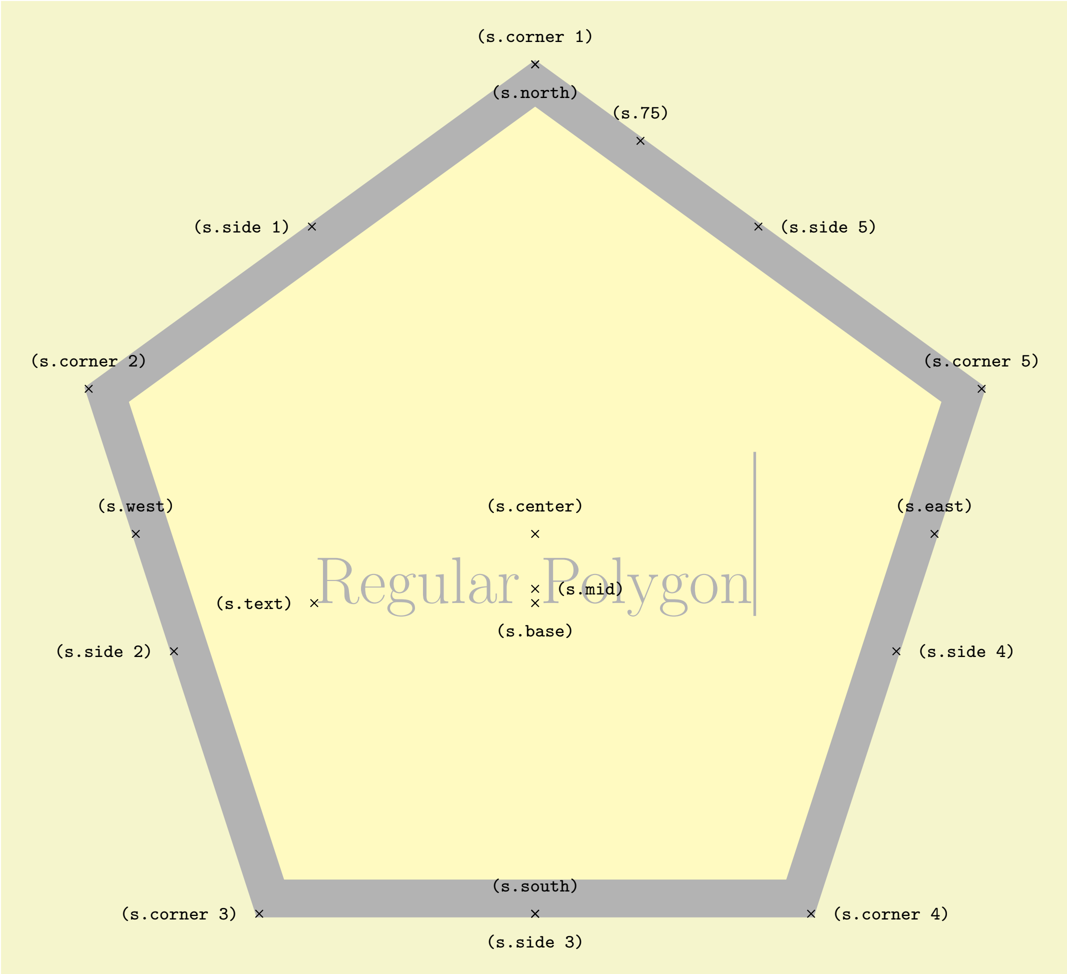
`\pgfsetpolygonrotate`{⟨*angle*⟩}

Rotate the border of the polygon independently of the node contents (but in addition to any concurrent coordinate or canvas transformation).

The corresponding Ti*k*Z options are:

- `regular polygon sides`=⟨*integer*⟩ set the number of points for the star.
- `regular polygon rotate`=⟨*angle*⟩ rotate the border of the polygon independently of the node contents.

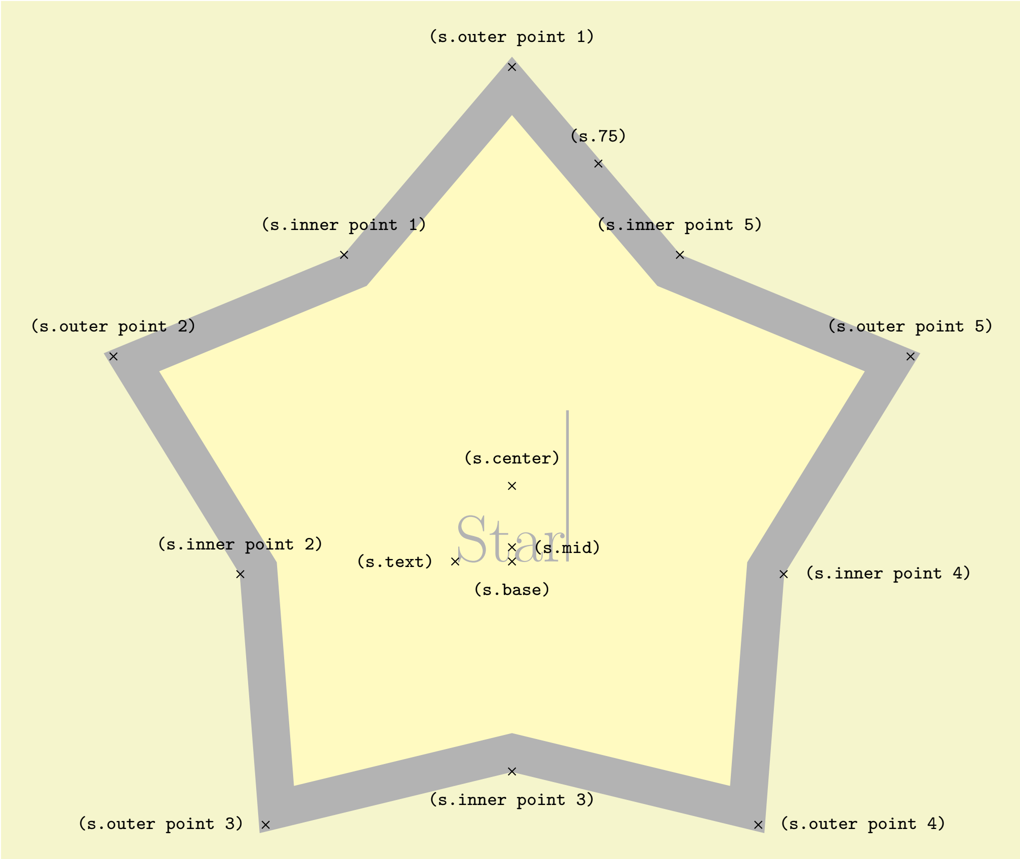
The anchors for the regular polygon shape are shown below:




```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=regular polygon, regular polygon sides=5, style=shape example, inner sep=.5cm]
    {Regular Polygon\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {corner 1/above, corner 2/above, corner 3/left, corner 4/right,
     corner 5/above, side 1/left,    side 2/left,   side 3/below,
     side 4/right,   side 5/right,   mid/right,     base/below,
     center/above,   text/left,      75/above,      west/above,
     east/above,     north/below,    south/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape star

This shape is a star, which by default (minus any transformations) is drawn with the first point pointing upwards. A star should be thought of as having an set of “inner points” and and “outer points”. These points form the principle anchors for the star, as shown below:




```
\Huge
\begin{tikzpicture}
  \node[name=s, shape=star, star points=5, star point ratio=1.5, style=shape example, inner sep=1.5cm]
    {Star\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
     {inner point 1/above, inner point 2/above, inner point 3/below,
      inner point 4/right, inner point 5/above, outer point 1/above,
      outer point 2/above, outer point 3/left,  outer point 4/right,
      outer point 5/above, mid/right,           base/below,
      center/above,        text/left,           75/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

The inner points of the border are based on the radius of the circle which tightly fits the node contents. Any specified minimum size, width or height, is interpreted as the diameter of the circle that passes through every outer point.

There are PGF commands and Ti*k*Z options to set various parameters for the star, such as the number of points, the height of the points and the rotation of the star border.

The PGF commands are as follows:

`\pgfsetstarpoints{`⟨*integer*⟩`}`

Sets the number of points for the star.

`\pgfsetstarpointheight{`⟨*distance*⟩`}`

Sets the height of the star points. This is the distance between the inner point and outer point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the point height.

`\pgfsetstarpointratio{`⟨*number*⟩`}`

Sets the ratio between the outer point and inner point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the ratio.

`\pgfsetstarrotate{`⟨*angle*⟩`}`

Rotates the border of the star, independently of the node contents, but subject to any coordinate or canvas transformations.

The corresponding Ti*k*Z options are:

- `star points`=⟨*integer*⟩ set the number of points for the star.
- `star point height`=⟨*distance*⟩ set the height of the points for the star.
- `star point ratio`=⟨*number*⟩ set the ratio between the outer point radius and the inner point radius.
- `star rotate`=⟨*angle*⟩ rotate the star shape border indepently of the node contents.

30.2 Symbol Shapes

```
\usepgflibrary{shapes.symbols}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.symbols]  % ConTeXt and pure pgf
\usetikzlibrary{shapes.symbols} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.symbols] % ConTeXt when using TikZ
```

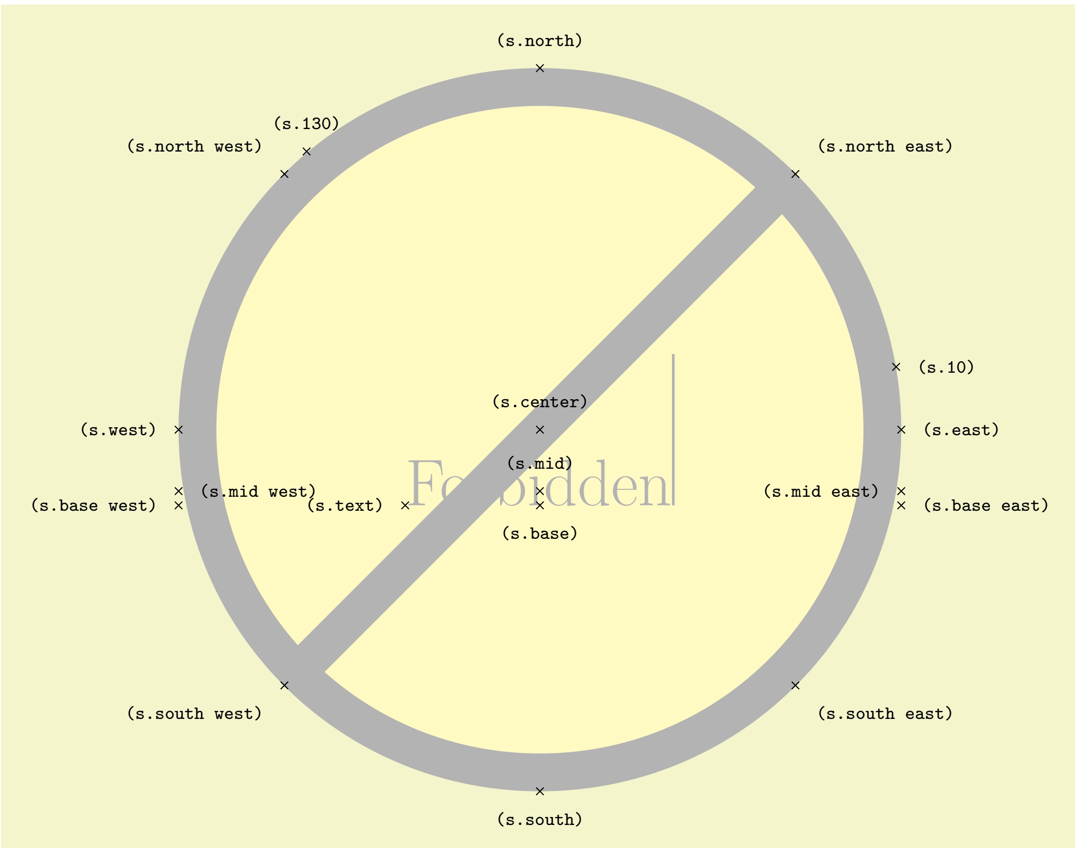
This library defines shapes that can be used for drawing symbols like a forbidden sign or a cloud.

Shape `forbidden sign`

This shape places the node inside a circle with a diagonal from the lower left to the upper right added. The circle is part of the background, the diagonal line part of the foreground path; thus, the diagonal line is on top of the text.


```
\begin{tikzpicture}
  \node [forbidden sign,line width=1ex,draw=red,fill=white] {Smoking};
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape, see also the following figure:



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=forbidden sign,style=shape example] {Forbidden\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

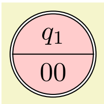
30.3 Shapes with Multiple Text Parts

```
\usepgflibrary{shapes.multipart}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.multipart]  % ConTeXt and pure pgf
\usetikzlibrary{shapes.multipart} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.multipart] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that are composed of multiple (text) parts.

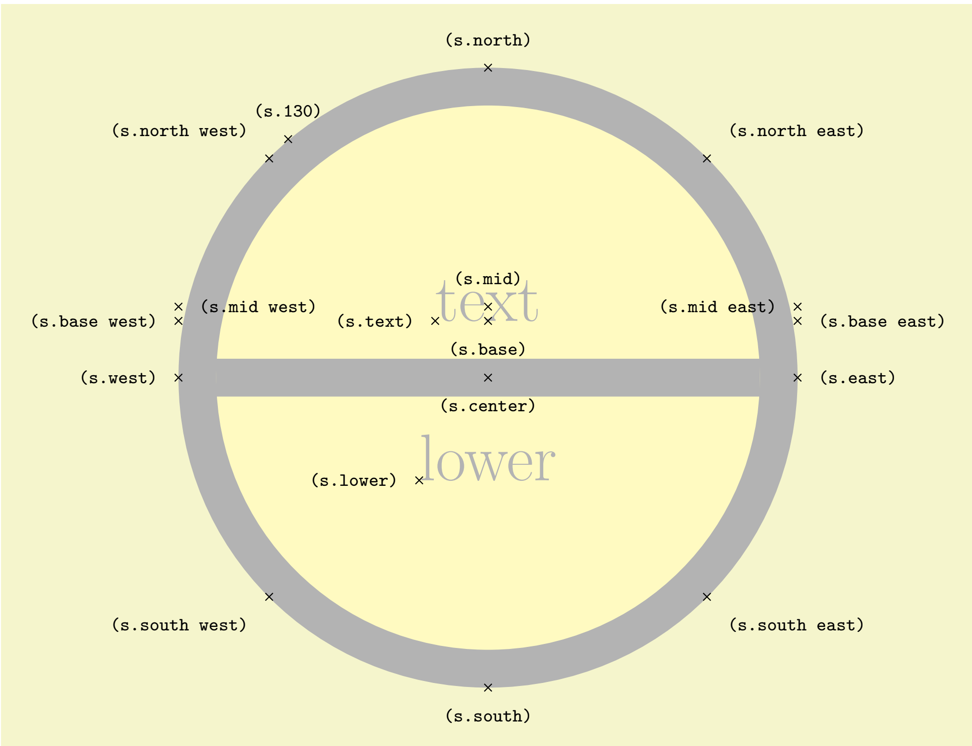
Shape `circle split`

This shape is a multi-part shape consisting of a circle with a line in the middle. The upper part is the main part (the `text` part), the lower part is the `lower` part.




```
\begin{tikzpicture}
  \node [circle split,draw,double,fill=red!20]
  {
    $q_1$
    \nodepart{lower}
    $00$
  };
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape and defines the `lower` anchor in addition. See also the following figure:



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle split,style=shape example] {text\nodepart{lower}lower};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/below, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, lower/left, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

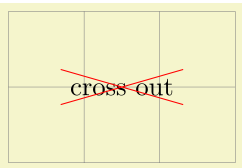
30.4 Miscellaneous Shapes

```
\usepgflibrary{shapes.misc}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[shapes.misc]  % ConTeXt and pure pgf
\usetikzlibrary{shapes.misc} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.misc] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that do not fit in the previous categories.

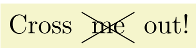
Shape `cross out`

This shape "crosses out" the node. Its foreground path are simply two diagonal lines that between the corners of the node's bounding box. Here is an example:



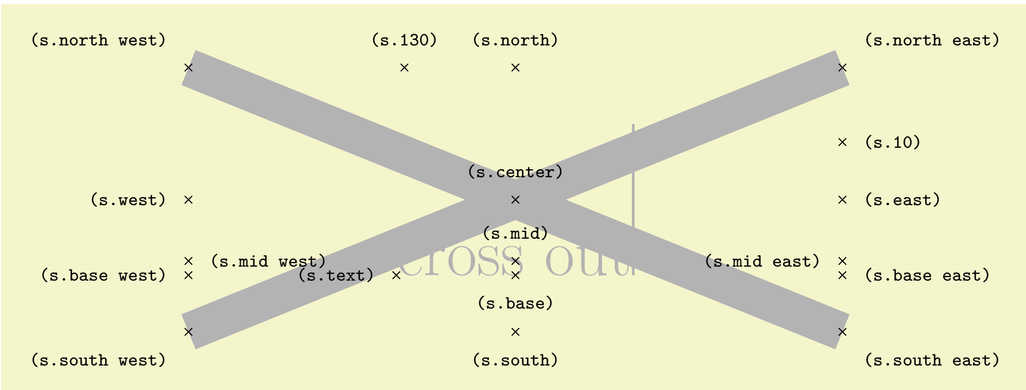

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \node [cross out,draw=red] at (1.5,1) {cross out};
\end{tikzpicture}
```

A useful application is inside text as in the following example:




```
Cross \tikz[baseline] \node [cross out,draw,anchor=text] {me}; out!
```

This shape inherits all anchors from the `rectangle` shape, see also the following figure:




```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=cross out,style=shape example] {cross out\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape strike out

This shape is idential to the `cross out` shape, only its foreground path consists of a single line from the lower left to the upper right.

```
Strike \tikz[baseline] \node [strike out,draw,anchor=text] {me}; out!
```

See the `cross out` shape for the anchors.

31 Snake Library

```
\usepgflibrary{snakes}  % LaTeX and plain TeX and pure pgf
\usepgflibrary[snakes]  % ConTeXt and pure pgf
\usetikzlibrary{snakes} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[snakes] % ConTeXt when using TikZ
```

This library package defines basic snakes. Section 11.2.3 explains how snakes are used in TikZ, Section 46 explains how new snakes can be defined.

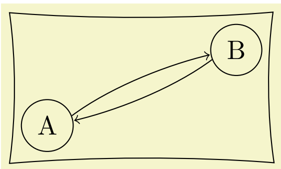
The snakes are influenced by the current values of parameters like `\pgfsnakesegmentamplitude`. Only this parameter and `\pgfsnakesegmentlength` are proper TeX dimensions, all other parameters are TeX macros.

In TikZ, each parameter can be set using an option having the parameters name minus the `\pgfsnake` part.

Snake `bent`

This snake adds a slightly bent line from the start to the target. The amplitude of the bent is given by the segement amplitude (and amplitude of zero gives a straight line).

- `\pgfsnakesegmentamplitude` determines the amplitude of the bent.
- `\pgfsnakesegmentaspect` determines how tight the bent is. A good value is around `0.3`.

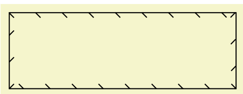



```
\begin{tikzpicture}[segment aspect=.3]
  \node[circle,draw] (A) at (.5,.5) {A};
  \node[circle,draw] (B) at (3,1.5) {B};
  \draw[->,snake=bent,raise snake=2pt] (A) -- (B);
  \draw[->,snake=bent,raise snake=2pt] (B) -- (A);

  \draw [snake=bent,mirror snake] (0,0) rectangle (3.5,2);
\end{tikzpicture}
```

Snake `border`

This snake adds straight lines the path that are at a specific angle to the line toward the target. The idea is to add these little lines to indicate the "border" or an area. The following parameters influence the snake:

- `\pgfsnakesegmentlength` determines the distance between consecutive ticks.
- `\pgfsnakesegmentamplitude` determines the length of the ticks.
- `\pgfsnakesegmentangle` determines the angle between the ticks and the line toward the target.



```
\tikz{\draw (0,0) rectangle (3,1)
            [snake=border,segment angle=-45] (0,0) rectangle (3,1);}
```

Snake `brace`

This snake adds a long brace to the path. The left and right end of the brace will be exactly on the start and endpoint of the snake. The following parameters influence the snake:

- `\pgfsnakesegmentamplitude` determines how much the brace rises above the path.
- `\pgfsnakesegmentaspect` determines the fraction of the total length where the "middle part" of the brace will be.


```
\tikz{\draw[snake=brace,segment aspect=0.25] (0,0) -- (3,0);}
```

Snake `bumps`

This snake consists of little half ellipses. The following parameters influence the snake:

- `\pgfsnakesegmentamplitude` determines the height of the half ellipse.

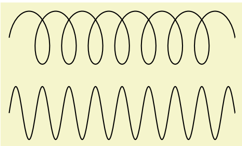
- `\pgfsnakesegmentlength` determines the width of the half ellipse.


```
\tikz{\draw[snake=bumps] (0,0) -- (3,0);}
```

Snake `coil`

This snake adds a coil to the path. To understand how this works, imagine a three-dimensional spring. The spring's axis points along the line toward the target. Then, we "view" the spring from a certain angle. If we look "straight from the side" we will see a perfect sine curve, if we look "more from the front" we will see a coil. The following parameters influence the snake:

- `\pgfsnakesegmentamplitude` determines how much the coil rises above the path and falls below it. Thus, this is the radius of the coil.
- `\pgfsnakesegmentlength` determines the distance between two consecutive "curls." Thus, when the spring is see "from the side" this will be the wave length of the sine curve.
- `\pgfsnakesegmentaspect` determines the "viewing direction." A value of `0` means "looking from the side" and a value of `0.5`, which is the default, means "look more from the front."

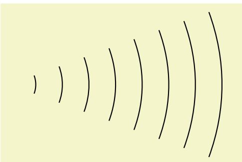


```
\begin{tikzpicture}[segment amplitude=10pt]
  \draw[snake=coil]                  (0,1) -- (3,1);
  \draw[snake=coil,segment aspect=0] (0,0) -- (3,0);
\end{tikzpicture}
```

Snake `expanding waves`

This snake adds arcs to the path that get bigger along the line towards the target. The following parameters influence the snake:

- `\pgfsnakesegmentlength` determines the distance between consecutive arcs.
- `\pgfsnakesegmentangle` determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.



```
\tikz{\draw[snake=expanding waves] (0,0) -- (3,0);}
```

Snake `saw`

This snake looks like the blade of a saw. The following parameters influence the snake:

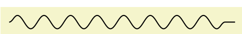
- `\pgfsnakesegmentamplitude` determines how much each spike raises above the straight line.
- `\pgfsnakesegmentlength` determines the length each spike.


```
\tikz{\draw[snake=saw] (0,0) -- (3,0);}
```

Snake `snake`

This snake is the "architypical" snake: It looks like a snake seen from above. More precisely, the snake is a sine wave with a "softened" start and ending. The following parameters influence the snake:

- `\pgfsnakesegmentamplitude` determines the sine wave's amplitude.
- `\pgfsnakesegmentlength` determines the sine wave's wave length.

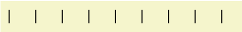


```
\tikz{\draw[snake=snake] (0,0) -- (3,0);}
```

Snake `ticks`

This snake adds straight lines the path that are orthogonal to the line toward the target. The following parameters influence the snake:

- `\pgfsnakesegmentlength` determines the distance between consecutive ticks.
- `\pgfsnakesegmentamplitude` determines half the length of the ticks.



```
\tikz{\draw[snake=ticks] (0,0) -- (3,0);}
```

Snake `triangles`

This snake adds triangles to the path that point toward the target. The following parameters influence the snake:

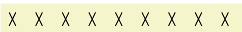
- `\pgfsnakesegmentlength` determines the distance between consecutive triangles.
- `\pgfsnakesegmentamplitude` determines half the length of the triangle side that is orthogonal to the path.
- `\pgfsnakesegmentobjectlength` determines the height of the triangle.

```
\tikz{\draw[snake=triangles] (0,0) -- (3,0);}
```

Snake `crosses`

This snake adds (diagonal) crosses to the path. The following parameters influence the snake:

- `\pgfsnakesegmentlength` determines the distance between consecutive crosses.
- `\pgfsnakesegmentamplitude` determines half the hieght of each cross.
- `\pgfsnakesegmentobjectlength` determines width of each cross.



```
\tikz{\draw[snake=crosses] (0,0) -- (3,0);}
```

Snake `waves`

This snake adds arcs to the path that have a constant size. The following parameters influence the snake:

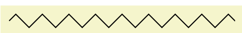
- `\pgfsnakesegmentlength` determines the distance between consecutive arcs.
- `\pgfsnakesegmentangle` determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.
- `\pgfsnakesegmentamplitude` determines the radius of each arc.


```
\tikz{\draw[snake=waves] (0,0) -- (3,0);}
```

Snake `zigzag`

This snake looks like a zig-zag line. The following parameters influence the snake:

- `\pgfsnakesegmentamplitude` determines how much the zig-zag lines raises above and falls below a straight line to the target point.
- `\pgfsnakesegmentlength` determines the length of a complete “up-down” cycle.



```
\tikz{\draw[snake=zigzag] (0,0) -- (3,0);}
```

32 To Path Library

```
\usetikzlibrary{topaths} % LaTeX and plain TeX
\usetikzlibrary[topaths] % ConTeXt
```

This library provides predefined to paths for use with the `to` path operation. After loading this package, you can say for instance `to [loop]` to add a loop to a node.

This library is loaded automatically by Ti*k*Z, so you do not need to load it yourself.

32.1 Straight Lines

The following style installs a to path that is simply a straight line from the start coordinate to the target coordinate.

- `style=line to` causes a straight line to be added to the path upon a `to` or an `edge` operation.


```
\tikz {\draw (0,0) to[line to] (1,0);}
```

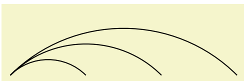
32.2 Curves

The `curve to` style causes the to path to be set to a curve. The exact way this curve looks can be influenced via a number of options.

- `style=curve to` Specifies that the `to path` should be a curve. This curve will leave the start coordinate at a certain angle, which can be specified using the `out` option. It reaches the target coordinate also at a certain angle, which is specified using the `in` option. The control points of the curve are at a certain distance that is computed in different ways, depending on which options are set.

  All of the following options implictly cause the `curve to` style to be installed.

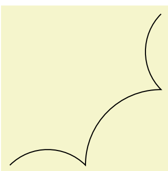
- `out=`⟨*angle*⟩ The angle at which the curve leaves the start coordinate. If the start coordinate is a node, the start coordinate is the point on the border of the node at the given ⟨*angle*⟩. The control point will, thus, lie at a certain distance in the direction ⟨*angle*⟩ from the start coordinate.



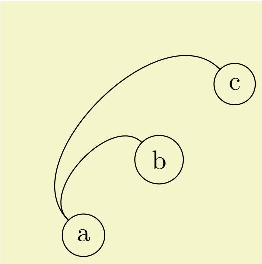
```
\begin{tikzpicture}[out=45,in=135]
  \draw (0,0) to (1,0)
        (0,0) to (2,0)
        (0,0) to (3,0);
\end{tikzpicture}
```

- `in=`⟨*angle*⟩ The angle at which the curve reaches the target coordinate.

- `relative=`⟨*true or false*⟩ This option tells Ti*k*Z whether the `in` and `out` angles should be considered absolute or relative. Absolute means that an `out` angle of 30° means that the curve leaves the start coordinate at an angle of 30° relative to the paper (unless, of course, further transformations have been installed). A *relative* angle is, by comparison, measured relative to a straight line from the start coordinate to the target coordinate. Thus, a relative angle of 30° means that the curve will bend to the left from the line going straight from the start to the target. For the target, the relative coordinate is measured in the same manner, namely relative to the line going from the start to the target. Thus, an angle of 150° means that the curve will reach target coming slightly from the left.

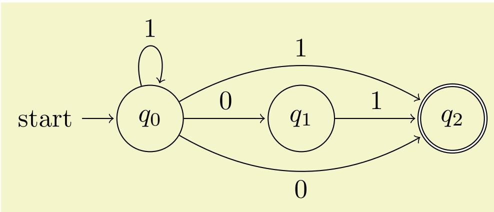


```
\begin{tikzpicture}[out=45,in=135,relative]
  \draw (0,0) to (1,0)
              to (2,1)
              to (2,2);
\end{tikzpicture}
```

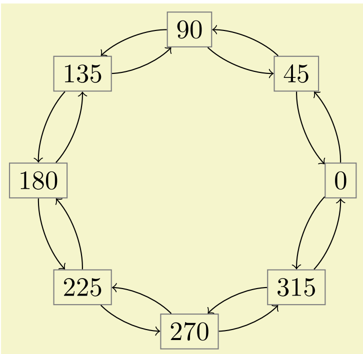



```
\begin{tikzpicture}[out=90,in=90,relative]
  \node [circle,draw] (a) at (0,0) {a};
  \node [circle,draw] (b) at (1,1) {b};
  \node [circle,draw] (c) at (2,2) {c};

  \path (a) edge (b)
            edge (c);
\end{tikzpicture}
```

- bend left=⟨*angle*⟩ This option sets out=⟨*angle*⟩,in=$180 - \langle angle \rangle$,relative. If no ⟨*angle*⟩ is given, the last given bend left or bend right angle is used.




```
\begin{tikzpicture}[shorten >=1pt,node distance=2cm]
  \node[state,initial]  (q_0)                {$q_0$};
  \node[state]          (q_1) [right of=q_0] {$q_1$};
  \node[state,accepting](q_2) [right of=q_1] {$q_2$};

  \path[->] (q_0) edge              node [above]  {0} (q_1)
                  edge [loop above] node          {1} ()
                  edge [bend left]  node [above]  {1} (q_2)
                  edge [bend right] node [below]  {0} (q_2)
            (q_1) edge              node [above]  {1} (q_2);
\end{tikzpicture}
```

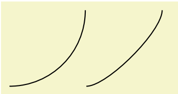



```
\begin{tikzpicture}
  \foreach \angle in {0,45,...,315}
    \node[rectangle,draw=black!50] (\angle) at (\angle:2) {\angle};

  \foreach \from/\to in {0/45,45/90,90/135,135/180,
                         180/225,225/270,270/315,315/0}
    \path (\from) edge [->,bend right=22,looseness=0.8] (\to)
                  edge [<-,bend left=22,looseness=0.8] (\to);
\end{tikzpicture}
```

- bend right=⟨*angle*⟩ Works like the bend left option, only the bend is to the other side.
- bend angle=⟨*angle*⟩ Sets the angle to be used by the bend left or bend right, but without actually selecting the curve to or the relative option. This is useful for globally specifying a bend angle for a whole picture.
- looseness=⟨*number*⟩ This number specifies how "loose" the curve will be. In detail, the following happens: Ti*k*Z computes the distance between the start and the target coordinate (if the start and/or target coordinate are nodes, the distance is computed between the points on their border). This

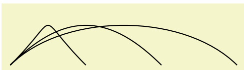
distance is then multiplied by a fixed factor and also by the factor ⟨*number*⟩. The resulting distance, let us call it d, is then used as the distance of the control points from the start and target coordinates.

The fixed factor has been chosen in such a way that if ⟨*number*⟩ is `1`, which is the default, if the `in` and `out` angles differ by 90○, then a quarter circle results:



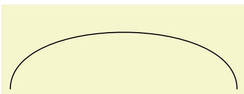
```
\tikz \draw (0,0) to [out=0,in=-90]               (1,1);
\tikz \draw (0,0) to [out=0,in=-90,looseness=0.5] (1,1);
```

- `out looseness`=⟨*number*⟩ specifies the looseness factor for the out distance only.
- `in looseness`=⟨*number*⟩ specifies the looseness factor for the in distance only.
- `min distance`=⟨*distance*⟩ If the computed distance for the start and target coordinates are below ⟨*distance*⟩, then ⟨*distance*⟩ is used instead.
- `max distance`=⟨*distance*⟩ If the computed distance for the start and target coordinates are above ⟨*distance*⟩, then ⟨*distance*⟩ is used instead.
- `out min distance`=⟨*distance*⟩ The min distance set only for the start coordinate.
- `out max distance`=⟨*distance*⟩ The max distance set only for the start coordinate.
- `in min distance`=⟨*distance*⟩ The min distance set only for the target coordinate.
- `in max distance`=⟨*distance*⟩ The max distance set only for the target coordinate.
- `distance`=⟨*distance*⟩ Set the min and max distance to the same value ⟨*distance*⟩. Note that this causes any computed distance d to be ignored and ⟨*distance*⟩ to be used instead.



```
\begin{tikzpicture}[out=45,in=135,distance=1cm]
  \draw (0,0) to (1,0)
        (0,0) to (2,0)
        (0,0) to (3,0);
\end{tikzpicture}
```

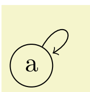
- `out distance`=⟨*distance*⟩ sets the min and max out distance.
- `in distance`=⟨*distance*⟩ sets the min and max in distance.
- `out control`=⟨*coordinate*⟩ This option causes the ⟨*coordinate*⟩ to be used as the start control point. All computations of d are ignored. You can use a coordinate like `+(1,0)` to specify a point relative to the start coordinate.
- `in control`=⟨*coordinate*⟩ This option causes the ⟨*coordinate*⟩ to be used as the target control point.
- `controls`=⟨*coordinate*⟩ `and` ⟨*coordinate*⟩ This option causes the ⟨*coordinate*⟩s to be used as control points.



```
\tikz \draw (0,0) to [controls=+(90:1) and +(90:1)] (3,0);
```

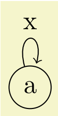
32.3 Loops

- `style=loop` This style is similar to the `curve to` style, but differs in the following ways: First, the actual target coordinate is ignored and the start coordiante is used as the target coordinate. Thus, it is allowed not to provide any target coordinate, which can be useful with unnamed nodes. Second, the `looseness` is set to `8` and the `min distance` to `5mm`. These settings result in rather nice loops when the opening angle (difference between `in` and `out`) is 30°.



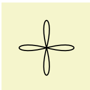

```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [in=30,out=60,loop] ();
\end{tikzpicture}
```

- style=loop above Sets the loop style and sets in and out angles such that loop is above the node. Furthermore, the above option is set, which causes a node label to be placed at the correct position.




```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [loop above] node {x} ();
\end{tikzpicture}
```

- style=loop below works like the previous option.
- style=loop left works like the previous option.
- style=loop right works like the previous option.
- style=every loop This style is installed at the beginning of every loop. By default, it is set to ->,shorten >=1pt, but feel free to change this.



```
\begin{tikzpicture}
  \tikzstyle{every loop}=[]
  \draw (0,0) to [loop above] () to [loop right] ()
              to [loop below] () to [loop left]  ();
\end{tikzpicture}
```

33 Tree Library

```
\usetikzlibrary{trees} % LaTeX and plain TeX
\usetikzlibrary[trees] % ConTeXt
```

This packages defines styles to be used when drawing trees.

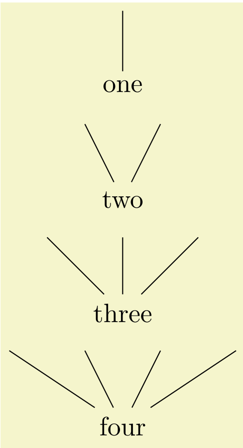
33.1 Growth Functions

The package `pgflibrarytikztrees` defines two new growth functions. They are installed using the following options:

- `grow via three points=one child at (`$\langle x\rangle$`) and two children at (`$\langle y\rangle$`) and (`$\langle z\rangle$`)` This option installs a growth function that works as follows: If a parent node has just one child, this child is placed at $\langle x\rangle$. If the parent node has two children, these are placed at $\langle y\rangle$ and $\langle z\rangle$. If the parent node has more than two children, the children are placed at points that are linearly extrapolated from the three points $\langle x\rangle$, $\langle y\rangle$, and $\langle z\rangle$. In detail, the position is $x + \frac{n-1}{2}(y-x)+(c-1)(z-y)$, where n is the number of children and c is the number of the current child (starting with 1).

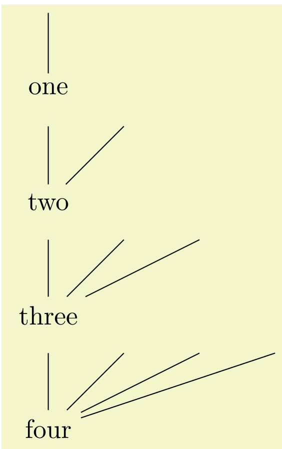
  The net effect of all this is that if you have a certain "linear arrangement" in mind and use this option to specify the placement of a single child and of two children, then any number of children will be placed correctly.

  Here are some arrangements based on this growth function. We start with a simple "above" arrangement:



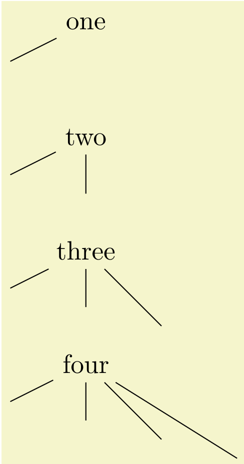

```
\begin{tikzpicture}[grow via three points={%
    one child at (0,1) and two children at (-.5,1) and (.5,1)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

  The next arrangement places children above, but "grows only to the right."




```
\begin{tikzpicture}[grow via three points={%
    one child at (0,1) and two children at (0,1) and (1,1)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

  In the final arrangement, the children are placed along a line going down and right.



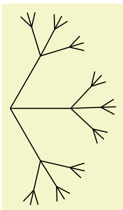

```
\begin{tikzpicture}[grow via three points={%
    one child at (-1,-.5) and two children at (-1,-.5) and (0,-.75)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

These examples should make it clear how you can create new styles to arrange your children along a line.

- `style=grow cyclic` This style causes the children to be arranged "on a circle." For this, the children are placed at distance `\tikzleveldistance` from the parent node, but not on a straight line, but points on a circle. Instead of a sibling distance, there is a `sibling angle` that denotes the angle between two given children.

  – `sibling angle=`⟨*angle*⟩ Sets the angle between siblings in the `grow cyclic` style.

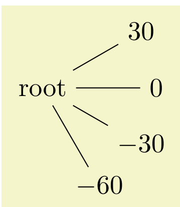
  Note that this function will rotate the coordinate system of the children to ensure that the grandchildren will grow in the right direction.



```
\begin{tikzpicture}[grow cyclic]
  \tikzstyle{level 1}=[level distance=8mm,sibling angle=60]
  \tikzstyle{level 2}=[level distance=4mm,sibling angle=45]
  \tikzstyle{level 3}=[level distance=2mm,sibling angle=30]
  \coordinate [rotate=-90] % going down
    child foreach \x in {1,2,3}
      {child foreach \x in {1,2,3}
        {child foreach \x in {1,2,3}}};
\end{tikzpicture}
```

- `clockwise from=`⟨*angle*⟩ This option also cuases children to be arranged on a circle. However, the rule for placing children is simpler thatn with the `grow cyclic` style: The first child is placed at ⟨*angle*⟩ at a distance of `\tikzleveldistance`. The second child is placed at the same distance from the parent, but at angle ⟨*angle*⟩ − `\tikzsiblingangle`. The third child is displaced by another `\tikzsiblingangle` in a clockwise fashion, and so on.

  Note that this function will not rotate the coordinate system.



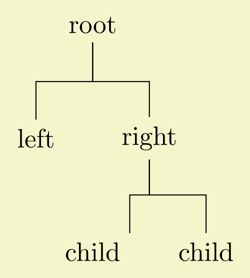

```
\begin{tikzpicture}
  \node {root}
  [clockwise from=30,sibling angle=30]
  child {node {$30$}}
  child {node {$0$}}
  child {node {$-30$}}
  child {node {$-60$}};
\end{tikzpicture}
```

- `counterclockwise from=`⟨*angle*⟩ Works the same way as `clockwise from`, but sibling angles are added instead of subtracted.

33.2 Edges From Parent

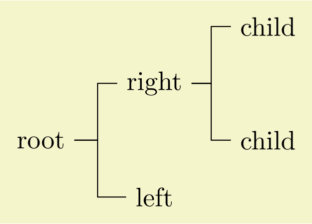
The following styles can be used to modify how the edges from parents are drawn:

- `style=edge from parent fork down` This style will draw a line from the parent downwards (for half the level distance) and then on to the child using only horizontal and vertical lines.




```
\begin{tikzpicture}
  \node {root}
    [edge from parent fork down]
    child {node {left}}
    child {node {right}
      child[child anchor=north east] {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

- `style=edge from parent fork right` This style behaves similarly, only it will first draw its edge to the right.



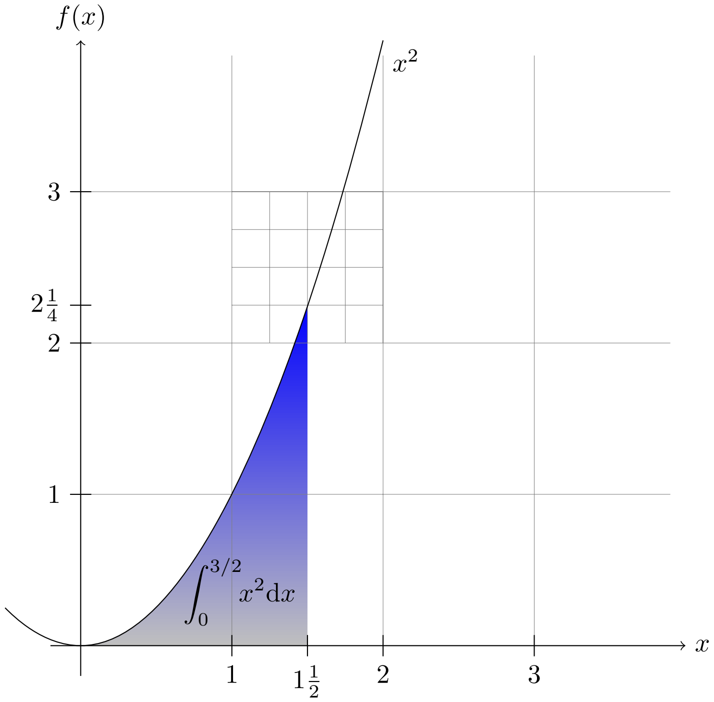

```
\begin{tikzpicture}
  \node {root}
    [edge from parent fork right,grow=right]
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

- `style=edge from parent fork left` behaves similary.
- `style=edge from parent fork up` behaves similary.

Part V
Utilities

by Till Tantau

The utility packages are not directly involved in creating graphics, but you may find them useful nonetheless. All of them either directly depend on PGF or they are designed to work well together with PGF even though they can be used in a stand-alone way.




```
\begin{tikzpicture}[scale=2]
  \shade[top color=blue,bottom color=gray!50] (0,0) parabola (1.5,2.25) |- (0,0);
  \draw (1.05cm,2pt) node[above] {$\displaystyle\int_0^{3/2} \!\!x^2\mathrm{d}x$};

  \draw[style=help lines] (0,0) grid (3.9,3.9)
       [step=0.25cm]      (1,2) grid +(1,1);

  \draw[->] (-0.2,0) -- (4,0) node[right] {$x$};
  \draw[->] (0,-0.2) -- (0,4) node[above] {$f(x)$};

  \foreach \x/\xtext in {1/1, 1.5/1\frac{1}{2}, 2/2, 3/3}
    \draw[shift={(\x,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\xtext$};

  \foreach \y/\ytext in {1/1, 2/2, 2.25/2\frac{1}{4}, 3/3}
    \draw[shift={(0,\y)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\ytext$};

  \draw (-.5,.25) parabola bend (0,0) (2,4) node[below right] {$x^2$};
\end{tikzpicture}
```

34 Repeating Things: The Foreach Statement

This section describes the package `pgffor`, which is loaded automatically by Ti*k*Z, but not by PGF:

```
\usepackage{pgffor} % LaTeX
\input pgffor.tex   % plain TeX
\usemodule[pgffor]  % ConTeXt
```

This package can be used independently of PGF, but works particularly well together with PGF and Ti*k*Z. It defines two new commands: `\foreach` and `\breakforeach`.

`\foreach` ⟨*variables*⟩ `in {`⟨*list*⟩`}` ⟨*commands*⟩

The syntax of this command is a bit complicated, so let us go through it step-by-step.

In the easiest case, ⟨*variables*⟩ is a single TeX-command like `\x` or `\point`. (If you want to have some fun, you can also use active characters. If you do not know what active characters are, you are blessed.)

Still in the easiest case, ⟨*list*⟩ is a comma-separated list of values. Anything can be used as a value, but numbers are most likely.

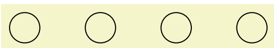
Finally, in the easiest case, ⟨*commands*⟩ is some TeX-text in curly braces.

With all these assumptions, the `\foreach` statement will execute the ⟨*commands*⟩ repeatedly, once for every element of the ⟨*list*⟩. Each time the ⟨*commands*⟩ are executed, the ⟨*variable*⟩ will be set to the current value of the list item.

[1][2][3][0]

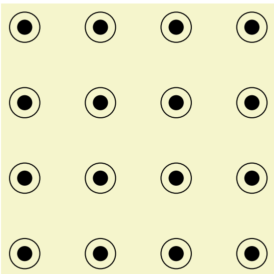
```
\foreach \x in {1,2,3,0} {[\x]}
```

Syntax for the commands. Let us move on to a more complicated setting. The first complication occurs when the ⟨*commands*⟩ are not some text in curly braces. If the `\foreach` statement does not encounter an opening brace, it will instead scan everything up to the next semicolon and use this as ⟨*commands*⟩. This is most useful in situations like the following:



```
\tikz
  \foreach \x in {0,1,2,3}
    \draw (\x,0) circle (0.2cm);
```

However, the "reading till the next semicolon" is not the whole truth. There is another rule: If a `\foreach` statement is directly followed by another `\foreach` statement, this second foreach statement is collected as ⟨*commands*⟩. This allows you to write the following:



```
\begin{tikzpicture}
  \foreach \x in {0,1,2,3}
    \foreach \y in {0,1,2,3}
      {
        \draw (\x,\y) circle (0.2cm);
        \fill (\x,\y) circle (0.1cm);
      }
\end{tikzpicture}
```

The dots notation. The second complication concerns the ⟨*list*⟩. If this ⟨*list*⟩ contains the list item "`...`", this list item is replaced by the "missing values." More precisely, the following happens:

Normally, when a list item `...` is encountered, there should already have been *two* list items before it, which where numbers. Examples of *numbers* are `1`, `-10`, or `-0.24`. Let us call these numbers x and y and let $d := y - x$ be their difference. Next, there should also be one number following the three dots, let us call this number z.

In this situation, the part of the list reading "x`,`y`,...,`z" is replaced by "x, $x + d$, $x + 2d$, $x + 3d$, $\dots$, $x + md$," where the last dots are semantic dots, not syntactic dots. The value m is the largest number such that $x + md \le z$ if d is positive or such that $x + md \ge z$ if d is negative.

Perhaps it is best to explain this by some examples: The following ⟨*list*⟩ have the same effects:

`\foreach \x in {1,2,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

`\foreach \x in {1,2,3,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

`\foreach \x in {1,3,...,11} {\x, }` yields 1, 3, 5, 7, 9, 11,

`\foreach \x in {1,3,...,10} {\x, }` yields 1, 3, 5, 7, 9,

`\foreach \x in {0,0.1,...,0.5} {\x, }` yields 0, 0.1, 0.20001, 0.30002, 0.40002,

`\foreach \x in {a,b,9,8,...,1,2,2.125,...,2.5} {\x, }` yields a, b, 9, 8, 7, 6, 5, 4, 3, 2, 1, 2, 2.125, 2.25, 2.375, 2.5,

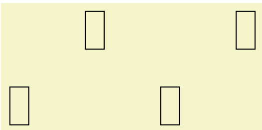
As can be seen, for fractional steps that are not multiples of 2^{-n} for some small n, rounding errors can occur pretty easily. Thus, in the second last case, `0.5` should probably be replaced by `0.501` for robustness.

There is yet another special case for the `...` statement: If the `...` is used right after the first item in the list, that is, if there is an x, but no y, the difference d obviously cannot be computed and is set to 1 if the number z following the dots is larger than x and is set to -1 if z is smaller:

`\foreach \x in {1,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

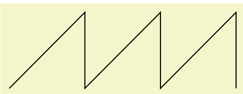
`\foreach \x in {9,...,3.5} {\x, }` yields 9, 8, 7, 6, 5, 4,

Special handling of pairs. Different list items are separated by commas. However, this causes a problem when the list items contain commas themselves as pairs like `(0,1)` do. In this case, you should put the items containing commas in braces as in `{(0,1)}`. However, since pairs are such a natural and useful case, they get a special treatment by the `\foreach` statement. When a list item starts with a `(` everything up to the next `)` is made part of the item. Thus, we can write things like the following:

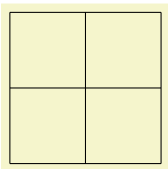


```
\tikz
  \foreach \position in {(0,0), (1,1), (2,0), (3,1)}
    \draw \position rectangle +(.25,.5);
```

Using the foreach-statement inside paths. Ti*k*Z allows you to use a `\foreach` statement inside a path construction. In such a case, the ⟨*commands*⟩ must be path construction commands. Here are two examples:



```
\tikz
  \draw (0,0)
    \foreach \x in {1,...,3}
      { -- (\x,1) -- (\x,0) }
    ;
```



```
\tikz \draw \foreach \p in {1,...,3} {(\p,1)--(\p,3) (1,\p)--(3,\p)};
```

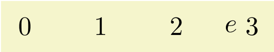
Multiple variables. You will often wish to iterate over two variables at the same time. Since you can nest `\foreach` loops, this is normally straight-forward. However, you sometimes wish variables to iterate "simultaneously." For example, we might be given a list of edges that connect two coordinates and might wish to iterate over these edges. While doing so, we would like the source and target of the edges to be set to two different variables.

To achieve this, you can use the following syntax: The ⟨*variables*⟩ may not only be a single TeX-variable. Instead, it can also be a list of variables separated by slashes (`/`). In this case the list items can also be lists of values separated by slashes.

Assuming that the ⟨*variables*⟩ and the list items are lists of values, each time the ⟨*commands*⟩ are executed, each of the variables in ⟨*variables*⟩ is set to one part of the list making up the current list item. Here is an example to clarify this:

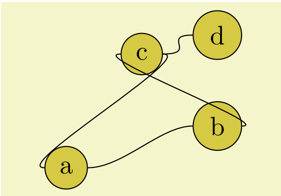
Example: `\foreach \x / \y in {1/2,a/b} {``\x\ and \y''}` yields "1 and 2" "a and b".

If some entry in the ⟨*list*⟩ does not have "enough" slashes, the last entry will be repeated. Here is an example:

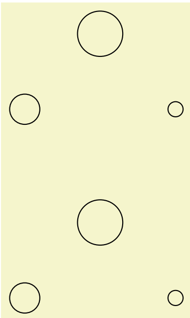



```
\begin{tikzpicture}
  \foreach \x/\xtext in {0,...,3,2.72 / e}
    \draw (\x,0) node{$\xtext$};
\end{tikzpicture}
```

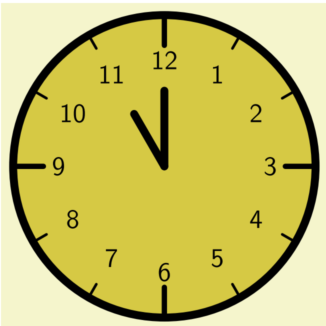
Here are more useful examples:



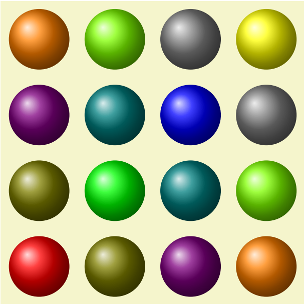

```
\begin{tikzpicture}
  % Define some coordinates:
  \tikzstyle{every node}=[draw,fill]
  \path[shape=circle,fill=examplefill]
    (0,0)    node(a) {a}
    (2,0.55) node(b) {b}
    (1,1.5)  node(c) {c}
    (2,1.75) node(d) {d};

  % Draw some connections:
  \foreach \source/\target in {a/b, b/c, c/a, c/d}
    \draw (\source) .. controls +(.75cm,0pt) and +(-.75cm,0pt)..(\target);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  % Let's draw circles at interesting points:
  \foreach \x / \y / \diameter in {0 / 0 / 2mm, 1 / 1 / 3mm, 2 / 0 / 1mm}
    \draw (\x,\y) circle (\diameter);

  % Same effect
  \foreach \center/\diameter in {{(0,0)/2mm}, {(1,1)/3mm}, {(2,0)/1mm}}
    \draw[yshift=2.5cm] \center circle (\diameter);
\end{tikzpicture}
```



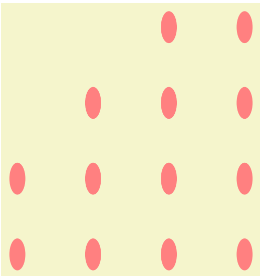

```
\begin{tikzpicture}[cap=round,line width=3pt]
  \filldraw [fill=examplefill] (0,0) circle (2cm);

  \foreach \angle / \label in
    {0/3, 30/2, 60/1, 90/12, 120/11, 150/10, 180/9,
     210/8, 240/7, 270/6, 300/5, 330/4}
  {
    \draw[line width=1pt] (\angle:1.8cm) -- (\angle:2cm);
    \draw (\angle:1.4cm) node{\textsf{\label}};
  }

  \foreach \angle in {0,90,180,270}
    \draw[line width=2pt] (\angle:1.6cm) -- (\angle:2cm);

  \draw (0,0) -- (120:0.8cm); % hour
  \draw (0,0) -- (90:1cm);    % minute
\end{tikzpicture}%
```



```
\tikz[shading=ball]
  \foreach \x / \cola in {0/red,1/green,2/blue,3/yellow}
    \foreach \y / \colb in {0/red,1/green,2/blue,3/yellow}
      \shade[ball color=\cola!50!\colb] (\x,\y) circle (0.4cm);
```

`\breakforeach`

If this command is given inside a `\foreach` command, no further executions of the ⟨*commands*⟩ will occur. However, the current execution of the ⟨*commands*⟩ is continued normally, so it is probably best to use this command only at the end of a `\foreach` command.



```
\begin{tikzpicture}
  \foreach \x in {1,...,4}
    \foreach \y in {1,...,4}
    {
      \fill[red!50] (\x,\y) ellipse (3pt and 6pt);

      \ifnum \x<\y
        \breakforeach
      \fi
    }
\end{tikzpicture}
```

35 Date and Calendar Utility Macros

This section describes the package `pgfcalendar`.

```
\usepackage{pgfcalendar} % LaTeX
\input pgfcalendar.tex   % plain TeX
\usemodule[pgfcalendar]  % ConTeXt
```

This package can be used independently of PGF. It has two purposes:

1. It provides functions for working with dates. Most noticably, it can convert a date in ISO-standard format (like 1975-12-26) to a so-called Julian day number, which is defined in Wikipedia as follows: "The Julian day or Julian day number is the (integer) number of days that have elapsed since the initial epoch at noon Universal Time (UT) Monday, January 1, 4713 BC in the proleptic Julian calendar." The package also provides a function for converting a Julian day number to an ISO-format date.

   Julian day numbers make it very easy to work with days. For example, the date ten days in the future of 2008-02-20 can be computed by converting this date to a Julian day number, adding 10, and then converting it back. Also, the day of week of a given date can be computed by taking the Julian day number modulo 7.

2. It provides a macro for typesetting a calendar. This macro is highly configurable and flexible (for example, it can produce both plain text calendars and also complicated Ti*k*Z-based calendars), but most users will not use the macro directly. It is the job of a frontend to provide useful configruations for typesetting calendars based on this command.

35.1 Handling Dates

35.1.1 Conversions Between Date Types

`\pgfcalendardatetojulian{`⟨*date*⟩`}{`⟨*counter*⟩`}`

This macro converts a date in a format to be described in a moment to the Julian day number in the Gregorian calendar. The ⟨*date*⟩ should expand to a string of the following form:

1. It should start with a number representing the year. Use `\year` for the current year, that is, the year the file is being typeset.
2. The year must be followed by a hyphen.
3. Next should come a number representing the month. Use `\month` for the current month. You can, but need not, use leading zeros. For example, `02` represents February, just like `2`.
4. The month must also be followed by a hyphen.
5. Next you must either provide a day of month (again, a number and, again, `\day` yields the current day of month) or the keyword `last`. This keyword refers to the last day of the month, which is automatically computed (and which is a bit tricky to compute, especially for February).
6. Optionally, you can next provide a plus sign followed by positive or negative number. This number of days will be added to the computed date.

Here are some examples:

- `2006-01-01` refers to the first day of 2006.
- `2006-02-last` refers to February 28, 2006.
- `\year-\month-\day` refers to today.
- `2006-01-01+2` refser to January 3, 2006.
- `\year-\month-\day+1` refers to tomorrow.
- `\year-\month-\day+-1` refers to yesterday.

The conversion method is taken from the English Wikipedia entry on Julian days.

Example: `\pgfcalendardatetojulian{2007-01-14}{\mycount}` sets `\mycount` to 2454115.

\pgfcalendarjuliantodate{⟨*Julian day*⟩}{⟨*year macro*⟩}{⟨*month macro*⟩}{⟨*day macro*⟩}

This command converts a Julian day number to an ISO-date. The ⟨*Julian day*⟩ must be a number or TeX counter, the ⟨*year macro*⟩, ⟨*month macro*⟩ and ⟨*day macro*⟩ must be TeX macro names. They will be set to numbers representing the year, month, and day of the given Julian day in the Gregorian calendar.

The ⟨*year macro*⟩ will be assigned the year without leading zeros. Note that this macro will produce year 0 (as opposed to other calendars, where year 0 does not exist). However, if you really need calendars for before the year 1, it is expected that you know what you are doing anyway.

The ⟨*month macro*⟩ gets assigned a two-digit number representing the month (with a leading zero, if necessary). Thus, the macro is set to `01` for January.

The ⟨*day macro*⟩ gets assigned a two-digit number representing the day of the month (again with a possible leading zero).

To convert a Julian day number to an ISO-date you use code like the following:

```
\pgfcalendardatetojulian{2454115}{\myyear}{\mymonth}{\myday}
\edef\isodate{\myyear-\mymonth-\myday}
```

The above code sets `\isodate` to `2007-01-14`.

\pgfcalendarjuliantoweekday{⟨*Julian day*⟩}{⟨*week day counter*⟩}

This command converts a Julian day to a week day by computing the day modulo 7. The ⟨*week day counter*⟩ must be a TeX counter. It will be set to 0 for a Monday, to 1 for a Tuesday, and so on.

Example: `\pgfcalendarjuliantoweekday{2454115}{\mycount}` sets `\mycount` to 6.

35.1.2 Checking Dates

\pgfcalendarifdate{⟨*date*⟩}{⟨*tests*⟩}{⟨*code*⟩}{⟨*else code*⟩}

This command is used to execute code based on properties of ⟨*date*⟩. The ⟨*date*⟩ must be a date in ISO-format. For this date, the ⟨*tests*⟩ are checked (to be detailed later) and if one of the tests applied, the ⟨*code*⟩ is executed. If none of the tests applies, the ⟨*else code*⟩ is executed.

Example: `\pgfcalendarifdate{2007-02-07}{Wednesday}{Is a Wednesday}{Is not a Wednesday}` yields `Is a Wednesday`.

The ⟨*tests*⟩ is a comma-separated list of key-value pairs. The following are defined by default:

- `all` This test is passed by all dates.
- `Monday` This test is passed by all dates that are Mondays.
- `Tuesday` as above.
- `Wednesday` as above.
- `Thursday` as above.
- `Friday` as above.
- `Saturday` as above.
- `Sunday` as above.
- `workday` Passed by Mondays, Tuesdays, Wednesdays, Thursdays, and Fridays.
- `weekend` Passed Saturdays and Sundays.
- `equals=`⟨*reference*⟩ The ⟨*reference*⟩ can be in one of two forms: Either, it is a full ISO format date like `2007-01-01` or the year may be missing as in `12-31`. In the first case, the test is passed if ⟨*date*⟩ is the same as ⟨*reference*⟩. In the second case, the test is passed if the month and day part of ⟨*date*⟩ is the same as ⟨*reference*⟩.

  For example, the test `equals=2007-01-10` will only be passed by this particular date. The test `equals=05-01` will be passed by every first of May on any year.
- `at least=`⟨*reference*⟩ This test works similarly to the `equals` test, only it is checked whether ⟨*date*⟩ is equal to ⟨*reference*⟩ or to any later date. Again, the ⟨*reference*⟩ can be a full date like `2007-01-01` or a short version like `07-01`. For example, `at least=07-01` is true for every day in the second half of any year.

- at most=⟨*reference*⟩ as above.
- between=⟨*start reference*⟩ and ⟨*end refernce*⟩ This test checks whether the current date lies between the two given reference dates. Both full and short version may be given.

  For example between=2007-01-01 and 2007-02-28 is true for the days in January and February of 2007.

  For another example, between=05-01 and 05-07 is true for the days of the first week of May of any year.
- day of month=⟨*number*⟩ Passed by the day of month of the ⟨*date*⟩ is ⟨*number*⟩. For example, the test day of month=1 is passed by every first of every month.
- end of month=⟨*number*⟩ Passed by the day of month of the ⟨*date*⟩ that is ⟨*number*⟩ from the end of the month. For example, the test end of month=1 is passed by the last day of every month, the test end of month=2 is passed by the second last day of every month. If ⟨*number*⟩ is omitted, it is assumed to be 1.

In addition to the above checks, you can also define new checks. To do so, you must add a new key to the key-value group pgfcalendar using \define@key. The job of the code of this new key is to possibly set the TEX-if \ifpgfcalendarmatches to true (if it is already true, no action should be taken) to indicate that the ⟨*date*⟩ passes the test setup by this new key.

In order to perform the test, the key code needs to know the date that should be checked. The date is available through a macro, but a whole bunch of additional information about this date is also available through the following macros:

- \pgfcalendarifdatejulian is the Julian day number of the ⟨*date*⟩ to be checked.
- \pgfcalendarifdateweekday is the weekday of the ⟨*date*⟩ to be checked.
- \pgfcalendarifdateyear is the year of the ⟨*date*⟩ to be checked.
- \pgfcalendarifdatemonth is the month of the ⟨*date*⟩ to be checked.
- \pgfcalendarifdateday is the day of month of the ⟨*date*⟩ to be checked.

For example, let us define a new key that checks whether the ⟨*date*⟩ is a Workers day (first of May). This can be done as follows:

```
\define@key{pgfcalendar}{workers day}[]
{
  \ifnum\pgfcalendarifdatemonth=5\relax
    \ifnum\pgfcalendarifdateday=1\relax
      \pgfcalendarmatchestrue
    \fi
  \fi
}
```

35.1.3 Typesetting Dates

\pgfcalendarweekdayname{⟨*week day number*⟩}

This command expands to a textual representation of the day of week, given by the ⟨*week day number*⟩. Thus, \pgfcalendarweekdayname{0} expands to Monday if the current language is English and to Montag if the current language is German, and so on. See Section 35.1.4 for more details on translations.

Example: \pgfcalendarweekdayname{2} yields Wednesday.

\pgfcalendarweekdayshortname{⟨*week day number*⟩}

This command works similiarly to the previous command, only an abbreviated version of the week day is produced.

Example: \pgfcalendarweekdayshortname{2} yields Wed.

\pgfcalendarmonthname{⟨*month number*⟩}

This command expands to a textual representation of the month, which is given by the ⟨*month number*⟩.

Example: \pgfcalendarmonthname{12} yields December.

`\pgfcalendarmonthshortname{`⟨*month number*⟩`}`

As above, only an abbreviated version is produced.

Example: `\pgfcalendarmonthshortname{12}` yields `Dec`.

35.1.4 Localization

All textual representations of week days or months (like "Monday" or "February") are wrapped with `\translate` commands from the `translator` package (it this package is not loaded, no translation takes place). Furthermore, the `pgfcalendar` package will try to load the `translator-months-dictionary`, if the `translator` package is loaded.

The net effect of all this is that all dates will be translated to the current language setup in the `translator` package. See the documentation of this package for more details.

35.2 Typesetting Calendars

`\pgfcalendar{`⟨*prefix*⟩`}{`⟨*start date*⟩`}{`⟨*end date*⟩`}{`⟨*rendering code*⟩`}`

This command can be used to typeset a calendar. It is a very general command, the actual work has to be done by giving clever implementations of ⟨*rendering code*⟩. Note that this macro need *not* be called inside a `{pgfpicture}` environment (even though it typically will be) and you can use it to typeset calendars in normal TEX or using packages other than PGF.

Basic typesetting process. A calendar is typeset as follows: The ⟨*start date*⟩ and ⟨*end date*⟩ specify a range of dates. For each date in this range the ⟨*rendering code*⟩ is executed with certain macros setup to yield information about the *current date* (the current date in the enumeration of dates of the range). Typically, the ⟨*rendering code*⟩ places nodes inside a picture, but it can do other things as well. Note that it is also the job of the ⟨*rendering code*⟩ to position the calendar correctly.

The different calls of the ⟨*rending code*⟩ are not surrounded by TEX groups (though you can do so yourself, of course). This means that settings can accumulate between different calls, which is often desirable and useful.

Information about the current date. Inside the ⟨*rendering code*⟩, different macros can be access:

- `\pgfcalendarprefix` The ⟨*prefix*⟩ parameter. This prefix is recomended for nodes inside the calendar, but you have to use it yourself explicitly.
- `\pgfcalendarbeginiso` The ⟨*start date*⟩ of range being typeset in ISO format (like 2006-01-10).
- `\pgfcalendarbeginjulian` Julian day number of ⟨*start date*⟩.
- `\pgfcalendarendiso` The ⟨*end date*⟩ of range being typeset in ISO format.
- `\pgfcalendarendjulian` Julian day number of ⟨*end date*⟩.
- `\pgfcalendarcurrentjulian` This TEX count holds the Julian day number of day currently begin rendered.
- `\pgfcalendarcurrentweekday` The weekday (a number with zero representing Monday) of the current date.
- `\pgfcalendarcurrentyear` The year of the current date.
- `\pgfcalendarcurrentmonth` The month of the current date (always two digits with a leading zero, if necessary).
- `\pgfcalendarcurrentday` The day of month of the current date (alwyas two digits).

The `\ifdate` command. Inside the `\pgfcalendar` the macro `\ifdate` is available locally:

`\ifdate{`⟨*tests*⟩`}{`⟨*code*⟩`}{`⟨*else code*⟩`}`

This command has the same effect as calling `\pgfcalendarifdate` for the current date.

Examples. In a first example, let us create a very simple calendar: It just lists the dates in a certain range.

20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10

```
\pgfcalendar{cal}{2007-01-20}{2007-02-10}{\pgfcalendarcurrentday\ }
```

Let us now make this a little more interesting: Let us add a line break after each Sunday.

20 21
22 23 24 25 26 27 28
29 30 31 01 02 03 04
05 06 07 08 09 10

```
\pgfcalendar{cal}{2007-01-20}{2007-02-10}
{
  \pgfcalendarcurrentday\
  \ifdate{Sunday}{\par}{}
}
```

We now want to have all Mondays to be aligned on a column. For this, different approaches work. Here is one based positioning each day horizontally using a skip.

					20	21
22	23	24	25	26	27	28
29	30	31	01	02	03	04
05	06	07	08	09	10	

```
\pgfcalendar{cal}{2007-01-20}{2007-02-10}
{%
  \leavevmode%
  \hbox to0pt{\hskip\pgfcalendarcurrentweekday cm\pgfcalendarcurrentday\hss}%
  \ifdate{Sunday}{\par}{}%
}
```

Let us now typeset two complete months.

January

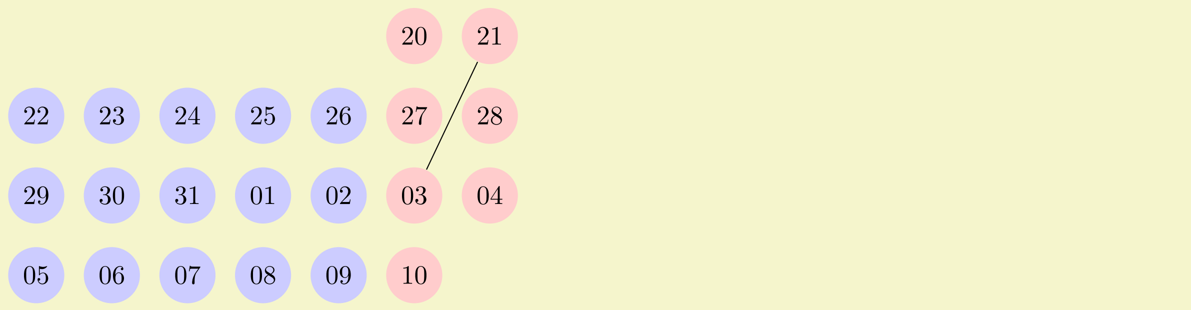
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

February

			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28				

```
\pgfcalendar{cal}{2007-01-01}{2007-02-28}{%
  \ifdate{day of month=1}{
    \par\bigskip\hbox to7.5cm{\itshape\hss\pgfcalendarshorthand mt\hss}\par
  }{}%
  \leavevmode%
  {%
    \ifdate{weekend}{\color{black!50}}{\color{black}}%
    \hbox to0pt{%
      \hskip\pgfcalendarcurrentweekday cm%
      \hbox to1cm{\hss\pgfcalendarshorthand d-}\hss%
    }%
  }%
  \ifdate{Sunday}{\par}{}%
}
```

For our final example, we use a `{tikzpicture}`.



```
\begin{tikzpicture}
  \pgfcalendar{cal}{2007-01-20}{2007-02-10}{%
    \ifdate{workday}{\tikzstyle{filling}=[fill=blue!20]}{\tikzstyle{filling}=[fill=red!20]}
    \node (\pgfcalendarsuggestedname) at (\pgfcalendarcurrentweekday,0)
      [anchor=base,circle,filling] {\pgfcalendarcurrentday};
    \ifdate{Sunday}{\pgftransformyshift{-3em}}{}%
  }
  \draw (cal-2007-01-21) -- (cal-2007-02-03);
\end{tikzpicture}
```

`\pgfcalendarshorthand{`⟨*kind*⟩`}{`⟨*representation*⟩`}`

This command can be used inside a `\pgfcalendar`, where it will expand to a representation of the current day, month, year or day of week, depending on whether ⟨*kind*⟩ is `d`, `m`, `y` or `w`. The ⟨*representation*⟩ can be one of the following: `-`, `=`, `0`, `.`, and `t`. They have the following meanings:

- The minus sign selects the shortest numerical representation possible (no leading zeros).
- The equal sign also selects the shortest numerical representation, but a space is added to single digit days and months (thereby ensuring that they have the same length as other days).
- The zero digit selects a two-digit numerical representation for days and months. For years it is allowed, but has no effect.
- The letter `t` selects a textual representation.
- The dot selects an abbreviated textual representation.

Normally, you should say `\let\%=\pgfcalendarshorthand` locally, so that you can write `\%wt` instead of the much more cumbersome `\pgfcalendarshorthand{w}{t}`.

ISO form: 2007-01-20, long form: Saturday, January 20, 2007

```
\let\%=\pgfcalendarshorthand
\pgfcalendar{cal}{2007-01-20}{2007-01-20}
{ ISO form: \%y0-\%m0-\%d0, long form: \%wt, \%mt \%d-, \%y0}
```

`\pgfcalendarsuggestedname`

This macro expands to a suggested name for nodes representing days in a calendar. If the ⟨*prefix*⟩ is empty, it expands to the empty string, otherwise it expands to the ⟨*prefix*⟩ of the calendar, followed by a hyphen, followed by the ISO format version of the date. Thus, when the date `2007-01-01` is typeset in a calendar for the prefix `mycal`, the macro expands to `mycal-2007-01-01`.

36 Page Management

This section describes the pgfpages packages. Although this package is not concerned with creating pictures, its implementation relies so heavily on PGF that it is documented here. Currently, pgfpages only works with LaTeX, but if you are adventurous, feel free to hack the code so that it also works with plain TeX.

The aim of pgfpages is to provide a flexible way of putting multiple pages on a single page *inside TeX*. Thus, pgfpages is quite different from useful tools like psnup or pdfnup insofar as it creates its output in a single pass. Furthermore, it works uniformly with both latex and pdflatex, making it easy to put multiple pages on a single page without any fuss.

A word of warning: *using pgfpages will destroy hyperlinks*. Actually, the hyperlinks are not destroyed, only they will appear at totally wrong positions on the final output. This is due to a fundamental flaw in the PDF specification: In PDF the bounding rectangle of a hyperlink is given in "absolute page coordinates" and translations or rotations do not affect them. Thus, the transformations applied by pgfpages to put the pages where you want them are (cannot, even) be applied to the coordinates of hyperlinks. It is unlikely that this will change in the foreseeable future.

36.1 Basic Usage

The internals of pgfpages are complex since the package can do all sorts of interesting tricks. For this reason, so-called *layouts* are predefined that setup all option in appropriate ways.

You use a layout as follows:

```
\documentclass{article}

\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The layout 2 on 1 puts two pages on a single page. The option a4paper tells pgfpages that the *resulting* page (called the *physical* page in the following) should be a4paper and it should be landscape (which is quite logical since putting two portrait pages next to each other gives a landscape page). Normally, the *logical* pages, that is, the pages that TeX "thinks" that it is typesetting, will have the same sizes, but this need not be the case. pgfpages will automatically scale down the logical pages such that two logical pages fit next to each other inside a DIN A4 page.

The border shrink tells pgfpages that it should add an additional 5mm to the shrinking such that a 5mm-wide border is shown around the resulting logical pages.

As a second example, let us put two pages produced by the BEAMER class on a single page:

```
\documentclass{beamer}

\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,border shrink=5mm]

\begin{document}
\begin{frame}
  This text is shown at the top.
\end{frame}
\begin{frame}
  This text is shown at the bottom.
\end{frame}
\end{document}
```

Note that we do not use the landscape option since BEAMER's logical pages are already in landscape mode and putting two landscape pages on top of each other results in a portrait page. However, if you had used the 4 on 1 layout, you would have had to add landscape once more, using the 8 on 1 you must not, using 16 on 1 you need it yet again. And, no, there is no 32 on 1 layout.

Another word of caution: *using pgfpages will produce wrong page numbers in the .aux file*. The reason is that TeX instantiates the page numbers when writing an .aux file only when the physical page is shipped out. Fortunately, this problem is easy to fix: First, typeset our file normally without using the \pgfpagesuselayout command (just put the comment marker % before it) Then, rerun TeX with the

\pgfpagesuselayout command included and add the command \nofiles. This command ensures that the .aux file is not modified, which is exactly what you want. So, to typeset the above example, you should actually first TEX the following file:

```
\documentclass{article}

\usepackage{pgfpages}
%%\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]
%%\nofiles

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

and then typeset

```
\documentclass{article}

\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]
\nofiles

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The final basic example is the `resize to` layout (it works a bit like a hypothetical `1 on 1` layout). This layout resizes the logical page such that is fits the specified physical size. Since this does not change the page numbering, you need not worry about the .aux files with this layout. For example, adding the following lines will ensure that the physical output will fit on DIN A4 paper:

```
\usepackage{pgfpages}
\pgfpagesuselayout{resize to}[a4paper]
```

This can be very useful when you have to handle lots of papers that are typeset for, say, letter paper and you have an A4 printer or the other way round. For example, the following article will be fit for printing on letter paper:

```
\documentclass[a4paper]{article}
%% a4 is currently the logical size and also the physical size

\usepackage{pgfpages}
\pgfpagesuselayout{resize to}[letterpaper]
%% a4 is still the logical size, but letter is the physical one

\begin{document}
  \title{My Great Article}
...
\end{document}
```

36.2 The Predefined Layouts

This section explains the predefined layouts in more detail. You select a layout using the following command:

\pgfpagesuselayout{⟨*layout*⟩}[⟨*options*⟩]

Installs the specified ⟨*layout*⟩ with the given ⟨*options*⟩ set. The predefined layouts and their permissible options are explained below.

If this function is called multiple times, only the last call "wins." You can thereby overwrite any previous settings. In particular, layouts *do not* accumulate.

Example: `\pgfpagesuselayout{resize to}[a4paper]`

\pgfpagesuselayout{resize to}[⟨*options*⟩]

This layout is used to resize every logical page to a specified physical size. To determine the target size, the following options may be given:

- physical paper height=⟨*size*⟩ sets the height of the physical pape size to ⟨*size*⟩.
- physical paper width=⟨*size*⟩ sets the width of the physical pape size to ⟨*size*⟩.
- a0paper sets the physical page size to DIN A0 paper.
- a1paper sets the physical page size to DIN A1 paper.
- a2paper sets the physical page size to DIN A2 paper.
- a3paper sets the physical page size to DIN A3 paper.
- a4paper sets the physical page size to DIN A4 paper.
- a5paper sets the physical page size to DIN A5 paper.
- a6paper sets the physical page size to DIN A6 paper.
- letterpaper sets the physical page size to the American letter paper size.
- legalpaper sets the physical page size to the American legal paper size.
- executivepaper sets the physical page size to the American executive paper size.
- landscape swaps the height and the width of the physical paper.
- border shrink=⟨*size*⟩ additionally reduces the size of the logical page on the physical page by ⟨*size*⟩.

`\pgfpagesuselayout{2 on 1}[⟨options⟩]`

Puts two logical pages alongside each other on each physical page if the logical height is larger than the logical width (logical pages are in portrait mode). Otherwise, two logical pages are put on top of each other (logical pages are in landscape mode). When using this layout, it is advisable to use the `\nofiles` command, but this is not done automatically.

The same ⟨*options*⟩ as for the `resize to` layout an be used, plus the following option:

- odd numbered pages right places the first page on the right.

`\pgfpagesuselayout{4 on 1}[⟨options⟩]`

Puts four logical pages on a single physical page. The same ⟨*options*⟩ as for the `resize to` layout an be used.

`\pgfpagesuselayout{8 on 1}[⟨options⟩]`

Puts eight logical pages on a single physical page. As for `2 on 1`, the orientation depends on whether the logical pages are in landscape mode or in portrait mode.

`\pgfpagesuselayout{16 on 1}[⟨options⟩]`

This is for the CEO.

`\pgfpagesuselayout{rounded corners}[⟨options⟩]`

This layout adds "rounded corners" to every page, which, supposedly, looks nicer during presentations with projectors (personally, I doubt this). This is done by (possibly) resizing the page to the physical page size. Then four black rectangles are drawn in each corner. Next, a clipping region is set up that contains all of the logical page except for little rounded corners. Finally, the logical page is draw, clipped against the clipping region.

Note that every logical page should fill its background for this to work.

In addition to the ⟨*options*⟩ that can be given to `resize to` the following options may be given.

- corner width=⟨*size*⟩ specifies the size of the corner.

```
\documentclass{beamer}
\usepackage{pgfpages}
\pgfpagesuselayout{rounded corners}[corner width=5pt]
\begin{document}
...
\end{document}
```

\pgfpagesuselayout{two screens with lagging second}[⟨*options*⟩]

This layout puts two logical pages alongside each other. The second page always shows what the main page showed on the previous physical page. Thus, the second page "lags behind" the main page. This can be useful when you have to projectors attached to your computer and can show different parts of a physical page on different projectors.

The following ⟨*options*⟩ may be given:

- second right puts the second page right of the main page. This will make the physical pages twice as wide as the logical pages, but it will retain the height.
- second left puts the second page left, otherwise it behave the same as second right.
- second bottom puts the second page below the main page. This make the physical pages twice as high as the logical ones.
- second top works like second bottom.

\pgfpagesuselayout{two screens with optional second}[⟨*options*⟩]

This layout works similarly to two screens with lagging second. The difference is that the contents of the second screen only changes when one of the commands \pgfshipoutlogicalpage{2}{⟨*box*⟩} or \pgfcurrentpagewillbelogicalpage{2} is called. The first puts the given ⟨*box*⟩ on the second page. The second specifies that the current page should be put there, once it is finished.

The same options as for two screens with lagging second may be given.

You can define your own predefined layouts using the following command:

\pgfpagesdeclarelayout{⟨*layout*⟩}{⟨*before actions*⟩}{⟨*after actions*⟩}

This command predefines a ⟨*layout*⟩ that can later be installed using the \pgfpagesuselayout command.

When \pgfpagesuselayout{⟨*layout*⟩}[⟨*options*⟩] is called, the following happens: First, the ⟨*before actions*⟩ are executed. They can be used, for example, to setup default values for keys. Next, \setkeys{pgfpagesuselayoutoption}{⟨*options*⟩} is executed. Finally, the ⟨*after actions*⟩ are executed.

Here is an example:

```
\pgfpagesdeclarelayout{resize to}
{
  \def\pgfpageoptionborder{0pt}
}
{
  \pgfpagesphysicalpageoptions
  {%
    logical pages=1,%
    physical height=\pgfpageoptionheight,%
    physical width=\pgfpageoptionwidth%
  }
  \pgfpageslogicalpageoptions{1}
  {%
    resized width=\pgfphysicalwidth,%
    resized height=\pgfphysicalheight,%
    border shrink=\pgfpageoptionborder,%
    center=\pgfpoint{.5\pgfphysicalwidth}{.5\pgfphysicalheight}%
  }%
}
```

36.3 Defining a Layout

If none of the predefined layouts meets your problem or if you wish to modify them, you can create layouts from scratch. This section explains how this is done.

Basically, pgfpages hooks into TEX's \shipout function. This function is called whenever TEX has completed typesetting a page and wishes to send this page to the .dvi or .pdf file. The pgfpages package redefines this command. Instead of sending the page to the output file, pgfpages stores it in an internal box and then acts as if the page had been output. When TEX tries to output the next page using \shipout, this call is once more intercepted and the page is stored in another box. These boxes are called *logical pages*.

At some point, enough logical pages have been accumulated such that a *physical page* can be output. When this happens, `pgfpages` possibly scales, rotates, and translates the logical pages (and possibly even does further modifications) and then puts them at certain positions of the *physical* page. Once this page is fully assembled, the "real" or "original" `\shipout` is called to send the physical page to the output file.

In reality, things are slightly more complicated. First, once a physical page has been shipped out, the logical pages are usually voided, but this need not be the case. Instead, it is possible that certain logical page just retain their contents after the physical page has been shipped out and these pages need not be filled once more before a physical shipout can occur. However, the contents of these logical pages can still be changed using special commands. It is also possible that after a shipout certain logical pages are filled with the contents of *other* logical pages.

A *layout* defines for each logical page where it will go on the physical page and which further modifications should be done. The following two commands are used to define the layout:

`\pgfpagesphysicalpageoptions{`⟨*options*⟩`}`

This command sets the characteristic of the "physical" page. For example, it is used to specify how many logical pages there are and how many logical pages must be accumulated before a physical page is shipped out. How each individual logical page is typeset is specified using the command `\pgfpageslogicalpageoptions`, described later.

Example: A layout for putting two portrait pages on a single landscape page:

```
\pgfpagesphysicalpageoptions
{%
  logical pages=2,%
  physical height=\paperwidth,%
  physical width=\paperheight,%
}

\pgfpageslogicalpageoptions{1}
{%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
\pgfpageslogicalpageoptions{2}
{%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.75\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
```

The following ⟨*options*⟩ may be set:

- `logical pages=`⟨*logical pages*⟩ specified how many logical pages there are, in total. These are numbered 1 to ⟨*logical pages*⟩.
- `first logical shipout=`⟨*first*⟩. See the the next option. By default, ⟨*first*⟩ is 1.
- `last logical shipout=`⟨*last*⟩. Together with the previous option, these two options define an interval of pages inside the range 1 to ⟨*logical pages*⟩. Only this range is used to store the pages that are shipped out by TEX. This means that after a physical shipout has just occured (or at the beginning), the first time TEX wishes to perform a shipout, the page to be shipped out is stored in logical page ⟨*first*⟩. The next time TEX performs a shipout, the page is stored in logical page ⟨*first*⟩ + 1 and so on, until the logical page ⟨*last*⟩ is also filled. Once this happens, a physical shipout occurs and the process starts once more.

  Note that logical pages that lie outside the interval between ⟨*first*⟩ and ⟨*last*⟩ are filled only indirectly or when special commands are used.

  By default, ⟨*last*⟩ equals ⟨*logical pages*⟩.
- `current logical shipout=`⟨*current*⟩ changes an internal counter such that TEX's next logical shipout will be stored in logical page ⟨*current*⟩.

  This option can be used to "warp" the logical page filling mechanism to a certain page. You can both skip logical pages and overwrite already filled logical pages. After the logical page ⟨*current*⟩ has been filled, the internal counter is incremented normally as if the logical page ⟨*current*⟩ had been "reached" normally. If you specify a ⟨*current*⟩ larger to ⟨*last*⟩, a physical shipout will occur after the logical page ⟨*current*⟩ has been filled.

- physical height=⟨*height*⟩ specifies the height of the physical pages. This height is typically different from the normal \paperheight, which is used by TeX for its typesetting and page breaking purposes.
- physical width=⟨*width*⟩ specifies the physical width.

\pgfpageslogicalpageoptions{⟨*logical page number*⟩}{⟨*options*⟩}

This command is used to specify where the logical page number ⟨*logical page number*⟩ will be placed on the physical page. In addition, this command can be used to install additional "code" to be executed when this page is put on the physical page.

The number ⟨*logical page number*⟩ should be between 1 and ⟨*logical pages*⟩, which has previously been installed using the \pgfpagesphysicalpageoptions command.

The following ⟨*options*⟩ may be given:

- center=⟨*pgf point*⟩ specifies the center of the logical page inside the physical page as a PGF-point. The origin of the coordinate system of the physical page is at the *lower* left corner.

```
\pgfpageslogicalpageoptions{1}
{% center logical page on middle of left side
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
}
```

- resized width=⟨*size*⟩ specifies the width that the logical page should have *at most* on the physical page. To achieve this width, the pages is scaled down appropriately *or more*. The "or more" part can happen if the resize height option is also used. In this case, the scaling is chosen such that both the specified height and width are met. The aspect ratio of a logical page is not modified.
- resized height=⟨*height*⟩ specifies the maximum height of the logical page.
- original width=⟨*width*⟩ specifies the width the TeX "thinks" that the logical page has. This width is \paperwidth at the point of invocation, by default. Note that setting this width to something different from \paperwidth does *not* change the \pagewidth during TeX's typesetting. You have to do that yourself.

  You need this option only for special logical pages that have a height or width different from the normal one and for which you will (later on) set these sizes yourself.
- original height=⟨*height*⟩ works like original width.
- scale=⟨*factor*⟩ scales the page by at least the given ⟨*factor*⟩. A ⟨*factor*⟩ of 0.5 will half the size of the page, a factor or 2 will double the size. "At least" means that if options like resize height are given and if the scaling required to meet that option is less than ⟨*factor*⟩, that other scaling is used instead.
- xscale=⟨*factor*⟩ scales the logical page along the x-axis by the given ⟨*factor*⟩. This scaling is done independently of any other scaling. Mostly, this option is useful for a factor of -1, which flips the page along the y-axis. The aspect ratio is not kept.
- yscale=⟨*factor*⟩ works like xscale, only for the y-axis.
- rotation=⟨*degree*⟩ rotates the page by ⟨*degree*⟩ around its center. Use a degree of 90 or -90 to go from portrait to landscape and back. The rotation need not be a multiple of 90.
- copy from=⟨*logical page number*⟩. Normally, after a physical shipout has occured, all logical pages are voided in a loop. However, if this option is given, the current logical page is filled with the contents of the old logical page number ⟨*logical page number*⟩.

  Example: Have logical page 2 retain its contents:

```
\pgfpageslogicalpageoptions{2}{copy from=2}
```

  Example: Let logical page 2 show what logical page 1 showed on the just-shipped-out physical page:

```
\pgfpageslogicalpageoptions{2}{copy from=1}
```

- border shrink=⟨*size*⟩ specifies an addition reduction of the size to which the page is page is scaled down.
- border code=⟨*code*⟩. When this option is given, the ⟨*code*⟩ is executed before the page box is inserted with a path preinstalled that is a rectangle around the current logical page. Thus, setting ⟨*code*⟩ to `\pgfstroke` draws a rectangle around the logical page. Setting ⟨*code*⟩ to `\pgfsetlinewidth{3pt}\pgfstroke` results in a thick (ugly) frame. Adding dashes and filling can result in arbitrarily funky and distracting borders.

  You can also call `\pgfdiscardpath` and add your own path construction code (for example to paint a rectangle with rounded corners). The coordinate system is setup in such a way that a rectangle starting at the origin and having the height and width of TEX-box 0 will result in a rectangle filling exactly the logical page currently being put on the physical page. The logical page is inserted *after* these commands have been executed.

  Example: Add a rectangle around the page:

  ```
  \pgfpageslogicalpageoptions{1}{border code=\pgfstroke}
  ```

- corner width=⟨*size*⟩ adds black "rounded corners" to the page. See the description of the predefined layout `rounded corners` on page 258.

36.4 Creating Logical Pages

Logical pages are created whenever a TEX thinks that a page is full and performs a `\shipout` command. This will cause `pgfpages` to store the box that was supposed to be shipped out internally until enough logical pages have been collected such that a physical shipout can occur.

Normally, whenever a logical shipout occurs that current page is stored in logical page number ⟨*current logical page*⟩. This counter is then incremented, until it is larger than ⟨*last logical shipout*⟩. You can, however, directly change the value of ⟨*current logical page*⟩ by calling `\pgfpagesphysicalpageoptions`.

Another way to set the contents of a logical page is to use the following command:

`\pgfpagesshipoutlogicalpage{`⟨*number*⟩`}`⟨*box*⟩

This command sets to logical page ⟨*number*⟩ to ⟨*box*⟩. The ⟨*box*⟩ should be the code of a TEX box command. This command does not influence the counter ⟨*current logical page*⟩ and does not cause a physical shipout.

```
\pgfpagesshipoutlogicalpage{0}\vbox{Hi!}
```

This command can be used to set the contents of logical pages that are normally not filled.

The final way of setting a logical page is using the following command:

`\pgfpagescurrentpagewillbelogicalpage{`⟨*number*⟩`}`

When the current TEX page has been typeset, it will be become the given logical page ⟨*number*⟩. This command "interrupts" the normal order of logical pages, that is, it behaves like the previous command and does not update the ⟨*current logical page*⟩ counter.

```
\pgfpagesuselayout{two screens with optional second}
...
Text for main page.
\clearpage

\pgfpagescurrentpagewillbelogicalpage{2}
Text that goes to second page
\clearpage

Text for main page.
```

37 Extended Color Support

This section documents the package `xxcolor`, which is currently distributed as part of PGF. This package extends the `xcolor` package, written by Uwe Kern, which in turn extends the `color` package. I hope that the commands in `xxcolor` will some day migrate to `xcolor`, such that this package becomes superfluous.

The main aim of the `xxcolor` package is to provide an environment inside which all colors are "washed out" or "dimmed." This is useful in numerous situations and must typically be achieved in a roundabout manner if such an environment is not available.

`\begin{colormixin}{`⟨*mix-in specification*⟩`}`
⟨*environment contents*⟩
`\end{colormixin}`

The mix-in specification is applied to all colors inside the environment. At the beginning of the environment, the mix-in is applied to the current color, i. e., the color that was in effect before the environment started. A mix-in specification is a number between 0 and 100 followed by an exclamation mark and a color name. When a `\color` command is encountered inside a mix-in environment, the number states what percentage of the desired color should be used. The rest is "filled up" with the color given in the mix-in specification. Thus, a mix-in specification like `90!blue` will mix in 10% of blue into everything, whereas `25!white` will make everything nearly white.

Red text, washed-out red text, washed-out blue text, dark washed-out blue text, dark washed-out green text, back to washed-out blue text, and back to red.

```
\begin{minipage}{3.5cm}\raggedright
\color{red}Red text,%
\begin{colormixin}{25!white}
  washed-out red text,
  \color{blue} washed-out blue text,
  \begin{colormixin}{25!black}
    dark washed-out blue text,
    \color{green} dark washed-out green text,%
  \end{colormixin}
  back to washed-out blue text,%
\end{colormixin}
and back to red.
\end{minipage}%
```

Note that the environment only changes colors that have been installed using the standard LaTeX `\color` command. In particular, the colors in images are not changed. There is, however, some support offered by the commands `\pgfuseimage` and `\pgfuseshading`. If the first command is invoked inside a `colormixin` environment with the parameter, say, `50!black` on an image with the name `foo`, the command will first check whether there is also a defined image with the name `foo.!50!black`. If so, this image is used instead. This allows you to provide a different image for this case. If you nest `colormixin` environments, the different mix-ins are all appended. For example, inside the inner environment of the above example, `\pgfuseimage{foo}` would first check whether there exists an image named `foo.!50!white!25!black`.

`\colorcurrentmixin`

Expands to the current accumulated mix-in. Each nesting of a `colormixin` adds a mix-in to this list.

!75!white should be "!75!white"
!75!black!75!white should be "!75!black!75!white"
!50!white!75!black!75!white should be "!50!white!75!black!75!white"

```
\begin{minipage}{\linewidth-6pt}\raggedright
\begin{colormixin}{75!white}
  \colorcurrentmixin\ should be ``!75!white''\par
  \begin{colormixin}{75!black}
    \colorcurrentmixin\ should be ``!75!black!75!white''\par
    \begin{colormixin}{50!white}
      \colorcurrentmixin\ should be ``!50!white!75!black!75!white''\par
    \end{colormixin}
  \end{colormixin}
\end{colormixin}
\end{minipage}
```

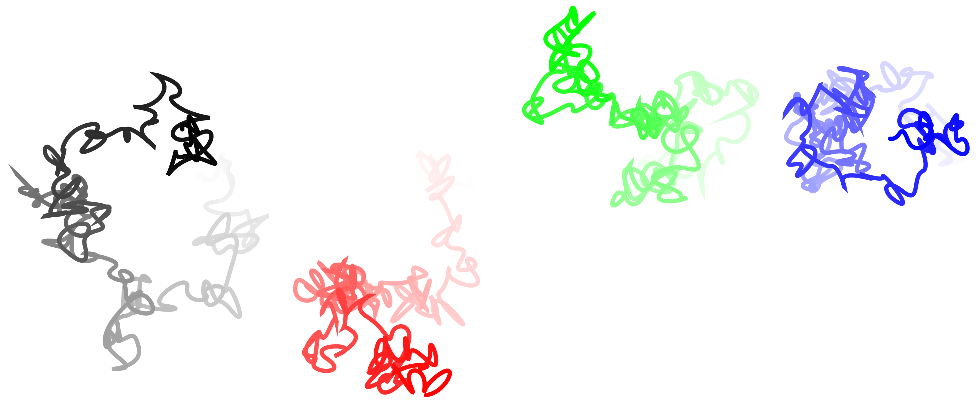
Part VI

Mathematical Engine

by Mark Wibrow and Till Tantau

PGF comes with its own mathematical engine. The job of this engine is to support mathematical operations like addition, subtraction, multiplication and division, using both integers and non-integers, but also functions such as square-roots, sine, cosine, and generate pseudo-random numbers.

Mostly, you will use the mathematical facilities of PGF indirectly, namely when you write a coordinate like `(5cm*3,6cm/4)`, but the mathematical engine can also be used independently of PGF and Ti*k*Z.



```
\pgfmathsetseed{1}
\foreach \col in {black,red,green,blue}
{
  \begin{tikzpicture}[x=10pt,y=10pt,ultra thick,baseline,cap=round]
    \coordinate (current point) at (0,0);
    \coordinate (old velocity) at (0,0);
    \coordinate (new velocity) at (rand,rand);

    \foreach \i in {0,1,...,100}
    {
      \draw[\col!\i] (current point)
      .. controls ++([scale=-1]old velocity) and
                  ++(new velocity) .. ++(rand,rand)
         coordinate (current point);
      \coordinate (old velocity) at (new velocity);
      \coordinate (new velocity) at (rand,rand);
    }
  \end{tikzpicture}
}
```

38 Design Principles

PGF needs to perform many computations while typesetting a picture. For this, PGF relies on a mathematical engine, which can also be used independently of PGF, but which is distributed as part of the PGF package nevertheless. Basically, the engine provides a parsing mechanism similar to the CALC package so that expressions like `2*3cm+5cm` can be parsed; but the PGF engine is more powerful and can be extended and enhanced.

PGF provides enhanced functionality, which permits the parsing of mathematical operations involving integers and non-integers with or without units. Furthermore, various functions, including trigonometric functions and random number generators can also be parsed (see Section 39.1). The CALC macros `\setlength` and friends have PGF versions which can parse these operations and functions (see Section 39.1). Additionally, each operation and function has an independent PGF command associated with it (see Section 40), and can be accessed outside the parser.

The mathematical engine of PGF is implicitly used whenever you specify a number or dimension in a higher-level macro. For instance, you can write `\pgfpoint{2cm+4cm/2}{3cm*sin(30)}` or suchlike. However, the mathematical engine can also be used independently of the PGF core, that is, you can also just load it to get access to a mathematical parser.

38.1 Loading the Mathematical Engine

The mathematical engine of PGF is loaded automatically by PGF, but if you wish to use the mathematical engine but you do not need PGF itself, you can load the following package:

```
\usepackage{pgfmath} % LaTeX
\input pgfmath.tex   % plain TeX
\usemodule[pgfmath]  % ConTeXt
```

This command will load the mathematical engine of PGF, but not PGF itself. It defines commands like `\pgfmathparse`.

38.2 Layers of the Mathematical Engine

Like PGF itself, the mathematical engine is also structured into different layers:

1. The top layer, which you will typically use directly, provides the command `\pgfmathparse`. This command parses a mathematical expression and evaluates it.

   Additionally, the top layer also defines some additional functions similar to the macros of the `calc` package for setting dimensions and counters. These macros are just wrappers around the `\pgfmathparse` macro.

2. The calculation layer provides macros for performing one specific computation like computing a reciprocal or a multiplication. The parser uses these macros for the actual computation.

3. The implementation layer provides the actual implementations of the computations. These can be changed (and possibly be made more efficient) without affecting the higher layers.

38.3 Efficiency and Accuracy of the Mathematical Engine

Currently, the mathematical algorithms are all implemented in TEX. This poses some intriguing programming challenges as TEX is a typesetting language not a mathematical one, and as with any programming language, there is a trade-off between accuracy and efficiency. If you find the level of accuracy insufficient for you purposes, you will have to replace the algorithms in the implementation layer.

All the fancy mathematical "bells-and-whistles" that the parser provides, come with an additional processing cost, and in some instances, such as simply setting a length to `1cm`, with no other operations involved, the addition processing time is undesirable. To overcome this, the following feature is implemented: when no mathematical operations are required, the value in ⟨*expression*⟩ can be preceded by `+`. This will bypass the parsing process and the assignment will be orders of magnitude faster. This feature *only* works with the macros for setting registers described in Section 39.1.

```
\pgfmathsetlength\mydimen{1cm}  % parsed     : slower.
\pgfmathsetlength\mydimen{+1cm} % not parsed : much faster.
```

39 Evaluating Mathematical Expressions

The easiest way of using PGF's mathematical engine is to provide a mathematical expression given in the usual infix notation (such as `1cm+4*2cm/5.5` or `2*3+3*sin(30)`). This expression can be parsed by the mathematical engine and the result be placed in a dimension register, a counter, or a macro. Supported are infix mathematical operations involving integers and non-integers, with or without units.

It should be noted that all calculations must not exceed ±16383.99999 at *any* point, because the underlying algorithms relie on TEX dimensions. This means that many of the underlying algorithms are necessarily approximate. It also means that some of the algorithms are not very fast. TEX is, after all, a typesetting language and not ideally suited to relatively advanced mathematical operations. However, it is possible to change the algorithms as described in Section 41.

In the present section, the high-level macros for parsing an expression are explained first, then the syntax for expression is explained.

39.1 Commands for Parsing Expressions

The basic command for invoking the parser of PGF's mathematical engine is the following:

`\pgfmathparse{`⟨*expression*⟩`}`

This macro parses ⟨*expression*⟩ and returns the result without units in the macro `\pgfmathresult`.

Example: `\pgfmathparse{2pt+3.5pt}` will set `\pgfmathresult` to the text `5.5`.

In the following, the special properties of this command are explained. The exact syntax of mathematical expressions is explained in Section 39.2. Note that unlike the rest of the manual, the examples show the result of the calculation (that is, the value of the macro `\pgfmathresult`), not what is displayed in the document.

- The result stored in the macro `\pgfmathresult` is a decimal *without units*. This is true regardless of whether the ⟨*expression*⟩ contains any unit specification. But, any units specified will be converted to points first.

5.4

```
\pgfmathparse{2pt+3.4pt}
```

153.64468

```
\pgfmathparse{2cm+3.4cm}
```

- If no units are specified *at any point* in the expression, the result will be multiplied by the value in `\pgfmathresultunitscale`, which can be a number or a dimension (which will be converted to points). By default it is set to 1, but can be changed with `\pgfmathsetresultunitscale`. Note that the result will still be a number *without* units.

5.4

```
\pgfmathparse{2pt+3.4pt}
```

153.64464

```
\pgfmathsetresultunitscale{1cm}
\pgfmathparse{2+3.4}
```

- You can check whether an expression contained a unit using the TEX-if `\ifpgfmathunitsdeclared`. After a call of `\pgfmathparse` this if will be true exactly if some unit was encountered in the expression.
- The parser handles numbers with or without units regardless of the operation.

1.32544

```
\pgfmathparse{54pt/3cm*2.1}
```

- the parser can cope with TEX registers, including those preceded by `\the`.

42.34

```
\pgf@x=12.34pt
\c@pgf@counta=5
\pgfmathparse{\pgf@x+\c@pgf@counta*6}
```

113.56

```
\pgf@x=56.78pt
\pgfmathparse{\pgf@x+\the\pgf@x}
```

- TeX dimension registers can be multiplied without the `*` operator by preceding them with a number (*not* a function), or a count register.

```
45.0
```

```
\c@pgf@counta=-4
\pgf@x=10pt
\pgfmathparse{.5\pgf@x-\c@pgf@counta\pgf@x}%
```

- Parenthesis can be used to group operations.

```
13.5
```

```
\pgfmathparse{(4pt+0.5)*3}
```

- functions are recognized, so it is possible to parse `sin(.5*pi r)*60`, which means "the sine of 0.5 times π radians, multiplied by 60". The argument of most functions can be any expression.

```
59.99908
```

```
\pgfmathparse{sin(pi/2 r)*60}
```

- Scientific notation in the form `1.234e+4` is recognised (but the restriction on the range of values still applies). The exponent symbol can be upper or lower case (i.e., `E` or `e`).

```
0.01234
```

```
\pgfmathparse{1.234567891e-2}
```

```
12345.67891
```

```
\pgfmathparse{1.234567891e4}
```

`\pgfmathqparse{`⟨*expression*⟩`}`

This macro is similar to `\pgfmathparse`: it parses ⟨*expression*⟩ and returns the result in the macro `\pgfmathresult`. It differs in two respects. Firstly, `\pgfmathqparse` does not parse functions or scientific notation. Secondly, numbers in ⟨*expression*⟩ *must* specify a TeX unit (except in such instances as `0.5\pgf@x`), which greatly simplifies the problem of parsing of non-integers. As a result of these restrictions `\pgfmathqparse` is about twice as fast as `\pgfmathparse`. Note that the result will still be a number *without* units.

`\pgfmathsetresultunitscale{`⟨*number or dimension*⟩`}`

Sets the value in `\pgfmathresultunitscale`, which scales the result of an expression parsed with `\pgfmathparse`, if that expression contains no units *at any point*. The argument can be an integer, non-integer or a dimension, but the result will still be a number *without* units. Note, that this will affect `\pgfmathsetlength` and friends, but not if the expression starts with `+` (which switches parsing off). By default the value in `\pgfmathresultunitscale` is 1.

Instead of the `\pgfmathparse` macro you can also wrapper commands, whose usage is very similar to their cousins in the CALC package. The only difference is that the expressions can be any expression that is handled by `\pgfmathparse`.

For all of the following commands, if ⟨*expression*⟩ starts with `+`, no parsing is done and a simple assignment or increment is done using normal TeX assignments or increments. This will be orders of magnitude faster than calling the parser.

`\pgfmathsetlength{`⟨*dimension register*⟩`}{`⟨*expression*⟩`}`

Sets the length of the TeX ⟨*dimension register*⟩, to the value (in points) specified by ⟨*expression*⟩. The ⟨*expression*⟩ will be parsed using `\pgfmathparse`.

`\pgfmathaddtolength{`⟨*dimension register*⟩`}{`⟨*expression*⟩`}`

Adds the value (in points) of ⟨*expression*⟩ to the TeX ⟨*dimension register*⟩.

`\pgfmathsetcount{`⟨*count register*⟩`}{`⟨*expression*⟩`}`

Sets the value of the TeX ⟨*count register*⟩, to the *truncated* value specified by ⟨*expression*⟩.

`\pgfmathaddtocount{`⟨*count register*⟩`}{`⟨*expression*⟩`}`

Adds the *truncated* value of ⟨*expression*⟩ to the TeX ⟨*count register*⟩.

`\pgfmathsetcounter{`⟨*counter*⟩`}{`⟨*expression*⟩`}`

Sets the value of the ⟨*counter*⟩, to the *truncated* value specified by ⟨*expression*⟩.

`\pgfmathaddtocounter{⟨counter⟩}{⟨expression⟩}`

Adds the *truncated* value of ⟨*expression*⟩ to ⟨*counter*⟩.

`\pgfmathsetmacro{⟨macro⟩}{⟨expression⟩}`

Defines ⟨*macro*⟩ as the value of ⟨*expression*⟩. The result is a decimal *without* units.

39.2 Syntax for mathematical expressions

The syntax for the expressions recognized by `\pgfmathparse` and friends is based on the syntax recognized by MATLAB. The following operations and functions are currently recognized:

x `+` y

Adds y to x.

6.0
```
\pgfmathparse{4+2pt}
```

x `-` y

Subtracts y from x.

41.53899
```
\pgfmathparse{155.35-4cm}
```

x `*` y

Multiplies x by y.

17.78395
```
\pgfmathparse{3.9pt*4.56}
```

x `/` y

Divides x by y.

-1.85881
```
\pgfmathparse{-31.6pt/17}
```

x `^` y

Raises x to the power y. y should be an integer, but it can be negative.

27.98418
```
\pgfmathparse{2.3^4}
```

0.0625
```
\pgfmathparse{2^-4}
```

x `==` y

This evaluates to `1` if x equals y, or `0` if x does not equal y. Note that equalities (and inequalities) are evaluated left to right, and are only evaluated when another equality (or inequality) operator is scanned, or the end of the current group or parse is reached. So `5+4==3+2==9` results in `0` because `5+4` does not equal `3+2`, resulting in zero, and the second equality is therefore evaluating `0==9`.

1.0
```
\pgfmathparse{3*5 == 15}
```

x `>` y

This evaluates to `1` if x is greater than y, or `0` if x is smaller or equal to y.

1.0
```
\pgfmathparse{17 > 4.2*1.97+4}
```

x `<` y

This evaluates to `1` if x is smaller than y, or `0` if x is greater or equal to y.

0.0
```
\pgfmathparse{2 < -5.2/-3.6-2}
```

`mod(`x`,`y`)`

This evaluates x modulo y. This function cannot be nested inside itself or the functions `max`, `min` or `veclen`.

2.0
```
\pgfmathparse{mod(20,6)}
```

`max(`x`,`y`)`

This evaluates to the maximum of x or y. This function cannot be nested inside itself or the functions `min`, `mod` or `veclen`.

23.0
```
\pgfmathparse{max(17,23)}
```

`min(`x`,`y`)`

This evaluates to the minimum of x or y. This function cannot be nested inside itself or the functions `max`, `mod` or `veclen`.

17.0 `\pgfmathparse{min(17,23)}`

`abs(`x`)`

Evaluates the absolute value of x.

5.0 `\pgfmathparse{abs(-5)}`

-12.0 `\pgfmathparse{-abs(4*-3)}`

`round(`x`)`

Rounds x to the nearest integer. It uses "asymmetric half-up" rounding. So `1.5` is rounded to `2`, but `-1.5` is rounded to `-2` (*not* `0`).

2.0 `\pgfmathparse{round(32.5/17)}`

33.0 `\pgfmathparse{round(398/12)}`

`floor(`x`)`

Rounds x down to the nearest integer.

1.0 `\pgfmathparse{floor(32.5/17)}`

33.0 `\pgfmathparse{floor(398/12)}`

`ceil(`x`)`

Rounds x up to the nearest integer.

2.0 `\pgfmathparse{ceil(32.5/17)}`

34.0 `\pgfmathparse{ceil(398/12)}`

`exp(`x`)`

Maclaurin series for e^x.

2.7182 `\pgfmathparse{exp(1)}`

10.3806 `\pgfmathparse{exp(2.34)}`

`pow(`x`,`y`)`

Raises x to the power y. y should be an integer, but it can be negative.

128.0 `\pgfmathparse{pow(2,7)}`

`sqrt(`x`)`

Calculates $\sqrt{x}$.

3.16228 `\pgfmathparse{sqrt(10)}`

93.62389 `\pgfmathparse{sqrt(8765.432)}`

`veclen(`x`,`y`)`

Evaluates the Euclidean distance from `(0,0)` to `(`x`,`y`)`. This function cannot be nested inside itself, or the functions `max`, `min` or `mod`.

20.51811 `\pgfmathparse{veclen(15,14)}`

4.99994 `\pgfmathparse{veclen(3,4)}`

`pi`

The constant $\pi = 3.14159$.

3.14159 `\pgfmathparse{pi}`

179.99962 `\pgfmathparse{pi r}`

x `r`

This converts x from radians to degrees. Note that `r` will evaluate any preceding series of multiplication or division *before* conversion, but not other operations. So `3*4/6r` converts 2 radians to degrees, but `3-4+6r`, converts 6 radians to degrees and adds the result to `-1`.

179.99963 `\pgfmathparse{2*pi r-pi r}`

44.99924 `\pgfmathparse{2*pi/8 r}`

-59.99908 `\pgfmathparse{sin(3*pi/2r)*60}`

`rad(`x`)`

Convert x to radians. x is assumed to be in degrees.

1.57079 `\pgfmathparse{rad(90)}`

`deg(`x`)`

Convert x to degrees. x is assumed to be in radians.

269.999 `\pgfmathparse{deg(3*pi/2)}`

`sin(`x`)`

Sine of x. By employing the `r` operator, x can be in radians.

0.86603 `\pgfmathparse{sin(60)}`

0.86601 `\pgfmathparse{sin(pi/3 r)}`

`cos(`x`)`

Cosine of x. By employing the `r` operator, x can be in radians.

0.5 `\pgfmathparse{cos(60)}`

0.49998 `\pgfmathparse{cos(pi/3 r)}`

`tan(`x`)`

Tangent of x. By employing the `r` operator, x can be in radians.

1.0 `\pgfmathparse{tan(45)}`

1.0 `\pgfmathparse{tan(2*pi/8 r)}`

`asin(`x`)`

Arcsine of x. The result is in degrees.

44.99133 `\pgfmathparse{asin(0.7071)}`

`acos(`x`)`

Arccosine of x in degrees.

60.0 `\pgfmathparse{acos(0.5)}`

`atan(`x`)`

Arctangent of x in degrees.

45.0 `\pgfmathparse{atan(1)}`

`rnd`

Generates a pseudo-random number between 0 and 1.

0.32524 `\pgfmathparse{rnd}`

0.63 `\pgfmathparse{2*rnd}`

4.02074 `\pgfmathparse{-rnd+5}`

`rand`

Generates a pseudo-random number between -1 and 1.

0.80753 `\pgfmathparse{rand}`

8.9788 `\pgfmathparse{rand*15}`

40 Evaluating Mathematical Operations

Instead of parsing and evaluating complex expressions, you can also use the mathematical engine to evaluate a single mathematical operation. The macros used for these computations are described in the following.

40.1 Basic Operations and Functions

`\pgfmathadd{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as $\langle x\rangle + \langle y\rangle$.

`\pgfmathsubtract{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as $\langle x\rangle - \langle y\rangle$.

`\pgfmathmultiply{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as $\langle x\rangle \times \langle y\rangle$.

`\pgfmathdivide{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as $\langle x\rangle \div \langle y\rangle$.

`\pgfmathgreaterthan{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as 1.0 if $\langle x\rangle > \langle y\rangle$, but 0.0 otherwise.

`\pgfmathlessthan{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as 1.0 if $\langle x\rangle < \langle y\rangle$, but 0.0 otherwise.

`\pgfmathequalto{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` 1.0 if $\langle x\rangle = \langle y\rangle$, but 0.0 otherwise.

`\pgfmathround{`$\langle x\rangle$`}`

Defines `\pgfmathresult` as $\lfloor\langle x\rangle\rceil$. This uses asymmetric half-up rounding.

`\pgfmathfloor{`$\langle x\rangle$`}`

Defines `\pgfmathresult` as $\lfloor\langle x\rangle\rfloor$.

`\pgfmathceil{`$\langle x\rangle$`}`

Defines `\pgfmathresult` as $\lceil\langle x\rangle\rceil$.

`\pgfmathpow{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as $\langle x\rangle^{\langle y\rangle}$. $\langle y\rangle$ is expected to be an integer. But it can be negative.

`\pgfmathmod{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as $\langle x\rangle$ modulo $\langle y\rangle$.

`\pgfmathmax{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as the maximum of $\langle x\rangle$ or $\langle y\rangle$.

`\pgfmathmin{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as the minimum $\langle x\rangle$ or $\langle y\rangle$.

`\pgfmathabs{`$\langle x\rangle$`}`

Defines `\pgfmathresult` as absolute value of $\langle x\rangle$.

`\pgfmathreciprocal{`$\langle x\rangle$`}`

Defines `\pgfmathresult` as $1 \div \langle x\rangle$.

`\pgfmathexp{`$\langle x\rangle$`}`

Defines `\pgfmathresult` as $e^{\langle x\rangle}$. Here, $\langle x\rangle$ can be a non-integer. The algorithm uses a Maclaurin series.

`\pgfmathsqrt{`$\langle x\rangle$`}`

Defines `\pgfmathresult` as $\sqrt{\langle x\rangle}$.

`\pgfmathveclen{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

Defines `\pgfmathresult` as $\sqrt{\langle x\rangle^2 + \langle y\rangle^2}$. This uses a polynomial approximation, based on ideas due to Rouben Rostamian.

40.2 Trignometric Functions

\pgfmathpi

Defines `\pgfmathresult` as 3.14159.

\pgfmathdeg{⟨*x*⟩}

Defines `\pgfmathresult` as ⟨*x*⟩ (given in radians) converted to degrees.

\pgfmathrad{⟨*x*⟩}

Defines `\pgfmathresult` as ⟨*x*⟩ (given in degrees) converted to radians.

\pgfmathsin{⟨*x*⟩}

Defines `\pgfmathresult` as the sine of ⟨*x*⟩.

\pgfmathcos{⟨*x*⟩}

Defines `\pgfmathresult` as the cosine of ⟨*x*⟩.

\pgfmathtan{⟨*x*⟩}

Defines `\pgfmathresult` as the tangant of ⟨*x*⟩.

\pgfmathasin{⟨*x*⟩}

the arcsine of ⟨*x*⟩.

\pgfmathacos{⟨*x*⟩}

Defines `\pgfmathresult` as the arccosine of ⟨*x*⟩.

\pgfmathatan{⟨*x*⟩}

Defines `\pgfmathresult` as the arctangent of ⟨*x*⟩.

40.3 Pseudo-Random Numbers

\pgfmathgeneratepseudorandomnumber

Defines `\pgfmathresult` as a pseudo-random integer between 1 and $2^{31}-1$. This uses a linear congruency generator, based on ideas due to Erich Janka.

\pgfmathrnd

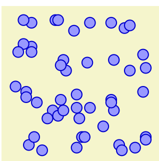
Defines `\pgfmathresult` as a pseudo-random number between 0.0 and 1.0

\pgfmathrand

Defines `\pgfmathresult` as a pseudo-random number between -1.0 and 1.0

\pgfmathrandominteger{⟨*macro*⟩}{⟨*maximum*⟩}{⟨*minimum*⟩}

This defines ⟨*macro*⟩ as a pseudo-randomly generated integer from the range ⟨*maximum*⟩ to ⟨*minimum*⟩ (inclusive).



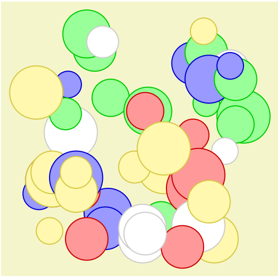
```
\begin{pgfpicture}
   \foreach \x in {1,...,50}{
      \pgfmathrandominteger{\a}{1}{50}
      \pgfmathrandominteger{\b}{1}{50}
      \pgfpathcircle{\pgfpoint{+\a pt}{+\b pt}}{+2pt}
      \color{blue!40!white}
      \pgfsetstrokecolor{blue!80!black}
      \pgfusepath{stroke, fill}
   }
\end{pgfpicture}
```

\pgfmathdeclarerandomlist{⟨*list name*⟩}{{⟨*item-1*⟩}{⟨*item 2*⟩}...}

This creates a list of items with the name ⟨*list name*⟩.

`\pgfmathrandomitem{`⟨*macro*⟩`}{`⟨*list name*⟩`}`

Select an item from a random list ⟨*list name*⟩. The selected item is placed in ⟨*macro*⟩.



```
\begin{pgfpicture}
   \pgfmathdeclarerandomlist{color}{{red}{blue}{green}{yellow}{white}}
   \foreach \a in {1,...,50}{
      \pgfmathrandominteger{\x}{1}{85}
      \pgfmathrandominteger{\y}{1}{85}
      \pgfmathrandominteger{\r}{5}{10}
      \pgfmathrandomitem{\c}{color}
      \pgfpathcircle{\pgfpoint{+\x pt}{+\y pt}}{+\r pt}
      \color{\c!40!white}
      \pgfsetstrokecolor{\c!80!black}
      \pgfusepath{stroke, fill}
   }
\end{pgfpicture}
```

`\pgfmathsetseed{`⟨*integer*⟩`}`

Explicitly set seed for the pseudo-random number generator. By default it is set to the value of `\time`×`\year`.

40.4 Conversion Between Bases

PGF provides limited support for conversion between *representations* of numbers. Currently the numbers must be positive integers in the range 0 to $2^{31} - 1$, and the bases in the range 2 to 36. All digits representing numbers greater than 9 (in base ten), are alphabetic, but may be upper or lower case. Note, that again, examples in this section, show the result of the calculation *not* what is displayed in the document.

`\pgfmathbasetodec{`⟨*macro*⟩`}{`⟨*number*⟩`}{`⟨*base*⟩`}`

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base ⟨*base*⟩ to base 10. Alphabetic digits can be upper or lower case.

4223

```
\pgfmathbasetodec\mynumber{107f}{16}
```

25512

```
\pgfmathbasetodec\mynumber{33FC}{20}
```

`\pgfmathdectobase{`⟨*macro*⟩`}{`⟨*number*⟩`}{`⟨*base*⟩`}`

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base 10 to base ⟨*base*⟩. Any resulting alphabetic digits are in *lower case*.

ffff

```
\pgfmathdectobase\mynumber{65535}{16}
```

`\pgfmathdectoBase{`⟨*macro*⟩`}{`⟨*number*⟩`}{`⟨*base*⟩`}`

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base 10 to base ⟨*base*⟩. Any resulting alphabetic digits are in *upper case*.

FFFF

```
\pgfmathdectoBase\mynumber{65535}{16}
```

`\pgfmathbasetobase{`⟨*macro*⟩`}{`⟨*number*⟩`}{`⟨*base-1*⟩`}{`⟨*base-2*⟩`}`

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base ⟨*base-1*⟩ to base ⟨*base-2*⟩. Alphabetic digits in ⟨*number*⟩ can be upper or lower case, but any resulting alphabetic digits are in *lower case*.

db

```
\pgfmathbasetobase\mynumber{11011011}{2}{16}
```

`\pgfmathbasetoBase{`⟨*macro*⟩`}{`⟨*number*⟩`}{`⟨*base-1*⟩`}{`⟨*base-2*⟩`}`

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base ⟨*base-1*⟩ to base ⟨*base-2*⟩. Alphabetic digits in ⟨*number*⟩ can be upper or lower case, but any resulting alphabetic digits are in *upper case*.

31B

```
\pgfmathbasetoBase\mynumber{121212}{3}{12}
```

\pgfmathsetbasenumberlength{⟨*integer*⟩}

Set the number of digits in the result of a base conversion to ⟨*integer*⟩. If the result of a conversion has less digits than this number it is prefixed with zeros.

00001111

```
\pgfmathsetbasenumberlength{8}
\pgfmathdectobase\mynumber{15}{2}
```

41 Reimplementing the Computations of the Mathematical Engine

Perhaps you are not satisfied with the Maclaurin series for e^x. Perhaps you have a fantastically more accurate and efficient way of calculating the sine or cosine of angles. Perhaps you would like the library to interface with a package such as `fp` for fixed-point arithmetic (but note that `fp` is *very* slow). In these case you will want to replace the current implementations of the computations done by the mathematical engine by your own code.

The mathematical engine was designed with such a replacement in mind. For this reason, the operations and functions like `\pgfmathadd` are implemented in the following manner:

- `\pgfmath`⟨*function name*⟩

  This macro is the "public" interface for the function ⟨*function name*⟩. All arguments passed to this macro are evaluated using `\pgfmathparse` and then passed on to the following function:

- `\pgfmath`⟨*function name*⟩`@`

  This macro is the "non-public" implementation of the functions algorithm (but note that, for speed, the parser calls this macro rather than the "public" one). Arguments passed to this macro are expected to be numbers *without units*. This is the macro which should be rewritten with your prize-winning new algorithm.

The effect of `\pgfmath`⟨*function name*⟩`@` should be to set the macro `\pgfmathresult` to the correct value (namely to the result of the computation without units). Furthermore, the function should have no other side effects, that is, it should not change any global values. One way to achieve this is to use the following code:

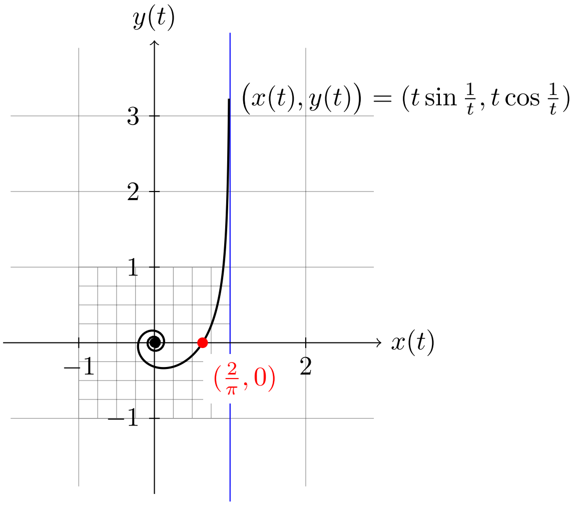
```
\def\pgfmath...@#1#2...{%
   \begingroup%
      ... code for algorithm XXX ...
      \pgfmath@returnone\pgfmath@x%
   \endgroup%
}
```

The macro `\pgfmath@returnone<macro>` uses some `\aftergroup` magic to save result of the algorithm, by defining `\pgfmathresult` as the expansion of `<macro>` *without units*. `<macro>` can be a macro containing a number (with or without units), or a dimension or count (or possibly a skip) register. By performing the algorithm within a TeX group, PGF registers such as `\pgf@x`, `\pgf@y` and `\c@pgf@counta`, `\c@pgfcountb`, and so forth can be used at will. Note that current the implementation uses `\pgfmath@x`, `\pgfmath@y`, and `\c@pgfmath@counta`, `\c@pgfmath@countb` throughout, so for consistency these should be employed. Whilst they are currently `\let` to their PGF equivalents (see `pgfmathutil.code.tex`), this could change (as could the PGF registers), so keeping things consistent is probably a good idea.

Part VII

The Basic Layer

by Till Tantau




```
\begin{tikzpicture}
  \draw[gray,very thin] (-1.9,-1.9) grid (2.9,3.9)
          [step=0.25cm] (-1,-1) grid (1,1);
  \draw[blue] (1,-2.1) -- (1,4.1); % asymptote

  \draw[->] (-2,0) -- (3,0) node[right] {$x(t)$};
  \draw[->] (0,-2) -- (0,4) node[above] {$y(t)$};

  \foreach \pos in {-1,2}
    \draw[shift={(\pos,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\pos$};

  \foreach \pos in {-1,1,2,3}
    \draw[shift={(0,\pos)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\pos$};

  \fill (0,0) circle (0.064cm);
  \draw[thick,parametric,domain=0.4:1.5,samples=200]
    % The plot is reparameterised such that there are more samples
    % near the center.
    plot[id=asymptotic-example] function{(t*t*t)*sin(1/(t*t*t)),(t*t*t)*cos(1/(t*t*t))}
    node[right] {$\bigl(x(t),y(t)\bigr) = (t\sin \frac{1}{t}, t\cos \frac{1}{t})$};

  \fill[red] (0.63662,0) circle (2pt)
    node [below right,fill=white,yshift=-4pt] {$(\frac{2}{\pi},0)$};
\end{tikzpicture}
```

42 Design Principles

This section describes the basic layer of PGF. This layer is build on top of the system layer. Whereas the system layer just provides the absolute minimum for drawing graphics, the basic layer provides numerous commands that make it possible to create sophisticated graphics easily and also quickly.

The basic layer does not provide a convenient syntax for describing graphics, which is left to frontends like Ti*k*Z. For this reason, the basic layer is typically used only by "other programs." For example, the BEAMER package uses the basic layer extensively, but does not need a convenient input syntax. Rather, speed and flexibility are needed when BEAMER creates graphics.

The following basic design principles underlie the basic layer:

1. Structuring into a core and several optional packages.
2. Consistently named TEX macros for all graphics commands.
3. Path-centered description of graphics.
4. Coordinate transformation system.

42.1 Core and Optional Packages

The basic layer consists of a *core package*, called `pgfcore`, which provides the most basic commands, and several optional package like `pgfbaseshade` that offer more special-purpose commands.

```
\usepackage{pgfbaseplot} % LaTeX
\input pgfbaseplot.tex   % plain TeX
\usemodule[pgfbaseplot]  % ConTeXt
```

provides commands for plotting functions

```
\usepackage{pgfbaseshapes} % LaTeX
\input pgfbaseshapes.tex   % plain TeX
\usemodule[pgfbaseshapes]  % ConTeXt
```

provides commands for drawing shapes and nodes

```
\usepackage{pgfbasepatterns} % LaTeX
\input pgfbasepatterns.tex   % plain TeX
\usemodule[pgfbasepatterns]  % ConTeXt
```

provides commands for declaring and using tiling patterns

```
\usepackage{pgfbaseimage} % LaTeX
\input pgfbaseimage.tex   % plain TeX
\usemodule[pgfbaseimage]  % ConTeXt
```

This package provides commands for including external images. For LATEX-users the `graphicx` package does a better job at this than the `pgfbaseimage` package does, so you should normally use `\includegraphics` and not `\pgfimage`. However, in some situations (like when masking is needed or when plain TEX is used) this package is needed.

```
\usepackage{pgfbaselayers} % LaTeX
\input pgfbaselayers.tex   % plain TeX
\usemodule[pgfbaselayers]  % ConTeXt
```

This package provides commands for creating layered graphics. Using layers you can later on say that a certain path should be behind a path that was specified earlier.

```
\usepackage{pgfbasesnakes} % LaTeX
\input pgfbasesnakes.tex   % plain TeX
\usemodule[pgfbasesnakes]  % ConTeXt
```

This package provides commands for adding snaked lines to the path. Such lines are not straight but rather wind in some specific fashion.

```
\usepackage{pgfbasematrix} % LaTeX
```

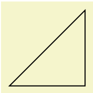
```
\input pgfbasematrix.tex    % plain TeX
\usemodule[pgfbasematrix]   % ConTeXt
```

This package provides the `\pgfmatrix` command.

If you say `\usepackage{pgf}` or `\input pgf.tex` or `\usemodule[pgf]`, all of the optional packages are loaded (as well as the core and the system layer).

42.2 Communicating with the Basic Layer via Macros

In order to "communicate" with the basic layer you use long sequences of commands that start with `\pgf`. You are only allowed to give these commands inside a `{pgfpicture}` environment. (Note that `{tikzpicture}` opens a `{pgfpicture}` internally, so you can freely mix PGF commands and Ti*k*Z commands inside a `{tikzpicture}`.) It is possible to "do other things" between the commands. For example, you might use one command to move to a certain point, then have a complicated computation of the next point, and then move there.



```
\newdimen\myypos
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{\myypos}}
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \advance \myypos by 1cm
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}
```

The following naming conventions are used in the basic layer:

1. All commands and environments start with `pgf`.
2. All commands that specify a point (a coordinate) start with `\pgfpoint`.
3. All commands that extend the current path start with `\pgfpath`.
4. All commands that set/change a graphics parameter start with `\pgfset`.
5. All commands that use a previously declared object (like a path, image or shading) start with `\pgfuse`.
6. All commands having to do with coordinate transformations start with `\pgftransform`.
7. All commands having to do with arrow tips start with `\pgfarrows`.
8. All commands for "quickly" extending or drawing a path start with `\pgfpathq` or `\pgfusepathq`.
9. All commands having to do with matrices start with `\pgfmatrix`.

42.3 Path-Centered Approach

In PGF the most important entity is the *path*. All graphics are composed of numerous paths that can be stroked, filled, shaded, or clipped against. Paths can be closed or open, they can self-intersect and consist of unconnected parts.

Paths are first *constructed* and then *used*. In order to construct a path, you can use commands starting with `\pgfpath`. Each time such a command is called, the current path is extended in some way.

Once a path has been completely constructed, you can use it using the command `\pgfusepath`. Depending on the parameters given to this command, the path will be stroked (drawn) or filled or subsequent drawings will be clipped against this path.

42.4 Coordinate Versus Canvas Transformations

PGF provides two transformation systems: PGF's own *coordinate* transformation matrix and PDF's or PostScript's *canvas* transformation matrix. These two systems are quite different. Whereas a scaling by a factor of, say, 2 of the canvas causes *everything* to be scaled by this factor (including the thickness of lines and text), a scaling of two in the coordinate system causes only the *coordinates* to be scaled, but not the line width nor text.

By default, all transformations only apply to the coordinate transformation system. However, using the command `\pgflowlevel` it is possible to apply a transformation to the canvas.

Coordinate transformations are often preferable over canvas transformations. Text and lines that are transformed using canvas transformations suffer from differing sizes and lines whose thickness differs depending on whether the line is horizontal or vertical. To appreciate the difference, consider the following two "circles" both of which have been scaled in the x-direction by a factor of 3 and by a factor of 0.5 in the y-direction. The left circle uses a canvas transformation, the right uses PGF's coordinate transformation (some viewers will render the left graphic incorrectly since they do no apply the low-level transformation the way they should):

43 Hierarchical Structures: Package, Environments, Scopes, and Text

43.1 Overview

PGF uses two kinds of hierarchical structuring: First, the package itself is structured hierarchically, consisting of different packages that are built on top of each other. Second, PGF allows you to structure your graphics hierarchically using environments and scopes.

43.1.1 The Hierarchical Structure of the Package

The PGF system consists of several layers:

System layer. The lowest layer is called the *system layer*, though it might also be called "driver layer" or perhaps "backend layer." Its job is to provide an abstraction of the details of which driver is used to transform the `.dvi` file. The system layer is implemented by the package `pgfsys`, which will load appropriate driver files as needed.

The system layer is documented in Part VIII.

Basic layer. The basic layer is loaded by the package `pgf`. Some applications do not need all of the functionality of the basic layer, so it is possible to load only the `pgfcore` and some other packages starting with `pgfbase`.

The basic layer is documented in the present part.

Frontend layer. The frontend layer is not loaded by a single packages. Rather, different packages, like Ti*k*Z or PGFPICT2E, are different frontends to the basic layer.

The Ti*k*Z frontend is documented in Part III.

Each layer will automatically load the necessary files of the layers below it.

In addition to the packages of these layers, there are also some library packages. These packages provide additional definitions of things like new arrow tips or new plot handlers.

The library packages are documented in Part IV.

43.1.2 The Hierarchical Structure of Graphics

Graphics in PGF are typically structured hierarchically. Hierarchical structuring can be used to identify groups of graphical elements that are to be treated "in the same way." For example, you might group together a number of paths, all of which are to be drawn in red. Then, when you decide later on that you like them to be drawn in, say, blue, all you have to do is to change the color once.

The general mechanism underlying hierarchical structuring is known as *scoping* in computer science. The idea is that all changes to the general "state" of the graphic that are done inside a scope are local to that scope. So, if you change the color inside a scope, this does not affect the color used outside the scope. Likewise, when you change the line width in a scope, the line width outside is not changed, and so on.

There are different ways of starting and ending scopes of graphic parameters. Unfortunately, these scopes are sometimes "in conflict" with each other and it is sometimes not immediately clear which scopes apply. In essence, the following scoping mechanisms are available:

1. The "outermost" scope supported by PGF is the `{pgfpicture}` environment. All changes to the graphic state done inside a `{pgfpicture}` are local to that picture.

   In general, it is *not* possible to set graphic parameters globally outside any `{pgfpicture}` environments. Thus, you can *not* say `\pgfsetlinewidth{1pt}` at the beginning of your document to have a default line width of one point. Rather, you have to (re)set all graphic parameters inside each `{pgfpicture}`. (If this is too bothersome, try defining some macro that does the job for you.)

2. Inside a `{pgfpicture}` you can use a `{pgfscope}` environment to keep changes of the graphic state local to that environment.

   The effect of commands that change the graphic state are local to the current `{pgfscope}` but not always to the current TEX group. Thus, if you open a TEX group (some text in curly braces) inside a {pgfscope}, and if you change, for example, the dash pattern, the effect of this changed dash pattern will persist till the end of the {pgfscope}.

Unfortunately, this is not always the case. *Some* graphic parameters only persist till the end of the current TEX group. For example, when you use \pgfsetarrows to set the arrow tip inside a TEX group, the effect lasts only till the end of the current TEX group.

3. Some graphic parameters are not scoped by {pgfscope} but "already" by TEX groups. For example, the effect of coordinate transformation commands is always local to the current TEX group.

   Since every {pgfscope} automatically creates a TEX group, all graphic parameters that are local to the current TEX group are also local to the current {pgfscope}.

4. Some graphic parameters can only be scoped using TEX groups, since in some situations it is not possible to introduce a {pgfscope}. For example, a path always has to be completely constructed and used in the same {pgfscope}. However, we might wish to have different coordinate transformations apply to different points on the path. In this case, we can use TEX groups to keep the effect local, but we could not use {pgfscope}.

5. The \pgftext command can be used to create a scope in which TEX "escapes back" to normal TEX mode. The text passed to the \pgftext is "heavily guarded" against having any effect on the scope in which it is used. For example, it is possibly to use another {pgfpicture} environment inside the argument of \pgftext.

Most of the complications can be avoided if you stick to the following rules:

- Give graphic commands only inside {pgfpicture} environments.
- Use {pgfscope} to structure graphics.
- Do not use TEX groups inside graphics, *except* for keeping the effect of coordinate transformations local.

43.2 The Hierarchical Structure of the Package

Before we come to the structuring commands provided by PGF to structure your graphics, let us first have a look at the structure of the package itself.

43.2.1 The Main Package

To use PGF, include the following package:

```
\usepackage{pgf} % LaTeX
\input pgf.tex   % plain TeX
\usemodule[pgf]  % ConTeXt
```

This package loads the complete "basic layer" of PGF. That is, it will load all of the commands described in the current part of this manual, but it will not load frontends like Ti*k*Z.

In detail, this package will load the following packages, each of which can also be loaded individually:

- pgfsys, which is the lowest layer of PGF and which is always needed. This file will read pgf.cfg to discern which driver is to be used. See Section 59.1 for details.
- pgfcore, which is the central core of PGF and which is always needed unless you intend to write a new basic layer from scratch.
- pgfbaseimage, which provides commands for declaring and using images. An example is \pgfuseimage.
- pgfbaseshapes, which provides commands for declaring and using shapes. An example is \pgfdeclareshape.
- pgfbaseplot, which provides commands for plotting functions.

Including any of the last three packages will automatically load the first two.

In LATEX, the package takes two options:

`\usepackage[draft]{pgf}`

When this option is set, all images will be replaced by empty rectangles. This can speedup compilation.

`\usepackage[version=⟨version⟩]{pgf}`

Indicates that the commands of version ⟨*version*⟩ need to be defined. If you set ⟨*version*⟩ to `0.65`, then a large bunch of "compatibility commands" are loaded. If you set ⟨*version*⟩ to `0.96`, then these compatibility commands will not be loaded.

If this option is not given at all, then the commands of all versions are defined.

43.2.2 The Core Package

```
\usepackage{pgfcore} % LaTeX
\input pgfcore.tex   % plain TeX
\usemodule[pgfcore]  % ConTeXt
```

This package defines all of the basic layer's commands, except for the commands defined in the additional packages like `pgfbaseplot`. Typically commands defined by the core include `\pgfusepath` or `\pgfpoint`. The core is internally structured into several subpackages, but the subpackages cannot be loaded individually since they are all "interrelated."

43.2.3 The Optional Basic Layer Packages

The `pgf` package automatically loads the following packages, but you can also load them individually (all of them automatically include the core):

- `pgfbaseshapes` This package provides commands for drawing nodes and shapes. These commands are explained in Section 49.
- `pgfbaseplot` This package provides commands for plotting function. The commands are explained in Section 56.
- `pgfbaseimage` This package provides commands for including (external) images. The commands are explained in Section 53.

43.2.4 The Library Packages

There is a special command for loading library packages.

`\usepgflibrary{⟨list of libraries⟩}`

Use this command to load further libraries. The list of libraries should contain the names of libraries separated by commas. Instead of curly braces, you can also use square brackets. If you try to load a library a second time, nothing will happen.

Example: `\usepgflibrary{arrows}`

This command causes the the file `pgflibrary`⟨*library*⟩`.code.tex` to be loaded for each ⟨*library*⟩ in the ⟨*list of libraries*⟩. This means that in order to write your own library file, place a file of the appropriate name somewhere where TeX can find it. LaTeX, plain TeX, and ConTeXt users can then use your library.

You should also consider adding a Ti*k*Z library that simply includes your PGF library.

43.3 The Hierarchical Structure of the Graphics

43.3.1 The Main Environment

Most, but not all, commands of the PGF package must be given within a `{pgfpicture}` environment. The only commands that (must) be given outside are commands having to do with including images (like `\pgfuseimage`) and with inserting complete shadings (like `\pgfuseshading`). However, just to keep life entertaining, the `\pgfshadepath` command must be given *inside* a `{pgfpicture}` environment.

`\begin{pgfpicture}`
  ⟨*environment contents*⟩

`\end{pgfpicture}`

This environment will insert a TeX box containing the graphic drawn by the ⟨*environment contents*⟩ at the current position.

The size of the bounding box. The size of the box is determined in the following manner: While PGF parses the ⟨*environment contents*⟩, it keeps track of a bounding box for the graphic. Essentially, this bounding box is the smallest box that contains all coordinates mentioned in the graphics. Some coordinates may be "mentioned" by PGF itself; for example, when you add circle to the current path, the support points of the curve making up the circle are also "mentioned" despite the fact that you will not "see" them in your code.

Once the ⟨*environment contents*⟩ has been parsed completely, a TeX box is created whose size is the size of the computed bounding box and this box is inserted at the current position.

Hello ▭ World!

```
Hello \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} World!
```

Sometimes, you may need more fine-grained control over the size of the bounding box. For example, the computed bounding box may be too large or you intensionally wish the box to be "too small." In these cases, you can use the command `\pgfusepath{use as bounding box}`, as described in Section 47.5.

The baseline of the bounding box. When the box containing the graphic is inserted into the normal text, the baseline of the graphic is normally at the bottom of the graphic. For this reason, the following two sets of code lines have the same effect, despite the fact that the second graphic uses "higher" coordinates than the first:

Rectangles ▭ and ▭.

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture}.
```

You can change the baseline using the `\pgfsetbaseline` command, see below.

Rectangles ▭ and ▭.

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}
\end{pgfpicture}.
```

Including text and images in a picture. You cannot directly include text and images in a picture. Thus, you should *not* simply write some text in a `{pgfpicture}` or use a command like `\includegraphics` or even `\pgfimage`. In all these cases, you need to place the text inside a `\pgftext` command. This will "escape back" to normal TeX mode, see Section 43.3.3 for details.

Remembering a picture position for later reference. After a picture has been typeset, its position on the page is normally forgotten by PGF and also by TeX. This means that is not possible to reference a node in this picture later on. In particular, it is normally impossible to draw lines between nodes in different pictures automatically.

In order to make PGF "remember" a picture, the TeX-if `\ifpgfrememberpicturepositiononpage` should be set to `true`. It is only important that this TeX-if is `true` at the end of the `{pgfpicture}`-environment, so you can switch it on inside the environment. However, you can also just switch it on globally, then the positions of all pictures are remembered.

There are several reasons why the remembering is not switched on by default. First, it does not work for all backend drivers (currently, it works only for pdfTeX). Second, it requires two passes of TeX over the file; on the first pass all positions will be wrong. Third, for every remembered picture a line is added to the `.aux`-file, which may result in a large number of extra lines.

Despite all these "problems," for documents that are processed with pdfTeX and in which there is only a small number of pictures (less than a hundred or so), you can switch on this option globally, it will not cause any significant slowing of TeX.

`\pgfpicture`
⟨*environment contents*⟩
`\endpgfpicture`

The plain TeX version of the environment. Note that in this version, also, a TeX group is created around the environment.

`\startpgfpicture`
⟨*environment contents*⟩
`\stoppgfpicture`

This is the ConTeXt version of the environment.

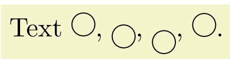
`\ifpgfrememberpicturepositiononpage`

Determines whether the position of pictures on the page should be recorded. The value of this TeX-if at the end of a `{pgfpicture}` environment is important, not the value at the beginning.

If this option is set to true of a picture, PGF will attempt to record the position of the picture on the page. (This attempt will fail with most drivers and when it works it typically requires two runs of TeX.) The position is not directly accessible. Rather, the nodes mechanism will use this position if you access a node from another picture. See Sections 49.3.2 and 13.12 for more details.

`\pgfsetbaseline{`⟨*dimension*⟩`}`

This command specifies a y-coordinate of the picture that should be used as the baseline of the whole picture. When a PGF picture has been typeset completely, PGF must decide at which height the baseline of the picture should lie. Normally, the baseline is set to the y-coordinate of the bottom of the picture, but it is often desirable to use another height.



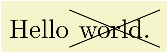

```
Text \tikz{\pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
     \tikz{\pgfsetbaseline{0pt}
          \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
     \tikz{\pgfsetbaseline{.5ex}
          \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}},
     \tikz{\pgfsetbaseline{-1ex}
          \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}}.
```

`\pgfsetbaselinepointnow{`⟨*point*⟩`}`

This command specifies the baseline indirectly, namely as the y-coordinate that the given ⟨*point*⟩ has when the command is called.

`\pgfsetbaselinepointlater{`⟨*point*⟩`}`

This command also specifies the baseline indirectly, but the y-coordinate of the given ⟨*point*⟩ is only computed at the end of the picture.




```
Hello
\tikz{
  \pgfsetbaselinepointlater{\pgfpointanchor{X}{base}}
  % Note: no shape X, yet
  \node [cross out,draw] (X) {world.};
}
```

43.3.2 Graphic Scope Environments

Inside a `{pgfpicture}` environment you can substructure your picture using the following environment:

`\begin{pgfscope}`
⟨*environment contents*⟩

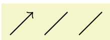
\end{pgfscope}

All changes to the graphic state done inside this environment are local to the environment. The graphic state includes the following:

- The line width.
- The stroke and fill colors.
- The dash pattern.
- The line join and cap.
- The miter limit.
- The canvas transformation matrix.
- The clipping path.

Other parameters may also influence how graphics are rendered, but they are *not* part of the graphic state. For example, the arrow tip kind is not part of the graphic state and the effect of commands setting the arrow tip kind are local to the current TEX group, not to the current {pgfscope}. However, since {pgfscope} starts and ends a TEX group automatically, a {pgfscope} can be used to limit the effect of, say, commands that set the arrow tip kind.

```
\begin{pgfpicture}
  \begin{pgfscope}
    {
      \pgfsetlinewidth{2pt}
      \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}
    }
    \pgfpathrectangle{\pgfpoint{3ex}{0ex}}{\pgfpoint{2ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathrectangle{\pgfpoint{6ex}{0ex}}{\pgfpoint{2ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```



```
\begin{pgfpicture}
  \begin{pgfscope}
    {
      \pgfsetarrows{-to}
      \pgfpathmoveto{\pgfpointorigin}\pgfpathlineto{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}
    }
    \pgfpathmoveto{\pgfpoint{3ex}{0ex}}\pgfpathlineto{\pgfpoint{5ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathmoveto{\pgfpoint{6ex}{0ex}}\pgfpathlineto{\pgfpoint{8ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

At the start of the scope, the current path must be empty, that is, you cannot open a scope while constructing a path.

It is usually a good idea *not* to introduce TEX groups inside a {pgfscope} environment.

\pgfscope
⟨*environment contents*⟩
\endpgfscope

Plain TEX version of the {pgfscope} environment.

\startpgfscope
⟨*environment contents*⟩
\stoppgfscope

This is the ConTEXt version of the environment.

The following scopes also encapsulate certain properties of the graphic state. However, they are typically not used directly by the user.

\begin{pgfinterruptpath}
  ⟨*environment contents*⟩
\end{pgfinterruptpath}

This environment can be used to temporarily interrupt the construction of the current path. The effect will be that the path currently under construction will be "stored away" and restored at the end of the environment. Inside the environment you can construct a new path and do something with it.

An example application of this environment is the arrow tip caching. Suppose you ask PGF to use a specific arrow tip kind. When the arrow tip needs to be rendered for the first time, PGF will "cache" the path that makes up the arrow tip. To do so, it interrupts the current path construction and then protocols the path of the arrow tip. The `{pgfinterruptpath}` environment is used to ensure that this does not interfere with the path to which the arrow tips should be attached.

This command does *not* install a `{pgfscope}`. In particular, it does not call any `\pgfsys@` commands at all, which would, indeed, be dangerous in the middle of a path construction.

\pgfinterruptpath
  ⟨*environment contents*⟩
\endpgfinterruptpath

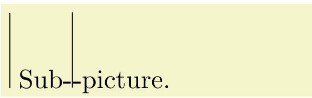
Plain TeX version of the environment.

\startpgfinterruptpath
  ⟨*environment contents*⟩
\stoppgfinterruptpath

ConTeXt version of the environment.

\begin{pgfinterruptpicture}
  ⟨*environment contents*⟩
\end{pgfinterruptpicture}

This environment can be used to temporarily interrupt a `{pgfpicture}`. However, the environment is intended only to be used at the beginning and end of a box that is (later) inserted into a `{pgfpicture}` using `\pgfqbox`. You cannot use this environment directly inside a `{pgfpicture}`.




```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}} % In the middle of path, now
  \newbox\mybox
  \setbox\mybox=\hbox{
    \begin{pgfinterruptpicture}
      Sub-\begin{pgfpicture} % a subpicture
        \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
        \pgfpathlineto{\pgfpoint{1cm}{1cm}}
        \pgfusepath{stroke}
      \end{pgfpicture}-picture.
    \end{pgfinterruptpicture}
  }
  \pgfqbox{\mybox}%
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}\hskip3.9cm
```

\pgfinterruptpicture
  ⟨*environment contents*⟩
\endpgfinterruptpicture

Plain TeX version of the environment.

\startpgfinterruptpicture
  ⟨*environment contents*⟩
\stoppgfinterruptpicture

ConTeXt version of the environment.

\begin{pgfinterruptboundingbox}
  ⟨*environment contents*⟩

`\end{pgfinterruptboundingbox}`

This environment temporarily interrupts the computation of the bounding box and sets up a new bounding box. At the beginning of the environment the old bounding box is saved and an empty bounding box is installed. After the environment the orginal bounding box is reinstalled as if nothing has happened.

`\pgfinterruptboundingbox`
⟨*environment contents*⟩
`\endpgfinterruptboundingbox`

Plain TEX version of the environment.

`\startpgfinterruptboundingbox`
⟨*environment contents*⟩
`\stoppgfinterruptboundingbox`

ConTEXt version of the environment.

43.3.3 Inserting Text and Images

Often, you may wish to add normal TEX text at a certain point inside a `{pgfpicture}`. You cannot do so "directly," that is, you cannot simply write this text inside the `{pgfpicture}` environment. Rather, you must pass the text as an argument to the `\pgftext` command.

You must *also* use the `\pgftext` command to insert an image or a shading into a `{pgfpicture}`.

`\pgftext[`⟨*options*⟩`]{`⟨*text*⟩`}`

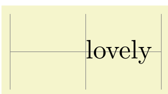
This command will typeset ⟨*text*⟩ in normal TEX mode and insert the resulting box into the `{pgfpicture}`. The bounding box of the graphic will be updated so that all of the text box is inside. Be default, the text box is centered at the origin, but this can be changed either by giving appropriate ⟨*options*⟩ or by applying an appropriate coordinate transformation beforehand.

The ⟨*text*⟩ may contain verbatim text. (In other words, the ⟨*text*⟩ "argument" is not a normal argument, but is put in a box and some `\aftergroup` hackery is used to find the end of the box.)

PGF's current (high-level) coordinate transformation is synchronized with the canvas transformation matrix temporarily when the text box is inserted. The effect is that if there is currently a high-level rotation of, say, 30 degrees, the ⟨*text*⟩ will also be rotated by thirty degrees. If you do not want this effect, you have to (possibly temporarily) reset the high-level transformation matrix.

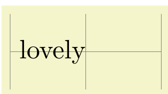
The following ⟨*options*⟩ may be given as conveniences:

- `left` causes the text box to be placed such that its left border is on the origin.



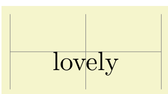

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[left] {lovely}}
```

- `right` causes the text box to be placed such that its right border is on the origin.

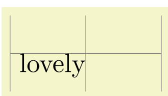



```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[right] {lovely}}
```

- `top` causes the text box to be placed such that its top is on the origin. This option can be used together with the `left` or `right` option.

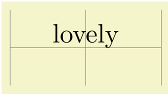



```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[top] {lovely}}
```




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[top,right] {lovely}}
```

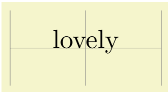
- `bottom` causes the text box to be placed such that its bottom is on the origin.



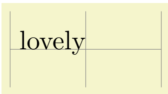

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[bottom] {lovely}}
```


```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[bottom,right] {lovely}}
```

- `base` causes the text box to be placed such that its baseline is on the origin.

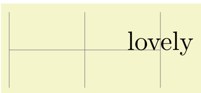



```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[base] {lovely}}
```



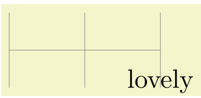

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[base,right] {lovely}}
```

- `at`=⟨*point*⟩ Translates the origin (that is, the point where the text is shown) to ⟨*point*⟩.



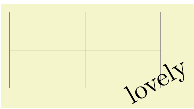

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[base,at={\pgfpoint{1cm}{0cm}}] {lovely}}
```

- `x`=⟨*dimension*⟩ Translates the origin by ⟨*dimension*⟩ along the x-axis.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[base,x=1cm,y=-0.5cm] {lovely}}
```

- `y`=⟨*dimension*⟩ works like the `x` option.
- `rotate`=⟨*degree*⟩ Rotates the coordinate system by ⟨*degree*⟩. This will also rotate the text box.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[base,x=1cm,y=-0.5cm,rotate=30] {lovely}}
```

44 Specifying Coordinates

44.1 Overview

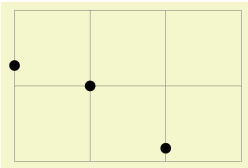
Most PGF commands expect you to provide the coordinates of a *point* (also called *coordinate*) inside your picture. Points are always “local” to your picture, that is, they never refer to an absolute position on the page, but to a position inside the current `{pgfpicture}` environment. To specify a coordinate you can use commands that start with `\pgfpoint`.

44.2 Basic Coordinate Commands

The following commands are the most basic for specifying a coordinate.

`\pgfpoint{`⟨*x coordinate*⟩`}{`⟨*y coordinate*⟩`}`

Yields a point location. The coordinates are given as TeX dimensions.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpoint{1cm}{1cm}} {2pt}
  \pgfpathcircle{\pgfpoint{2cm}{5pt}} {2pt}
  \pgfpathcircle{\pgfpoint{0pt}{.5in}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

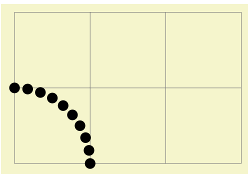
`\pgfpointorigin`

Yields the origin. Same as `\pgfpoint{0pt}{0pt}`.

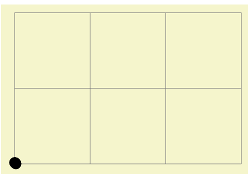
`\pgfpointpolar{`⟨*degree*⟩`}{`⟨*radius*⟩`/`⟨*y-radius*⟩`}`

Yields a point location given in polar coordinates. You can specify the angle only in degrees, radians are not supported, currently.

If the optional ⟨*y-radius*⟩ is given, the polar coordinate is actually a coordinate on an ellipse whose x-radius is given by ⟨*radius*⟩ and whose y-radius is given by ⟨*y-radius*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```



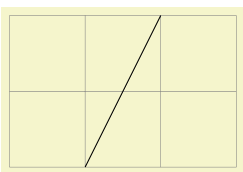
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm/2cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```

44.3 Coordinates in the XY-Coordinate System

Coordinates can also be specified as multiples of an x-vector and a y-vector. Normally, the x-vector points one centimeter in the x-direction and the y-vector points one centimeter in the y-direction, but using the commands `\pgfsetxvec` and `\pgfsetyvec` they can be changed. Note that the x- and y-vector do not necessarily point “horizontally” and “vertically.”

`\pgfpointxy{`⟨s_x⟩`}{`⟨s_y⟩`}`

Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetxvec{⟨*point*⟩}

Sets that current x-vector for usage in the xyz-coordinate system.

Example:

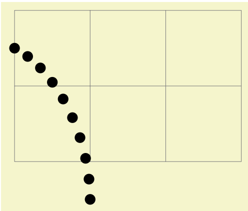
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}

  \color{red}
  \pgfsetxvec{\pgfpoint{0.75cm}{0cm}}
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetyvec{⟨*point*⟩}

Works like `\pgfsetyvec`.

\pgfpointpolarxy{⟨*degree*⟩}{⟨*radius*⟩/⟨*y-radius*⟩}

This command is similar to the `\pgfpointpolar` command, but the ⟨*radius*⟩ is now a factor to be interpreted in the xy-coordinate system. This means that a degree of `0` is the same as the x-vector of the xy-coordinate system times ⟨*radius*⟩ and a degree of `90` is the y-vecotr times ⟨*radius*⟩. As for `\pgfpointpolar`, a ⟨*radius*⟩ can also be a pair separated by a slash. In this case, the x- and y-vectors are multiplied by different factors.



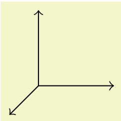
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \begin{scope}[x={(1cm,-5mm)},y=1.5cm]
    \foreach \angle in {0,10,...,90}
      {\pgfpathcircle{\pgfpointpolarxy{\angle}{1}}{2pt}}
    \pgfusepath{fill}
  \end{scope}
\end{tikzpicture}
```

44.4 Three Dimensional Coordinates

It is also possible to specify a point as a multiple of three vectors, the x-, y-, and z-vector. This is useful for creating simple three dimensional graphics.

\pgfpointxyz{⟨s_x⟩}{⟨s_y⟩}{⟨s_z⟩}

Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector plus s_z times the z-vector.



```
\begin{pgfpicture}
  \pgfsetarrowsend{to}

  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{0}{0}{1}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{0}{1}{0}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{1}{0}{0}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

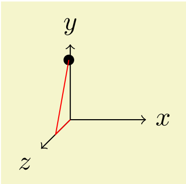
\pgfsetzvec{⟨*point*⟩}

Works like `\pgfsetzvec`.

Inside the xyz-coordinate system, you can also specify points using spherical and cylindrical coordinates.

`\pgfpointcylindrical{`⟨*degree*⟩`}{`⟨*radius*⟩`}{`⟨*height*⟩`}`

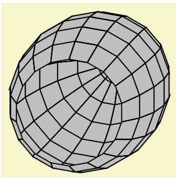
This command yields the same as
`\pgfpointadd{\pgfpointpolarxy{degree}{radius}}{\pgfpointxyz{0}{0}{height}}`




```
\begin{tikzpicture}
  \draw [->] (0,0) -- (1,0,0) node [right] {$x$};
  \draw [->] (0,0) -- (0,1,0) node [above] {$y$};
  \draw [->] (0,0) -- (0,0,1) node [below left] {$z$};

  \pgfpathcircle{\pgfpointcylindrical{80}{1}{.5}}{2pt}
  \pgfusepath{fill}

  \draw[red] (0,0) -- (0,0,.5) -- +(80:1);
\end{tikzpicture}
```

`\pgfpointspherical{`⟨*longitude*⟩`}{`⟨*latitude*⟩`}{`⟨*radius*⟩`}`

This command yields a point "on the surface of the earth" specified by the ⟨*longitude*⟩ and the `{`⟨*latitude*⟩`}`. The radius of the earth is given by ⟨*radius*⟩. The equator lies in the xy-plane.



```
\begin{tikzpicture}
  \pgfsetfillcolor{lightgray}

  \foreach \latitude in {-90,-75,...,30}
  {
    \foreach \longitude in {0,20,...,360}
    {
      \pgfpathmoveto{\pgfpointspherical{\longitude}{\latitude}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude+15}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude}{\latitude+15}{1}}
      \pgfpathclose
    }
    \pgfusepath{fill,stroke}
  }
\end{tikzpicture}
```

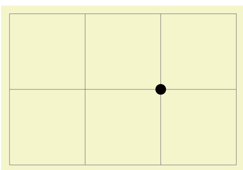
44.5 Building Coordinates From Other Coordinates

Many commands allow you to construct a coordinate in terms of other coordinates.

44.5.1 Basic Manipulations of Coordinates

`\pgfpointadd{`⟨v_1⟩`}{`⟨v_2⟩`}`

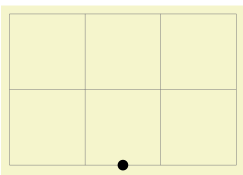
Returns the sum vector ⟨v_1⟩ + ⟨v_2⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointadd{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

`\pgfpointscale{`⟨*factor*⟩`}{`⟨*coordinate*⟩`}`

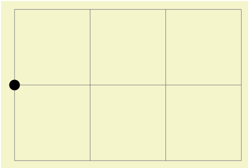
Returns the vector ⟨*factor*⟩⟨*coordinate*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointscale{1.5}{\pgfpoint{1cm}{0cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

`\pgfpointdiff{⟨start⟩}{⟨end⟩}`

Returns the difference vector ⟨*end*⟩ − ⟨*start*⟩.

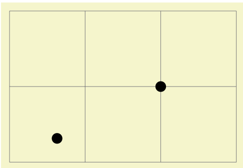


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointdiff{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

`\pgfpointnormalised{⟨point⟩}`

This command returns a normalized version of ⟨*point*⟩, that is, a vector of length 1pt pointing in the direction of ⟨*point*⟩. If ⟨*point*⟩ is the 0-vector or extremely short, a vector of length 1pt pointing upwards is returned.

This command is *not* implemented by calculating the length of the vector, but rather by calculating the angle of the vector and then using (something equivalent to) the `\pgfpointpolar` command. This ensures that the point will really have length 1pt, but it is not guaranteed that the vector will *precisely* point in the direction of ⟨*point*⟩ due to the fact that the polar tables are accurate only up to one degree. Normally, this is not a problem.



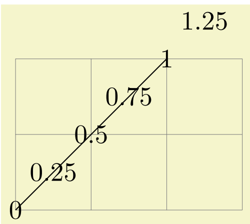
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointscale{20}
    {\pgfpointnormalised{\pgfpoint{2cm}{1cm}}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

44.5.2 Points Traveling along Lines and Curves

The commands in this section allow you to specify points on a line or a curve. Imaging a point "traveling" along a curve from some point p to another point q. At time $t = 0$ the point is at p and at time $t = 1$ it is at q and at time, say, $t = 1/2$ it is "somewhere in the middle." The exact location at time $t = 1/2$ will not necessarily be the "halfway point," that is, the point whose distance on the curve from p and q is equal. Rather, the exact location will depend on the "speed" at which the point is traveling, which in turn depends on the lengths of the support vectors in a complicated manner. If you are interested in the details, please see a good book on Bézier curves.

`\pgfpointlineattime{⟨time t⟩}{⟨point p⟩}{⟨point q⟩}`

Yields a point that is the tth fraction between p and q, that is, $p + t(q - p)$. For $t = 1/2$ this is the middle of p and q.

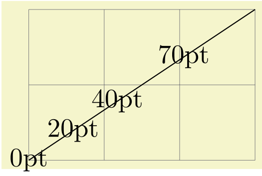



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{2cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,...,1.25}
    {\pgftext[at=
      \pgfpointlineattime{\t}{\pgfpointorigin}{\pgfpoint{2cm}{2cm}}]{\t}}
\end{tikzpicture}
```

`\pgfpointlineatdistance{⟨distance⟩}{⟨start point⟩}{⟨end point⟩}`

Yields a point that is located ⟨*distance*⟩ many units removed from the start point in the direction of the end point. In other words, this is the point that results if we travel ⟨*distance*⟩ steps from ⟨*start point*⟩ towards ⟨*end point*⟩.

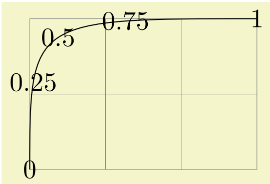
Example:




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \d in {0pt,20pt,40pt,70pt}
    {\pgftext[at=
      \pgfpointlineatdistance{\d}{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}]{\d}}
\end{tikzpicture}
```

\pgfpointcurveattime{⟨*time t*⟩}{⟨*point p*⟩}{⟨*point s_1*⟩}{⟨*point s_2*⟩}{⟨*point q*⟩}

Yields a point that is on the Bézier curve from p to q with the support points s_1 and s_2. The time t is used to determine the location, where $t = 0$ yields p and $t = 1$ yields q.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,0.5,0.75,1}
    {\pgftext[at=\pgfpointcurveattime{\t}{\pgfpointorigin}
                                        {\pgfpoint{0cm}{2cm}}
                                        {\pgfpoint{0cm}{2cm}}
                                        {\pgfpoint{3cm}{2cm}}]{\t}}
\end{tikzpicture}
```

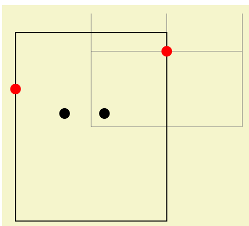
44.5.3 Points on Borders of Objects

The following commands are useful for specifying a point that lies on the border of special shapes. They are used, for example, by the shape mechanism to determine border points of shapes.

\pgfpointborderrectangle{⟨*direction point*⟩}{⟨*corner*⟩}

This command returns a point that lies on the intersection of a line starting at the origin and going towards the point ⟨*direction point*⟩ and a rectangle whose center is in the origin and whose upper right corner is at ⟨*corner*⟩.

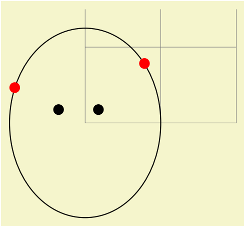
The ⟨*direction point*⟩ should have length "about 1pt," but it will be normalized automatically. Nevertheless, the "nearer" the length is to 1pt, the less rounding errors.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,1.5);
  \pgfpathrectanglecorners{\pgfpoint{-1cm}{-1.25cm}}{\pgfpoint{1cm}{1.25cm}}
  \pgfusepath{stroke}

  \pgfpathcircle{\pgfpoint{5pt}{5pt}}{2pt}
  \pgfpathcircle{\pgfpoint{-10pt}{5pt}}{2pt}
  \pgfusepath{fill}
  \color{red}
  \pgfpathcircle{\pgfpointborderrectangle
    {\pgfpoint{5pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfpathcircle{\pgfpointborderrectangle
    {\pgfpoint{-10pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointborderellipse{⟨*direction point*⟩}{⟨*corner*⟩}

This command works like the corresponding command for rectangles, only this time the ⟨*corner*⟩ is the corner of the bounding rectangle of an ellipse.

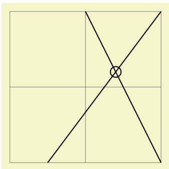


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,1.5);
  \pgfpathellipse{\pgfpointorigin}{\pgfpoint{1cm}{0cm}}{\pgfpoint{0cm}{1.25cm}}
  \pgfusepath{stroke}

  \pgfpathcircle{\pgfpoint{5pt}{5pt}}{2pt}
  \pgfpathcircle{\pgfpoint{-10pt}{5pt}}{2pt}
  \pgfusepath{fill}
  \color{red}
  \pgfpathcircle{\pgfpointborderellipse
    {\pgfpoint{5pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfpathcircle{\pgfpointborderellipse
    {\pgfpoint{-10pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

44.5.4 Points on the Intersection of Lines

`\pgfpointintersectionoflines`{⟨*p*⟩}{⟨*q*⟩}{⟨*s*⟩}{⟨*t*⟩}

This command returns the intersection of a line going through p and q and a line going through s and t. If the lines do not intersection, an arithmetic overflow will occur.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \draw (.5,0) -- (2,2);
  \draw (1,2) -- (2,0);
  \pgfpathcircle{%
    \pgfpointintersectionoflines
      {\pgfpointxy{.5}{0}}{\pgfpointxy{2}{2}}
      {\pgfpointxy{1}{2}}{\pgfpointxy{2}{0}}}
    {2pt}
  \pgfusepath{stroke}
\end{tikzpicture}
```

44.6 Extracting Coordinates

There are two commands that can be used to "extract" the x- or y-coordinate of a coordinate.

`\pgfextractx`{⟨*dimension*⟩}{⟨*point*⟩}

Sets the TEX-⟨*dimension*⟩ to the x-coordinate of the point.

```
\newdimen\mydim
\pgfextractx{\mydim}{\pgfpoint{2cm}{4pt}}
%% \mydim is now 2cm
```

`\pgfextracty`{⟨*dimension*⟩}{⟨*point*⟩}

Like `\pgfextractx`, except for the y-coordinate.

44.7 Internals of How Point Commands Work

As a normal user of PGF you do not need to read this section. It is relevant only if you need to understand how the point commands work internally.

When a command like `\pgfpoint{1cm}{2pt}` is called, all that happens is that the two TEX-dimension variables `\pgf@x` and `\pgf@y` are set to `1cm` and `2pt`, respectively. A command like `\pgfpathmoveto` that takes a coordinate as parameter will just execute this parameter and then use the values of `\pgf@x` and `\pgf@y` as the coordinates to which it will move the pen on the current path.

since commands like `\pgfpointnormalised` modify other variables besides `\pgf@x` and `\pgf@y` during the computation of the final values of `\pgf@x` and `\pgf@y`, it is a good idea to enclose a call of a command like `\pgfpoint` in a TEX-scope and then make the changes of `\pgf@x` and `\pgf@y` global as in the following example:

```
...
{ % open scope
  \pgfpointnormalised{\pgfpoint{1cm}{1cm}}
  \global\pgf@x=\pgf@x % make the change of \pgf@x persist past the scope
  \global\pgf@y=\pgf@y % make the change of \pgf@y persist past the scope
}
% \pgf@x and \pgf@y are now set correctly, all other variables are
% unchanged
```

Since this situation arises very often, the macro `\pgf@process` can be used to perform the above code:

`\pgf@process{`⟨*code*⟩`}`

Executes the ⟨*code*⟩ in a scope and then makes `\pgf@x` and `\pgf@y` global.

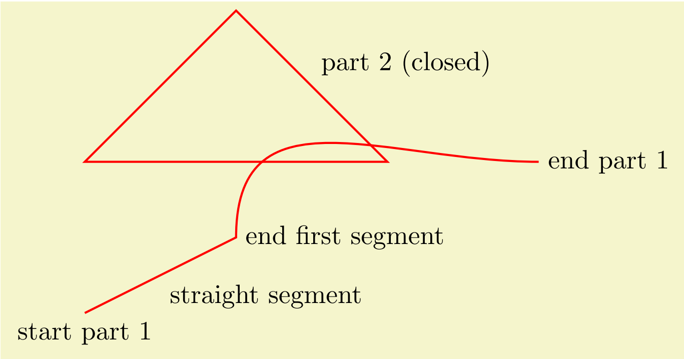
Note that this macro is used often internally. For this reason, it is not a good idea to keep anything important in the variables `\pgf@x` and `\pgf@y` since they will be overwritten and changed frequently. Instead, intermediate values can ge stored in the TEX-dimensions `\pgf@xa`, `\pgf@xb`, `\pgf@xc` and their `y`-counterparts `\pgf@ya`, `\pgf@yb`, `pgf@yc`. For example, here is the code of the command `\pgfpointadd`:

```
\def\pgfpointadd#1#2{%
  \pgf@process{#1}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#2}%
  \advance\pgf@x by\pgf@xa%
  \advance\pgf@y by\pgf@ya}
```

45 Constructing Paths

45.1 Overview

The "basic entity of drawing" in PGF is the *path*. A path consists of several parts, each of which is either a closed or open curve. An open curve has a starting point and an end point and, in between, consists of several *segments*, each of which is either a straight line or a Bézier curve. Here is an example of a path (in red) consisting of two parts, one open, one closed:




```
\begin{tikzpicture}[scale=2]
  \draw[thick,red]
       (0,0) coordinate (a)
    -- coordinate (ab) (1,.5) coordinate (b)
    .. coordinate (bc) controls +(up:1cm) and +(left:1cm) .. (3,1)  coordinate (c)
       (0,1) -- (2,1) -- coordinate (x) (1,2) -- cycle;

  \draw (a)  node[below] {start part 1}
        (ab) node[below right] {straight segment}
        (b)  node[right] {end first segment}
        (c)  node[right] {end part 1}
        (x)  node[above right]  {part 2 (closed)};
\end{tikzpicture}
```

A path, by itself, has no "effect," that is, it does not leave any marks on the page. It is just a set of points on the plane. However, you can *use* a path in different ways. The most natural actions are *stroking* (also known as *drawing*) and *filling*. Stroking can be imagined as picking up a pen of a certain diameter and "moving it along the path." Filling means that everything "inside" the path is filled with a uniform color. Naturally, the open parts of a path must first be closed before a path can be filled.

In PGF, there are numerous commands for constructing paths, all of which start with `\pgfpath`. There are also commands for *using* paths, though most operations can be performed by calling `\pgfusepath` with an appropriate parameter.

As a side-effect, the path construction commands keep track of two bounding boxes. One is the bounding box for the current path, the other is a bounding box for all paths in the current picture. See Section 45.13 for more details.

Each path construction command extends the current path in some way. The "current path" is a global entity that persists across TeX groups. Thus, between calls to the path construction commands you can perform arbitrary computations and even open and closed TeX groups. The current path only gets "flushed" when the `\pgfusepath` command is called (or when the soft-path subsystem is used directly, see Section 61).

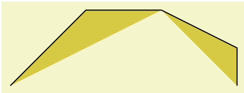
45.2 The Move-To Path Operation

The most basic operation is the move-to operation. It must be given at the beginning of paths, though some path construction command (like `\pgfpathrectangle`) generate move-tos implicitly. A move-to operation can also be used to start a new part of a path.

`\pgfpathmoveto`{⟨*coordinate*⟩}

This command expects a PGF-coordinate like `\pgfpointorigin` as its parameter. When the current path is empty, this operation will start the path at the given ⟨*coordinate*⟩. If a path has already been partly constructed, this command will end the current part of the path and start a new one.

```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}} % New part
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

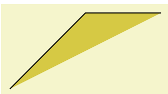
The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

The command will update the bounding box of the current path and picture, if necessary.

45.3 The Line-To Path Operation

`\pgfpathlineto{`⟨*coordinate*⟩`}`

This command extends the current path in a straight line to the given ⟨*coordinate*⟩. If this command is given at the beginning of path without any other path construction command given before (in particular without a move-to operation), the TeX file may compile without an error message, but a viewer application may display an error message when trying to render the picture.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

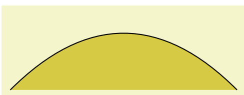
The command will update the bounding box of the current path and picture, if necessary.

45.4 The Curve-To Path Operation

`\pgfpathcurveto{`⟨*support 1*⟩`}{`⟨*support 2*⟩`}{`⟨*coordinate*⟩`}`

This command extends the current path with a Bézier curve from the last point of the path to ⟨*coordinate*⟩. The ⟨*support 1*⟩ and ⟨*support 2*⟩ are the first and second support point of the Bézier curve. For more information on Bézier curve, please consult a standard textbook on computer graphics.

Like the line-to command, this command may not be the first path construction command in a path.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{1cm}{1cm}}{\pgfpoint{2cm}{1cm}}{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

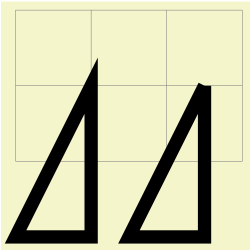
The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

The command will update the bounding box of the current path and picture, if necessary. However, the bounding box is simply made large enough such that it encompasses all of the support points and the ⟨*coordinate*⟩. This will guarantee that the curve is completely inside the bounding box, but the bounding box will typically be quite a bit too large. It is not clear (to me) how this can be avoided without resorting to "some serious math" in order to calculate a precise bounding box.

45.5 The Close Path Operation

`\pgfpathclose`

This command closes the current part of the path by appending a straight line to the start point of the current part. Note that there *is* a difference between closing a path and using the line-to operation to add a straight line to the start of the current path. The difference is demonstrated by the upper corners of the triangles in the following example:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{5pt}
  \pgfpathmoveto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{0cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{-1cm}}
  \pgfpathclose
  \pgfpathmoveto{\pgfpoint{2.5cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{1cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

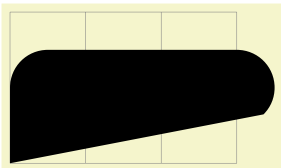
45.6 Arc, Ellipse and Circle Path Operations

The path construction commands that we have discussed up to now are sufficient to create all paths that can be created "at all." However, it is useful to have special commands to create certain shapes, like circles, that arise often in practice.

In the following, the commands for adding (parts of) (transformed) circles to a path are described.

`\pgfpatharc`{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*radius*⟩/⟨*y-radius*⟩}

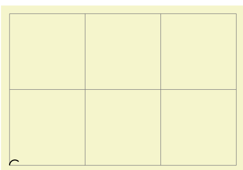
This command appends a part of a circle (or an ellipse) to the current path. Imaging the curve between ⟨*start angle*⟩ and ⟨*end angle*⟩ on a circle of radius ⟨*radius*⟩ (if ⟨*start angle*⟩ < ⟨*end angle*⟩, the curve goes around the circle counterclockwise, otherwise clockwise). This curve is now moved such that the point where the curve starts is the previous last point of the path. Note that this command will *not* start a new part of the path, which is important for example for filling purposes.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfpatharc{180}{90}{.5cm}
  \pgfpathlineto{\pgfpoint{3cm}{1.5cm}}
  \pgfpatharc{90}{-45}{.5cm}
  \pgfusepath{fill}
\end{tikzpicture}
```

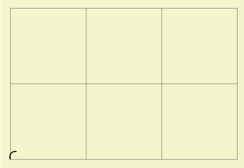
Saying `\pgfpatharc{0}{360}{1cm}` "nearly" gives you a full circle. The "nearly" refers to the fact that the circle will not be closed. You can close it using `\pgfpathclose`.

If the optional ⟨*y-radius*⟩ is given, the ⟨*radius*⟩ is the x-radius and the ⟨*y-radius*⟩ the y-radius of the ellipse from which the curve is taken:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm/1cm}
  \pgfusepath{draw}
\end{tikzpicture}
```

The axes of the circle or ellipse from which the arc is "taken" always point up and right. However, the current coordinate transformation matrix will have an effect on the arc. This can be used to, say, rotate an arc:



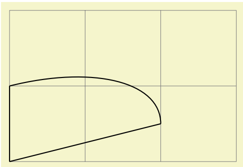
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm/1cm}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will update the bounding box of the current path and picture, if necessary. Unless rotation or shearing transformations are applied, the bounding box will be tight.

\pgfpatharcaxes{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*first axis*⟩}{⟨*second axis*⟩}

This command is similar to `\pgfpatharc`. The main difference is how the ellipse or circle is specified from which the arc is taken. The two parameters ⟨*first axis*⟩ and ⟨*second axis*⟩ are the 0°-axis and the 90°-axis of the ellipse from which the path is taken. Thus, `\pgfpatharc{0}{90}{1cm/2cm}` has the same effect as

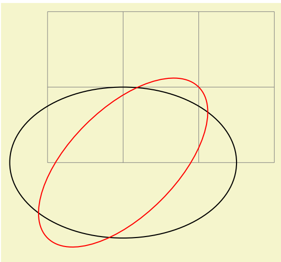
`\pgfpatharcaxes{0}{90}{\pgfpoint{1cm}{0cm}}{\pgfpoint{0cm}{2cm}}`



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2cm,5mm) (0,0) -- (0cm,1cm);

  \pgfpathmoveto{\pgfpoint{2cm}{5mm}}
  \pgfpatharcaxes{0}{90}{\pgfpoint{2cm}{5mm}}{\pgfpoint{0cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

\pgfpathellipse{⟨*center*⟩}{⟨*first axis*⟩}{⟨*second axis*⟩}

The effect of this command is to append an ellipse to the current path (if the path is not empty, a new part is started). The ellipse's center will be ⟨*center*⟩ and ⟨*first axis*⟩ and ⟨*second axis*⟩ are the axis *vectors*. The same effect as this command can also be achieved using an appropriate sequence of move-to, arc, and close operations, but this command is easier and faster.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathellipse{\pgfpoint{1cm}{0cm}}
                 {\pgfpoint{1.5cm}{0cm}}
                 {\pgfpoint{0cm}{1cm}}
  \pgfusepath{draw}
  \color{red}
  \pgfpathellipse{\pgfpoint{1cm}{0cm}}
                 {\pgfpoint{1cm}{1cm}}
                 {\pgfpoint{-0.5cm}{0.5cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations to all coordinates of the ellipse. However, the coordinate transformations are applied only after the ellipse is "finished conceptually." Thus, a transformation of 1cm to the right will simply shift the ellipse one centimeter to the right; it will not add 1cm to the x-coordinates of the two axis vectors.

The command will update the bounding box of the current path and picture, if necessary.

\pgfpathcirlce{⟨*center*⟩}{⟨*radius*⟩}

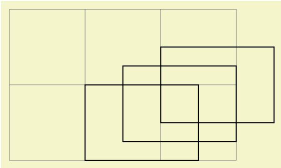
A shorthand for `\pgfpathellipse` applied to ⟨*center*⟩ and the two axis vectors (⟨*radius*⟩, 0) and (0, ⟨*radius*⟩).

45.7 Rectangle Path Operations

Another shape that arises frequently is the rectangle. Two commands can be used to add a rectangle to the current path. Both commands will start a new part of the path.

\pgfpathrectangle{⟨*corner*⟩}{⟨*diagonal vector*⟩}

Adds a rectangle to the path whose one corner is ⟨*corner*⟩ and whose opposite corner is given by ⟨*corner*⟩ + ⟨*diagonal vector*⟩.

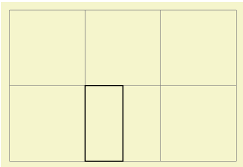


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectangle{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1.5cm}{0.25cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{2cm}{0.5cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

`\pgfpathrectanglecorners{`⟨*corner*⟩`}{`⟨*opposite corner*⟩`}`

Adds a rectangle to the path whose two opposing corners are ⟨*corner*⟩ and ⟨*opposite corner*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectanglecorners{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

45.8 The Grid Path Operation

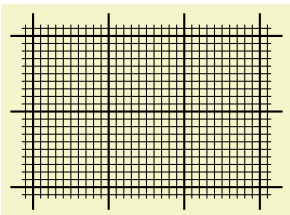
`\pgfpathgrid[`⟨*options*⟩`]{`⟨*lower left*⟩`}{`⟨*upper right*⟩`}`

Appends a grid to the current path. That is, a (possibly large) number of parts are added to the path, each part consisting of a single horizontal or vertical straight line segment.

Conceptually, the origin is part of the grid and the grid is clipped to the rectangle specified by the ⟨*lower left*⟩ and the ⟨*upper right*⟩ corner. However, no clipping occurs (this command just adds parts to the current path). Rather, the points where the lines enter and leave the "clipping area" are computed and used to add simple lines to the current path.

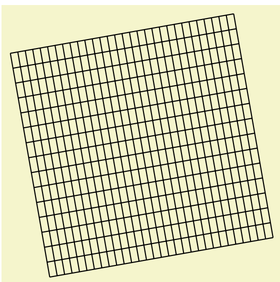
Allowed ⟨*options*⟩ are:

- `stepx=`⟨*dimension*⟩ Sets the horizontal stepping to ⟨*dimension*⟩. Default is 1cm.
- `stepy=`⟨*dimension*⟩ Sets the vertical stepping to ⟨*dimension*⟩. Default is 1cm.
- `step=`⟨*vector*⟩ Sets the horizontal stepping to the x-coordinate of ⟨*vector*⟩ and the vertical stepping to its y-coordinate.



```
\begin{pgfpicture}
  \pgfsetlinewidth{0.8pt}
  \pgfpathgrid[step={\pgfpoint{1cm}{1cm}}]
    {\pgfpoint{-3mm}{-3mm}}{\pgfpoint{33mm}{23mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{0.4pt}
  \pgfpathgrid[stepx=1mm,stepy=1mm]
    {\pgfpoint{-1.5mm}{-1.5mm}}{\pgfpoint{31.5mm}{21.5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly. As for ellipses, the transformations are applied to the "conceptually finished" grid.



```
\begin{pgfpicture}
  \pgftransformrotate{10}
  \pgfpathgrid[stepx=1mm,stepy=2mm]{\pgfpoint{0mm}{0mm}}{\pgfpoint{30mm}{30mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

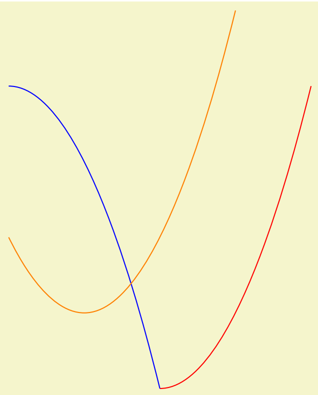
45.9 The Parabola Path Operation

\pgfpathparabola{⟨*bend vector*⟩}{⟨*end vector*⟩}

This command appends two half-parabolas to the current path. The first starts at the current point and ends at the current point plus ⟨*bend vector*⟩. At his point, it has its bend. The second half parabola starts at that bend point and end at point that is given by the bend plus ⟨*end vector*⟩.

If you set ⟨*end vector*⟩ to the null vector, you append only a half parabola that goes from the current point to the bend; by setting ⟨*bend vector*⟩ to the null vector, you append only a half parabola that goes to current point plus ⟨*end vector*⟩ and has its bend at the current point.

It is not possible to use this command to draw a part of a parabola that does not contain the bend.



```
\begin{pgfpicture}
  % Half-parabola going ``up and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpointorigin}{\pgfpoint{2cm}{4cm}}
  \color{red}
  \pgfusepath{stroke}

  % Half-parabola going ``down and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpoint{-2cm}{4cm}}{\pgfpointorigin}
  \color{blue}
  \pgfusepath{stroke}

  % Full parabola
  \pgfpathmoveto{\pgfpoint{-2cm}{2cm}}
  \pgfpathparabola{\pgfpoint{1cm}{-1cm}}{\pgfpoint{2cm}{4cm}}
  \color{orange}
  \pgfusepath{stroke}
\end{pgfpicture}
```

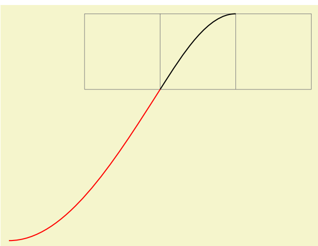
The command will apply coordinate transformations and update the bounding boxes.

45.10 Sine and Cosine Path Operations

Sine and cosine curves often need to be drawn and the following commands may help with this. However, they only allow you to append sine and cosine curves in intervals that are multiples of $\pi/2$.

\pgfpathsine{⟨*vector*⟩}

This command appends a sine curve in the interval $[0, \pi/2]$ to the current path. The sine curve is squeezed or stretched such that the curve starts at the current point and ends at the current point plus ⟨*vector*⟩.

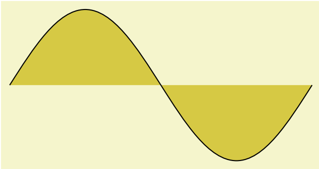


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,1);
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}

  \color{red}
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{-2cm}{-2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes.

\pgfpathcosine{⟨*vector*⟩}

This command appends a cosine curve in the interval $[0, \pi/2]$ to the current path. The curve is squeezed or stretched such that the curve starts at the current point and ends at the current point plus ⟨*vector*⟩. Using several sine and cosine operations in sequence allows you to produce a complete sine or cosine curve



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{-1cm}}
  \pgfpathsine{\pgfpoint{1cm}{-1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{1cm}}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply coordinate transformations and update the bounding boxes.

45.11 Plot Path Operations

There exist several commands for appending plots to a path. These commands are available through the package `pgfbaseplot`. They are documented in Section 56.

45.12 Rounded Corners

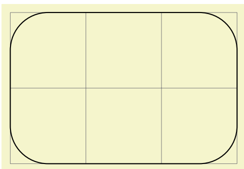
Normally, when you connect two straight line segments or when you connect two curves that end and start "at different angles" you get "sharp corners" between the lines or curves. In some cases it is desirable to produce "rounded corners" instead. Thus, the lines or curves should be shortened a bit and then connected by arcs.

PGF offers an easy way to achieve this effect, by calling the following two commands.

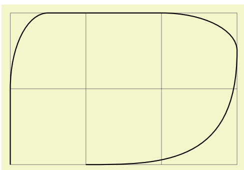
`\pgfsetcornersarced{`⟨*point*⟩`}`

This command causes all subsequent corners to be replaced by little arcs. The effect of this command lasts till the end of the current TeX scope.

The ⟨*point*⟩ dictates how large the corner arc will be. Consider a corner made by two lines l and r and assume that the line l comes first on the path. The x-dimension of the ⟨*point*⟩ decides by how much the line l will be shortened, the y-dimension of ⟨*point*⟩ decides by how much the line r will be shortened. Then, the shortened lines are connected by an arc.



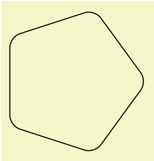
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfsetcornersarced{\pgfpoint{5mm}{5mm}}
  \pgfpathrectanglecorners{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfsetcornersarced{\pgfpoint{10mm}{5mm}}
  % 10mm entering,
  % 5mm leaving.
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{2cm}}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfpathcurveto
    {\pgfpoint{3cm}{0cm}}
    {\pgfpoint{2cm}{0cm}}
    {\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

If the x- and y-coordinates of ⟨*point*⟩ are the same and the corner is a right angle, you will get a perfect quarter circle (well, not quite perfect, but perfect up to six decimals). When the angle is not 90°, you only get a fair approximation.

More or less "all" corners will be rounded, even the corner generated by a `\pgfpathclose` command. (The author is a bit proud of this feature.)



```
\begin{pgfpicture}
  \pgfsetcornersarced{\pgfpoint{4pt}{4pt}}
  \pgfpathmoveto{\pgfpointpolar{0}{1cm}}
  \pgfpathlineto{\pgfpointpolar{72}{1cm}}
  \pgfpathlineto{\pgfpointpolar{144}{1cm}}
  \pgfpathlineto{\pgfpointpolar{216}{1cm}}
  \pgfpathlineto{\pgfpointpolar{288}{1cm}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}
```

To return to normal (unrounded) corners, use `\pgfsetcornersarced{\pgfpointorigin}`.

Note that the rounding will produce strange and undesirable effects if the lines at the corners are too short. In this case the shortening may cause the lines to "suddenly extend over the other end" which is rarely desirable.

45.13 Internal Tracking of Bounding Boxes for Paths and Pictures

The path construction commands keep track of two bounding boxes: One for the current path, which is reset whenever the path is used and thereby flushed, and a bounding box for the current `{pgfpicture}`.

The bounding boxes are not accessible by "normal" macros. Rather, two sets of four dimension variables are used for this, all of which contain the letter `@`.

`\pgf@pathminx`

The minimum x-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

`\pgf@pathmaxx`

The maximum x-coordinate "mentioned" in the current path. Initially, this is set to −16000pt.

`\pgf@pathminy`

The minimum y-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

`\pgf@pathmaxy`

The maximum y-coordinate "mentioned" in the current path. Initially, this is set to −16000pt.

`\pgf@picminx`

The minimum x-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

`\pgf@picmaxx`

The maximum x-coordinate "mentioned" in the current picture. Initially, this is set to −16000pt.

`\pgf@picminy`

The minimum y-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

`\pgf@picmaxy`

The maximum y-coordinate "mentioned" in the current picture. Initially, this is set to −16000pt.

Each time a path construction command is called, the above variables are (globally) updated. To facilitate this, you can use the following command:

`\pgf@protocolsizes{`⟨*x-dimension*⟩`}{`⟨*y-dimension*⟩`}`

Updates all of the above dimension in such a way that the point specified by the two arguments is inside both bounding boxes. For the picture's bounding box this updating occurs only if `\ifpgf@relevantforpicturesize` is true, see below.

For the bounding box of the picture it is not always desirable that every path construction command affects this bounding box. For example, if you have just used a clip command, you do not want anything outside the clipping area to affect the bounding box. For this reason, there exists a special "TEX if" that (locally) decides whether updating should be applied to the picture's bounding box. Clipping will set this if to false, as will certain other commands.

`\pgf@relevantforpicturesizefalse`

Suppresses updating of the picture's bounding box.

`\pgf@relevantforpicturesizetrue`

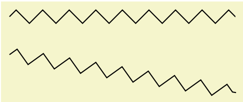
Causes updating of the picture's bounding box.

46 Snakes

46.1 Overview

A *snake* is a "way of adding a winding line to a path." To be a bit more precise, you use snakes to extend the path and the commands for using snakes start with `\pgfpath`. However, snakes do not necessarily extend the path using line-to and curve-to operations; rather, they can also contain move-to operations and, thereby, cause the path to be split into many subpaths.

As an example, let us consider a simple snake like the `zigzag` snake. It looks like this:



```
\begin{tikzpicture}
  \draw[snake=zigzag] (0,0)   -- (3,0);
  \draw[snake=zigzag] (0,-.5) -- (3,-1);
\end{tikzpicture}
```

The above example demonstrates the two key features of snakes:

1. Snakes are made up from little segments that are repeated several times. For the `zigzag` snake these segements look like this: .

2. Snakes "follow along a straight to the target point." Thus, when the target point is not to the right of the start point, the snake is rotated appropriately.

In order to use snakes, they first have to be *declared*. This declaration contains a detailed description of how each segement of the snake looks like and in what ordering the different possible segments are used.

Once a snake has been declared, it can be used. For this you specify a target point and a snaked line will be added from the last point current point to the target point.

46.2 Declaring a Snake

46.2.1 Segments

When you declare a snake, you provide a description of how the different segments of the snake will be rendered. The description of each segment should be given in a way as if the target of the snaked line where at $(+\infty, 0)$ and as if the segment's start where at the origin. Thus, for example, the segment of the `zigzag` might be defined using the following code:

```
\pgfpathlineto{\pgfpoint{5pt}{5pt}}
\pgfpathlineto{\pgfpoint{15pt}{-5pt}}
\pgfpathlineto{\pgfpoint{20pt}{0pt}}
```

PGF will ensure that an appropriate coordinate transformation is in place when the snake segment is added to the path such that the snake segment actually points in the right direction. Also subsequent snake segments will be transformed such that they are "further along the line" toward the target. All transformations are setup automatically.

Note that we did not use a `\pgfpathmoveto{\pgfpointorigin}` at the beginning of the segment code. Doing so would subdivide the path into numerous subpath. Rather, we assume that the previous segment caused the current point to be at the origin.

While we can now describe segments, we still need a way to "stop" the snake. Also, PGF needs to know the width of the different snake segments such that it can translate the next segment correctly. Even though it would be possible to compute all this information automatically, PGF does not do so and you have to provide the information "by hand" for each segment.

46.2.2 Snake Automata

Up to now our snakes only have one segment that is repeated again and again. However, we might also like to have *different* segments and use rules to describe which segment should be used where. For example, we might have special segments at the start and at the end.

For snakes we use a mechanism known in computer science as *finite automata* to describe which segment is used in each part of the snake. The idea is the following: For the first segment we start in a special *state* called the *initial state*. In this state, and also in all other state later, PGF first computes how much space is left on the snake. That is, PGF keeps track of the distance to the target. Attached to each state there is a set of rules of the following form: "If the remaining distance is less than x, switch to state q." PGF checks for each of these rules whether it applies and, if so, immediately switches to state q.

Only if none of the rules tell us to switch to another state, PGF will execute the state's code. This code will add a segment to the path. In addition to the rules there is also width parameter attached to each state. PGF then translates the coordinate system by this width and reduces the remaining distance. Then, PGF either stays in the current state or switches to another state, depending on yet another property attached of the state.

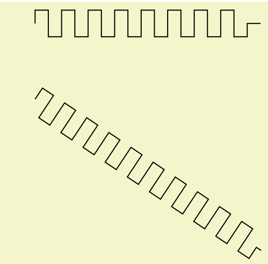
The whole process stops when a special state called `final` is reached. The segment of this state is added to the path immediately (it is often empty, though) and the process ends.

46.2.3 The Snake Declaration Command

`\pgfdeclaresnake{`⟨*name*⟩`}{`⟨*initial state*⟩`}{`⟨*states*⟩`}`

This command declares a new snake called ⟨*name*⟩. The ⟨*states*⟩ argument contains a description of the snake automaton's states and the transitions between them. The ⟨*initial state*⟩ is the state in which the automaton starts.

The ⟨*states*⟩ argument should consist of `\state` commands, one for each state of the snake automaton. The `\state` command is defined only when the ⟨*states*⟩ argument is executed.



```
\pgfdeclaresnake{example snake}{initial}
{
  \state{initial}[width=10pt]
  {
    \pgfpathlineto{\pgfpoint{0pt}{5pt}}
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
    \pgfpathlineto{\pgfpoint{5pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{10pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{10pt}{0pt}}
  }
  \state{final}
  {
    \pgfpathlineto{\pgfpoint{\pgfsnakeremainingdistance}{0pt}}
  }
}
\tikz{\draw[snake=example snake] (0,0) -- (3,0);
      \draw[snake=example snake] (0,-1) -- (3,-3);}
```

The `\state` command works as follows:

`\state{`⟨*name*⟩`}[`⟨*options*⟩`]{`⟨*code*⟩`}`

This command declares a new state inside the current snake automaton. The state is names ⟨*name*⟩.

When PGF is in state ⟨*name*⟩, the following things happen:

1. The ⟨*options*⟩ are parsed. This may lead, see below, to a state switch. When this happens, the following steps are not executed. The ⟨*options*⟩ are executed one after the other in the given order. If an option causes a state switch, the switch is immediate, even if later options might cause a different state switch.
2. The ⟨*code*⟩ is executed. When this happens, the coordinate transformation will be setup such that the origin is at the start of the segment and such that the target lies in the direction "east."
3. After the ⟨*code*⟩ has been executed, the state switches to whatever state has been specified inside the ⟨*options*⟩ using the `next state` option. If no `next state` has been specified, the state stays the same.

The following options are allowed inside the ⟨*options*⟩:

- `switch if less than=`⟨*dimension*⟩ `to` ⟨*new state*⟩ When this option is encountered, PGF checks whether the remaining distance to the target is less than ⟨*dimension*⟩. If so, an immediate state switch to ⟨*new state*⟩ occurs.
- `width=`⟨*dimension*⟩ This option tells PGF the width of the segment if it is appended to the path. The coordinate system will, afterward, be translated by ⟨*dimension*⟩ to ensure that the next segment starts where the current one ended.
  In addition, this option will cause an immediate switch to the state `final` if the remaining distance is less than ⟨*dimension*⟩. The effect is the same as if you had said `switch if less than=`⟨*dimension*⟩ `to final` just before the `width` option.

- next state=⟨*new state*⟩ After the segment has been added to the path, a state switch to ⟨*new state*⟩ is performed. If this option is not given, the next state is the same as the current state.

There are two dimensions that are useful inside snake automata:

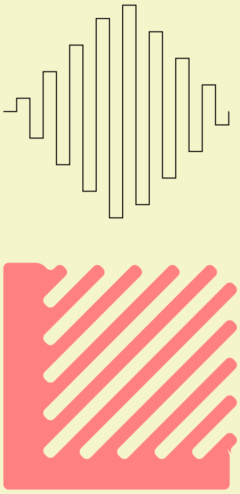
\pgfsnakeremainingdistance

This TeX dimension holds the remaining distance to the target.

\pgfsnakecompleteddistance

This TeX dimension holds the distance already completed on the snake.

As a final example we present a more complicated snake that makes use of the different options:



```
\pgfdeclaresnake{complicated example snake}{initial}
{
  \state{initial}[width=5pt,next state=up]
  { \pgfpathlineto{\pgfpoint{5pt}{0pt}} }

  \state{up}[width=5pt,next state=down]
  {
    \ifdim\pgfsnakeremainingdistance>\pgfsnakecompleteddistance
      % Growing
      \pgfpathlineto{\pgfpoint{0pt}{\pgfsnakecompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{\pgfsnakecompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \else
      % Shrinking
      \pgfpathlineto{\pgfpoint{0pt}{\pgfsnakeremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{\pgfsnakeremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \fi%
  }
  \state{down}[width=5pt,next state=up]
  {
    \ifdim\pgfsnakeremainingdistance>\pgfsnakecompleteddistance
      % Growing
      \pgfpathlineto{\pgfpoint{0pt}{-\pgfsnakecompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{-\pgfsnakecompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \else
      % Shrinking
      \pgfpathlineto{\pgfpoint{0pt}{-\pgfsnakeremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{-\pgfsnakeremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \fi%
  }
  \state{final}
  {
    \pgfpathlineto{\pgfpoint{\pgfsnakeremainingdistance}{0pt}}
  }
}
\begin{tikzpicture}[snake=complicated example snake]
  \draw[snake] (0,0) -- (3,0);
  \fill[red!50,snake,rounded corners=2pt] (.5,-2) -- ++(2.5,-2.5)
                              [snake=none] -- ++(0,-.5) -| (0,-2) -- cycle;
\end{tikzpicture}
```

46.2.4 Predefined Snakes

Only two very simple and basic snakes are predefined when you load PGF. For more interesting snakes you can use the package pgflibrarysnakes.

Snake lineto

This most trivial of all snakes is simply a straight line. This snake is typically not used in a "stand alone" fashion. Rather, it is useful as a "subsnake" of the \pgfsnakesto command.

Snake moveto

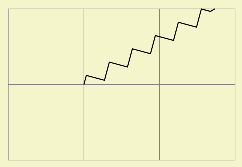
Arguably, this snake is even simpler than the previous snake. This snake consists of a simple move-to operation. Like the lineto snake, it is useful as a "subsnake" of the \pgfsnakesto command.

46.3 Using Snakes

Three commands can be used to use a snake.

`\pgfpathsnakealongvector{`⟨*snake*⟩`}{`⟨*length*⟩`}{`⟨*vector*⟩`}`

This command will append the ⟨*snake*⟩ to the current path. The length of the snake is given by ⟨*length*⟩. The ⟨*vector*⟩ should be a normalizes vector (a vector having length `1pt`) pointing in the direction in which the snake should grow.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpoint{1cm}{1cm}}
  \pgfpathsnakealongvector{zigzag}{2cm}{\pgfpointpolar{30}{1pt}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

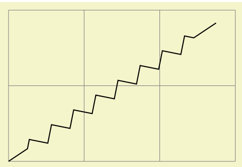
`\pgfpathsnakesto{`⟨*snake list*⟩`}{`⟨*target*⟩`}`

This command will append the snakes in the ⟨*snake list*⟩ to the current path such that it ends at ⟨*point*⟩. This command calls the previous one (repeatedly, possibly) after having computed the distance from the current point to ⟨*target*⟩ and normalized the vector connecting the current point to the target.

The ⟨*snake list*⟩ is a comma-separated list of pairs consisting of a snake name in curly braces and a distance in curly braces. For example, a snake list might be `{lineto}{1cm},{moveto}{2cm}`. This is interpreted as follows: Use a `lineto` snake for the first centimeter of the way to the ⟨*target*⟩, then use the `moveto` snake for the next two centimeters.

No attempt is made to check or ensure that the distances of the "subsnakes' add up to the distnace from the current point to the ⟨*target*⟩. However, it is possible to refer to this distance inside the ⟨*snake list*⟩: While the snake list is processed, the two TEX-dimensions `\pgfsnakeremainingdistance` and `\pgfsnakecompleteddistance` will be set to the correct values. For example, consider the example from a above, consisting of a `lineto` and a `moveto` snake. Suppose the distance to ⟨*target*⟩ where 5cm. Then, when the length of the first subsnake is computed, the remaining distance willbe set to 5cm and the completed distance to 0cm. When the length of the second subsnake is computed, the remaining distance wil be 4cm and the completed distance will be 1cm.

Here are some useful examles: `{zigzag}{\pgfsnakeremainingdistance}` is a ⟨*snake list*⟩ that consists only of a `zigzag` snake whose length is the total distance from the current point to the ⟨*target*⟩. Next, to get a snake that is a `zigzag` snake that is preceeded and succeeded by 3mm of `lineto`, you can use the following:

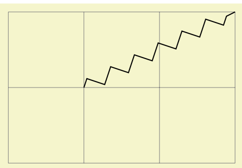


```
\begin{tikzpicture}
  \draw[help lines](0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathsnakesto{%
    {lineto}{3mm},%
    {zigzag}{\pgfsnakeremainingdistance-6mm},%
    {lineto}{3mm}}
  {\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

Note that the computation of the distance may be imprecise. In general, the placement precision of the snakes will not be perfect.

`\pgfpathsnaketo{`⟨*snake*⟩`}{`⟨*target*⟩`}`

This command is just a shortcut for calling `\pgfpathsnakesto` for a single snake whose length is `\pgfsnakeremainingdistance`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpoint{1cm}{1cm}}
  \pgfpathsnaketo{zigzag}{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

As was already mentioned, when each segment of the snake is added to the path, an appropriate coordinate transformation will be in force. It is sometimes useful to add an additional transformation locally. For example, by reflecting everything around the x-axis right before each segment is added, the snake will effectively be mirrored along the path. The following command allows you to install such a "last minute transformation."

`\pgfsetsnakesegmenttransformation{`⟨*code*⟩`}`

The ⟨*code*⟩ will be executed at the very beginning of each segment. Normally, this be a transformation command that changes the y-axis in some way.

47 Using Paths

47.1 Overview

Once a path has been constructed, it can be *used* in different ways. For example, you can draw the path or fill it or use it for clipping.

Numerous graph parameters influence how a path will be rendered. For example, when you draw a path, the line width is important as well as the dashing pattern. The options that govern how paths are rendered can all be set with commands starting with `\pgfset`. *All options that influence how a path is rendered always influence the complete path.* Thus, it is not possible to draw part of a path using, say, a red color and drawing another part using a green color. To achieve such an effect, you must use two paths.

In detail, paths can be used in the following ways:

1. You can *stroke* (also known as *draw*) a path.
2. You can *fill* a path with a uniform color.
3. You can *clip* subsequent renderings against the path.
4. You can *shade* a path.
5. You can *use the path as bounding box* for the whole picture.

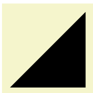
You can also perform any combination of the above, though it makes no sense to fill and shade a path at the same time.

To perform (a combination of) the first three actions, you can use the following command:

`\pgfusepath{`⟨*actions*⟩`}`

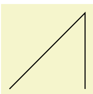
Applies the given ⟨*actions*⟩ to the current path. Afterwards, the current path is (globally) empty. The following actions are possible:

- `fill` fills the path. See Section 47.3 for further details.



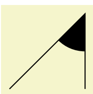
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{fill}
\end{pgfpicture}
```

- `stroke` strokes the path. See Section 47.2 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

- `clip` clips all subsequent drawings against the path. See Section 47.4 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke,clip}
  \pgfpathcircle{\pgfpoint{1cm}{1cm}}{0.5cm}
  \pgfusepath{fill}
\end{pgfpicture}
```

- `discard` discards the path, that is, it is not used at all. Giving this option (alone) has the same effect as giving an empty options list.

When more than one of the first three actions are given, they are applied in the above ordering, regardless of their ordering in ⟨*actions*⟩. Thus, `{stroke,fill}` and `{fill,stroke}` have the same effect.

To shade a path, use the `\pgfshadepath` command, which is explained in Section 55.

47.2 Stroking a Path

When you use \pgfusepath{stroke} to stroke a path, several graphic parameters influence how the path is drawn. The commands for setting these parameters are explained in the following.

Note that all graphic parameters apply to the path as a whole, never only to a part of it.

All graphic parameters are local to the current {pgfscope}, but they persists past TEX groups, *except* for the interior rule (even-odd or nonzero) and the arrow tip kinds. The latter graphic parameters only persist till the end of the current TEX group, but this may change in the future, so do not count on this.

47.2.1 Graphic Parameter: Line Width

\pgfsetlinewidth{⟨*line width*⟩}

This command sets the line width for subsequent strokes (in the current pgfscope). The line width is given as a normal TEX dimension like 0.4pt or 1mm.

```
\begin{pgfpicture}
  \pgfsetlinewidth{1mm}
  \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
  \pgfpathlineto{\pgfpoint{2cm}{0mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{2\pgflinewidth} % double in size
  \pgfpathmoveto{\pgfpoint{0mm}{5mm}}
  \pgfpathlineto{\pgfpoint{2cm}{5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgflinewidth

You can access the current line width via the TEX dimension \pgflinewidth. It will be set to the correct line width, that is, even when a TEX group closed, the value will be correct since it is set globally, but when a {pgfscope} closes, the value is set to the correct value it had before the scope.

47.2.2 Graphic Parameter: Caps and Joins

\pgfsetbuttcap

Sets the line cap to a butt cap. See Section 12.2.1 for an explanation of what this is.

\pgfsetroundcap

Sets the line cap to a round cap. See again Section 12.2.1.

\pgfsetrectcap

Sets the line cap to a square cap. See again Section 12.2.1.

\pgfsetroundjoin

Sets the line join to a round join. See again Section 12.2.1.

\pgfsetbeveljoin

Sets the line join to a bevel join. See again Section 12.2.1.

\pgfsetmiterjoin

Sets the line join to a miter join. See again Section 12.2.1.

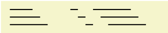
\pgfsetmiterlimit{⟨*miter limit factor*⟩}

Sets the miter limit to ⟨*miter limit factor*⟩. See again Section 12.2.1.

47.2.3 Graphic Parameter: Dashing

\pgfsetdash{⟨*list of even length of dimensions*⟩}{⟨*phase*⟩}

Sets the dashing of a line. The first entry in the list specifies the length of the first solid part of the list. The second entry specifies the length of the following gap. Then comes the length of the second solid part, following by the length of the second gap, and so on. The ⟨*phase*⟩ specifies where the first solid part starts relative to the beginning of the line.



```
\begin{pgfpicture}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0cm}
  \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
  \pgfpathlineto{\pgfpoint{2cm}{0mm}}
  \pgfusepath{stroke}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0.1cm}
  \pgfpathmoveto{\pgfpoint{0mm}{1mm}}
  \pgfpathlineto{\pgfpoint{2cm}{1mm}}
  \pgfusepath{stroke}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0.2cm}
  \pgfpathmoveto{\pgfpoint{0mm}{2mm}}
  \pgfpathlineto{\pgfpoint{2cm}{2mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

Use `\pgfsetdash{}{0pt}` to get a solid dashing.

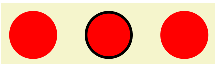
47.2.4 Graphic Parameter: Stroke Color

`\pgfsetstrokecolor{`⟨*color*⟩`}`

Sets the color used for stroking lines to ⟨*color*⟩, where ⟨*color*⟩ is a LaTeX color like `red` or `black!20!red`. Unlike the `\color` command, the effect of this command lasts till the end of the current `{pgfscope}` and not till the end of the current TeX group.

The color used for stroking may be different from the color used for filling. However, a `\color` command will always "immediately override" any special settings for the stroke and fill colors.

In plain TeX, this command will also work, but the problem of *defining* a color arises. After all, plain TeX does not provide LaTeX colors. For this reason, PGF implements a minimalistic "emulation" of the `\definecolor`, `\colorlet`, and `\color` commands. Only gray-scale and rgb colors are supported. For most cases this turns out to be enough.



```
\begin{pgfpicture}
  \pgfsetlinewidth{1pt}
  \color{red}
  \pgfpathcircle{\pgfpoint{0cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \pgfsetstrokecolor{black}
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \color{red}
  \pgfpathcircle{\pgfpoint{2cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
\end{pgfpicture}
```

`\pgfsetcolor{`⟨*color*⟩`}`

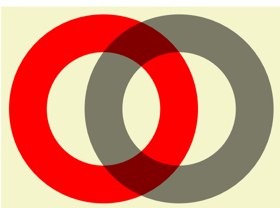
Sets both the stroke and fill color. The difference to the normal `\color` command is that the effect lasts till the end of the current `{pgfscope}`, not only till the end of the current TeX group.

47.2.5 Graphic Parameter: Stroke Opacity

`\pgfsetstrokeopacity{`⟨*value*⟩`}`

Sets the opacity of stroking operations. The ⟨*value*⟩ should be a number between `0` and `1`, where `1` means "fully opaque" and `0` means "fully transparent." A value like `0.5` will cause paths to be stroked in a semitransparent way.

Note: For PostScript output, opacity is rendered correctly only with the most recent versions of GhostScript. Printers and other programs will ignore the opacity setting.



```
\begin{pgfpicture}
  \pgfsetlinewidth{5mm}
  \color{red}
  \pgfpathcircle{\pgfpoint{0cm}{0cm}}{10mm} \pgfusepath{stroke}
  \color{black}
  \pgfsetstrokeopacity{0.5}
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{10mm} \pgfusepath{stroke}
\end{pgfpicture}
```

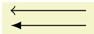
47.2.6 Graphic Parameter: Arrows

After a path has been drawn, PGF can add arrow tips at the ends. Currently, it will only add arrows correctly at the end of paths that consist of a single open part. For other paths, like closed paths or path consisting of multiple parts, the result is not defined.

\pgfsetarrowsstart{⟨*arrow kind*⟩}

Sets the arrow tip kind used at the start of a (possibly curved) path. When this option is used, the line will often be slightly shortened to ensure that the tip of the arrow will exactly "touch" the "real" start of the line.

To "clear" the start arrow, say `\pgfsetarrowsstart{}`.



```
\begin{pgfpicture}
  \pgfsetarrowsstart{latex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
  \pgfsetarrowsstart{to}
  \pgfpathmoveto{\pgfpoint{0cm}{2mm}}
  \pgfpathlineto{\pgfpoint{1cm}{2mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The effect of this command persists only till the end of the current TEX scope.

The different possible arrow kinds are explained in Section 48.

\pgfsetarrowsend{⟨*arrow kind*⟩}

Sets the arrow tip kind used at the end of a path.


```
\begin{pgfpicture}
  \pgfsetarrowsstart{latex}
  \pgfsetarrowsend{to}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetarrows{⟨*start kind*⟩-⟨*end kind*⟩}

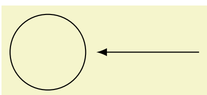
Sets the start arrow kind to ⟨*start kind*⟩ and the end kind to ⟨*end kind*⟩.


```
\begin{pgfpicture}
  \pgfsetarrows{latex-to}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetshortenstart{⟨*dimension*⟩}

This command will shortened the start of every stroked path by the given dimension. This shortening is done in addition to automatic shortening done by a start arrow, but it can be used even if no start arrow is given.

This command is useful if you wish arrows or lines to "stop shortly before" a given point.



```
\begin{pgfpicture}
  \pgfpathcircle{\pgfpointorigin}{5mm}
  \pgfusepath{stroke}
  \pgfsetarrows{latex-}
  \pgfsetshortenstart{4pt}
  \pgfpathmoveto{\pgfpoint{5mm}{0cm}} % would be on the circle
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetshortenend{⟨*dimension*⟩}

Works like `\pgfsetshortenstart`.

47.3 Filling a Path

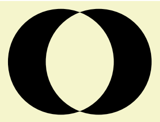
Filling a path means coloring every interior point of the path with the current fill color. It is not always obvious whether a point is "inside" a path when the path is self-intersecting and/or consists or multiple parts. In this case either the nonzero winding number rule or the even-odd crossing number rule is used to decide, which points lie "inside." These rules are explained in Section 12.3.

47.3.1 Graphic Parameter: Interior Rule

You can set which rule is used using the following commands:

`\pgfseteorule`

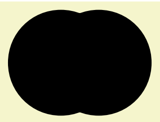
Dictates that the even-odd rule is used in subsequent fillings in the current *TEX scope*. Thus, for once, the effect of this command does not persist past the current TEX scope.



```
\begin{pgfpicture}
  \pgfseteorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

`\pgfsetnonzerorule`

Dictates that the nonzero winding number rule is used in subsequent fillings in the current TEX scope. This is the default.



```
\begin{pgfpicture}
  \pgfsetnonzerorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

47.3.2 Graphic Parameter: Filling Color

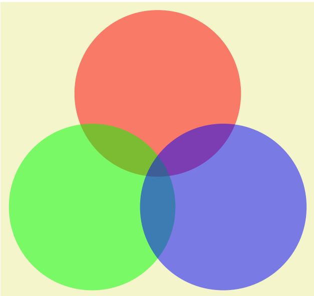
`\pgfsetfillcolor{⟨color⟩}`

Sets the color used for filling paths to ⟨*color*⟩. Like the stroke color, the effect lasts only till the next use of `\color`.

47.3.3 Graphic Parameter: Fill Opacity

`\pgfsetfillopacity{⟨value⟩}`

Sets the opacity of filling operations. As for stroking, the ⟨*value*⟩ should be a number between `0` and `1`. The "filling transparency" will also be used for text and images.



```
\begin{tikzpicture}
  \pgfsetfillopacity{0.5}
  \fill[red]   (90:1cm)  circle (11mm);
  \fill[green] (210:1cm) circle (11mm);
  \fill[blue]  (-30:1cm) circle (11mm);
\end{tikzpicture}
```

47.4 Clipping a Path

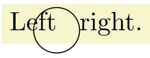
When you add the `clip` option, the current path is used for clipping subsequent drawings. The same rule as for filling is used to decide whether a point is inside or outside the path, that is, either the even-odd rule or the nonzero rule.

Clipping never enlarges the clipping area. Thus, when you clip against a certain path and then clip again against another path, you clip against the intersection of both.

The only way to enlarge the clipping path is to end the {pgfscope} in which the clipping was done. At the end of a {pgfscope} the clipping path that was in force at the beginning of the scope is reinstalled.

47.5 Using a Path as a Bounding Box

When you add the `use as bounding box` option, the bounding box of the picture will be enlarged such that the path in encompassed, but any *subsequent* paths of the current TEX scope will not have any effect on the size of the bounding box. Typically, you use this command at the very beginning of a {pgfpicture} environment.




```
Left
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{use as bounding box} % draws nothing

  \pgfpathcircle{\pgfpointorigin}{2ex}
  \pgfusepath{stroke}
\end{pgfpicture}
right.
```

48 Arrow Tips

48.1 Overview

48.1.1 When Does PGF Draw Arrow Tips?

PGF offers an interface for placing *arrow tips* at the end of lines. The interface works as follows:

1. You (or someone else) assigns a name to a certain kind of arrow tips. For example, the arrow tip `latex` is the arrow tip used by the standard LaTeX picture environment; the arrow tip `to` looks like the tip of the arrow in TeX's `\to` command; and so on.

   This is done once at the beginning of the document.

2. Inside some picture, at some point you specify that in the current scope from now on you would like tips of, say, kind `to` to be added at the end and/or beginning of all paths.

   When an arrow kind has been installed and when PGF is about to stroke a path, the following things happen:

   (a) The beginning and/or end of the path is shortened appropriately.
   (b) The path is stroked.
   (c) The arrow tip is drawn at the beginning and/or end of the path, appropriately rotated and appropriately resized.

In the above description, there are a number of "appropriately." The exact details are not quite trivial and described later on.

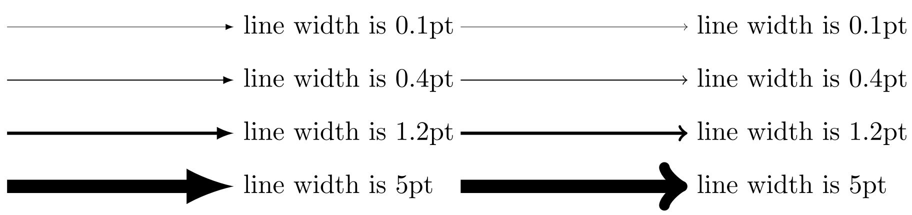
48.1.2 Meta-Arrow Tips

In PGF, arrows are "meta-arrows" in the same way that fonts in TeX are "meta-fonts." When a meta-arrow is resized, it is not simply scaled, but a possibly complicated transformation is applied to the size.

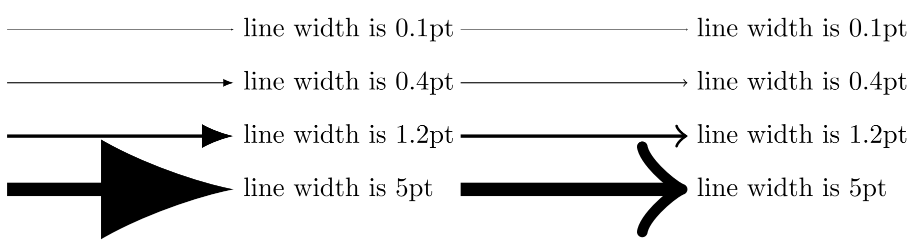
A meta-font is not one particular font at a specific size with a specific stroke width (and with a large number of other parameters being fixed). Rather, it is a "blueprint" (actually, more like a program) for generating such a font at a particular size and width. This allows the designer of a meta-font to make sure that, say, the font is somewhat thicker and wider at very small sizes. To appreciate the difference: Compare the following texts: "Berlin" and "Berlin". The first is a "normal" text, the second is the tiny version scaled by a factor of two. Obviously, the first look better. Now, compare "Berlin" and "Berlin". This time, the normal text was scaled down, while the second text is a "normal" tiny text. The second text is easier to read.

PGF's meta-arrows work in a similar fashion: The shape of an arrow tip can vary according to the line width of the arrow tip is used. Thus, an arrow tip drawn at a line width of 5pt will typically *not* be five times as large as an arrow tip of line width 1pt. Instead, the size of the arrow will get bigger only slowly as the line width increases.

To appreciate the difference, here are the `latex` and `to` arrows, as drawn by PGF at four different sizes:



Here, by comparison, is the same arrow when it is simply "resized" (as done by most programs):



As can be seen, simple scaling produces arrow tips that are way too large at larger sizes and way too small at smaller sizes.

48.2 Declaring an Arrow Tip Kind

To declare an arrow kind "from scratch," the following command is used:

\pgfarrowsdeclare{⟨*start name*⟩}{⟨*end name*⟩}{⟨*extend code*⟩}{⟨*arrow tip code*⟩}

This command declares a new arrow kind. An arrow kind has two names, which will typically be the same. When the arrow tip needs to be drawn, the ⟨*arrow tip code*⟩ will be invoked, but the canvas transformation is setup beforehand to a rotation such that when an arrow tip pointing right is specified, the arrow tip that is actually drawn points in the direction of the line.

Naming the arrow kind. The ⟨*start name*⟩ is the name used for the arrow tip when it is at the start of a path, the ⟨*end name*⟩ is the name used at the end of a path. For example, the arrow kind that looks like a parenthesis has the ⟨*start name*⟩ `(` and the ⟨*end name*⟩ `)` so that you can say `\pgfsetarrows{(-)}` to specify that you want parenthesis arrows and both ends.

The ⟨*end name*⟩ and ⟨*start name*⟩ can be quite arbitrary and may contain spaces.

Basics of the arrow tip code. Let us next have a look at the ⟨*arrow tip code*⟩. This code will be used to draw the arrow tip when PGF thinks this is necessary. The code should draw an arrow that "points right," which means that is should draw an arrow at the end of a line coming from the left and ending at the origin.

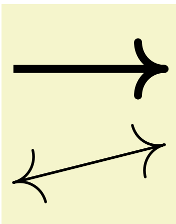
As an example, suppose we wanted to declare an arrow tip consisting of two arcs, that is, we want the arrow tip to look more or less like the red part of the following picture:

```
\begin{tikzpicture}[line width=3pt]
  \draw (-2,0) -- (0,0);
  \draw[red,join=round,cap=round]
        (-10pt,10pt) arc (180:270:10pt) arc (90:180:10pt);
\end{tikzpicture}
```

We could use the following as ⟨*arrow tip code*⟩ for this:

```
\pgfarrowsdeclare{arcs}{arcs}{...}
{
  \pgfsetdash{}{0pt} % do not dash
  \pgfsetroundjoin   % fix join
  \pgfsetroundcap    % fix cap
  \pgfpathmoveto{\pgfpoint{-10pt}{10pt}}
  \pgfpatharc{180}{270}{10pt}
  \pgfpatharc{90}{180}{10pt}
  \pgfusepathqstroke
}
```

Indeed, when the `...` is set appropriately (in a moment), we can write the following:



```
\begin{tikzpicture}
  \draw[-arcs,line width=3pt] (-2,0)  -- (0,0);
  \draw[arcs-arcs,line width=1pt] (-2,-1.5) -- (0,-1);
  \useasboundingbox (-2,-2) rectangle (0,0.75);
\end{tikzpicture}
```

As can be seen in the second example, the arrow tip is automatically rotated as needed when the arrow is drawn. This is achieved by a canvas rotation.

Special considerations about the arrow tip code. There are several things you need to be aware of when designing arrow tip code:

- Inside the code, you may not use the `\pgfusepath` command. The reason is that this command internally calls arrow construction commands, which is something you obviously do not want to happen.

  Instead of `\pgfusepath`, use the quick versions. Typically, you will use `\pgfusepathqstroke`, `\pgfusepathqfill`, or `\pgfusepathqfillstroke`.

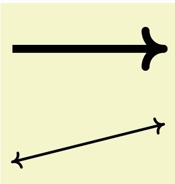
- The code will be executed only once, namely the first time the arrow tip needs to be drawn. The resulting low-level driver commands are protocoled and stored away. In all subsequent uses of the arrow tip, the protocoled code is directly inserted.
- However, the code will be executed anew for each line width. Thus, an arrow of line width 2pt may result in a different protocol than the same arrow for a line width of 0.4pt.
- If you stroke the path that you construct, you should first set the dashing to solid and setup fixed joins and caps, as needed. This will ensure that the arrow tip will always look the same.
- When the arrow tip code is executed, it is automatically put inside a low-level scope, so nothing will "leak out" from the scope.
- The high-level coordinate transformation matrix will be set to the identity matrix when the code is executed for the first time.

Designing meta-arrows. The ⟨*arrow tip code*⟩ should adjust the size of the arrow in accordance with the line width. For a small line width, the arrow tip should be small, for a large line width, it should be larger. However, the size of the arrow typically *should not* grow in direct proportion to the line width. On the other hand, the size of the arrow head typically *should* grow "a bit" with the line width.

For these reasons, PGF will not simply executed your arrow code within a scaled scope, where the scaling depends on the line width. Instead, your ⟨*arrow tip code*⟩ is reexecuted again for each different line width.

In our example, we could use the following code for the new arrow tip kind `arc'` (note the prime):

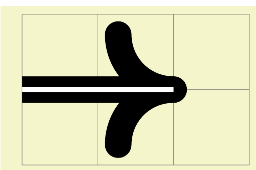
```
\newdimen\arrowsize
\pgfarrowsdeclare{arcs'}{arcs'}{...}
{
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfsetdash{}{0pt} % do not dash
  \pgfsetroundjoin   % fix join
  \pgfsetroundcap    % fix cap
  \pgfpathmoveto{\pgfpoint{-4\arrowsize}{4\arrowsize}}
  \pgfpatharc{180}{270}{4\arrowsize}
  \pgfpatharc{90}{180}{4\arrowsize}
  \pgfusepathqstroke
}
```



```
\begin{tikzpicture}
  \draw[-arcs',line width=3pt] (-2,0)  -- (0,0);
  \draw[arcs'-arcs',line width=1pt] (-2,-1.5) -- (0,-1);
  \useasboundingbox (-2,-1.75) rectangle (0,0.5);
\end{tikzpicture}
```

However, sometimes, it can also be useful to have arrows that do not resize at all when the line width changes. This can be achieved by giving absolute size coordinates in the code, as done for `arc`. On the other hand, you can also have the arrow resize linearly with the line width by specifying all coordinates as multiples of `\pgflinewidth`.

The left and right extend. Let us have another look at the exact left and right "ends" of our arrow tip. Let us draw the arrow tip `arc'` at a very large size:



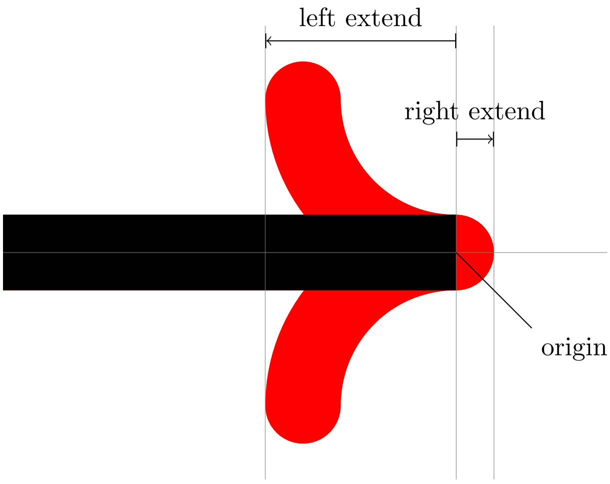
```
\begin{tikzpicture}
  \draw[help lines] (-2,-1) grid (1,1);
  \draw[line width=10pt,-arcs'] (-2,0) -- (0,0);
  \draw[line width=2pt,white] (-2,0) -- (0,0);
\end{tikzpicture}
```

As one can see, the arrow tip does not "touch" the origin as it should, but protrudes a little over the origin. One remedy to this undesirable effect is to change the code of the arrow tip such that everything is shifted half an `\arrowsize` to the left. While this will cause the arrow tip to touch the origin, the line itself will then interfere with the arrow: The arrow tip will be partly "hidden" by the line itself.

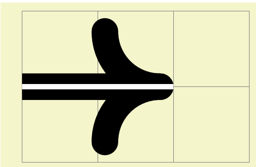
PGF uses a different approach to solving the problem: The ⟨*extend code*⟩ argument can be used to "tell" PGF how much the arrow protrudes over the origin. The argument is also used to tell PGF where the "left" end of the arrow is. However, this number is important only when the arrow is being reversed or composed with other arrow tips.

Once PGF knows the right extend of an arrow kind, it can *shorten* lines by this amount when drawing arrows.

Here is a picture that shows what the visualizes the extends. The arrow tip itself is shown in red once more:



The ⟨*extend code*⟩ is normal TEX code that is executed whenever PGF wants to know how far the arrow tip will protrude to the right and left. The code should call the following two commands: `\pgfarrowsrightextend` and `\pgfarrowsleftextend`. Both arguments take one argument that specifies the size. Here is the final code for the `arc''` arrow tip:



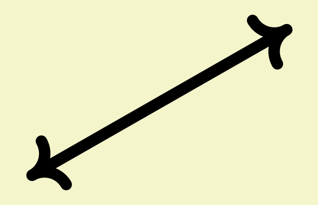
```
\pgfarrowsdeclare{arcs''}{arcs''}
{
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfarrowsleftextend{-4\arrowsize-.5\pgflinewidth}
  \pgfarrowsrightextend{.5\pgflinewidth}
}
{
  \arrowsize=0.2pt
  \advance\arrowsize by .5\pgflinewidth
  \pgfsetdash{}{0pt} % do not dash
  \pgfsetroundjoin   % fix join
  \pgfsetroundcap    % fix cap
  \pgfpathmoveto{\pgfpoint{-4\arrowsize}{4\arrowsize}}
  \pgfpatharc{180}{270}{4\arrowsize}
  \pgfusepathqstroke
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{90}{180}{4\arrowsize}
  \pgfusepathqstroke
}
\begin{tikzpicture}
  \draw[help lines] (-2,-1) grid (1,1);
  \draw[line width=10pt,-arcs''] (-2,0) -- (0,0);
  \draw[line width=2pt,white] (-2,0) -- (0,0);
\end{tikzpicture}
```

48.3 Declaring a Derived Arrow Tip Kind

It is possible to declare arrow kinds in terms of existing ones. For these command to work correctly, the left and right extends must be set correctly.

`\pgfarrowsdeclarealias`{⟨*start name*⟩}{⟨*end name*⟩}{⟨*old start name*⟩}{⟨*old end name*⟩}

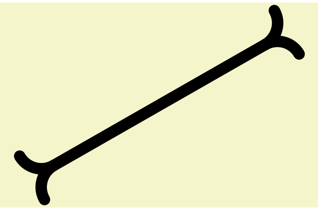
This command can be used to create an alias (another name) for an existing arrow kind.



```
\pgfarrowsdeclarealias{<}{>}{arcs''}{arcs''}%
\begin{tikzpicture}
  \pgfsetarrows{<->}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

`\pgfarrowsdeclarereversed`{⟨*start name*⟩}{⟨*end name*⟩}{⟨*old start name*⟩}{⟨*old end name*⟩}

This command creates a new arrow kind that is the "reverse" of an existing arrow kind. The (automatically cerated) code of the new arrow kind will contain a flip of the canvas and the meanings of the left and right extend will be reversed.

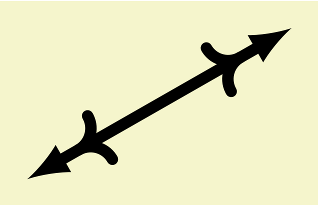


```
\pgfarrowsdeclarereversed{arcs reversed}{arcs reversed}{arcs''}{arcs''}%
\begin{tikzpicture}
  \pgfsetarrows{arcs reversed-arcs reversed}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

`\pgfarrowsdeclarecombine*`[⟨*offset*⟩]{⟨*start name*⟩}{⟨*end name*⟩}{⟨*first start name*⟩}{⟨*first end name*⟩}
{⟨*second start name*⟩}{⟨*second end name*⟩}

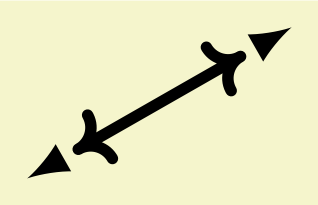
This command creates a new arrow kind that combines two existing arrow kinds. The first arrow kind is the "innermost" arrow kind, the second arrow kind is the "outermost."

The code for the combined arrow kind will install a canvas translation before the innermost arrow kind in drawn. This translation is calculated such that the right tip of the innermost arrow touches the right end of the outermost arrow. The optional ⟨*offset*⟩ can be used to increase (or decrease) the distance between the inner and outermost arrow.



```
\pgfarrowsdeclarecombine[\pgflinewidth]
  {combined}{combined}{arcs''}{arcs''}{latex}{latex}%
\begin{tikzpicture}
  \pgfsetarrows{combined-combined}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

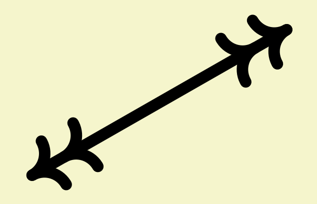
In the star variant, the end of the line is not in the outermost arrow, but inside the innermost arrow.



```
\pgfarrowsdeclarecombine*[\pgflinewidth]
  {combined'}{combined'}{arcs''}{arcs''}{latex}{latex}%
\begin{tikzpicture}
  \pgfsetarrows{combined'-combined'}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

`\pgfarrowsdeclaredouble`[⟨*offset*⟩]{⟨*start name*⟩}{⟨*end name*⟩}{⟨*old start name*⟩}{⟨*old end name*⟩}

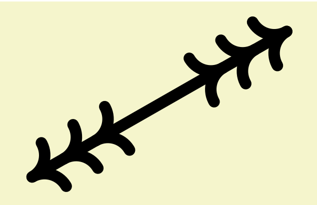
This command is a shortcut for combining an arrow kind with itself.



```
\pgfarrowsdeclaredouble{<<}{>>}{arcs''}{arcs''}%
\begin{tikzpicture}
  \pgfsetarrows{<<->>}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

`\pgfarrowsdeclaretriple[`⟨*offset*⟩`]{`⟨*start name*⟩`}{`⟨*end name*⟩`}{`⟨*old start name*⟩`}{`⟨*old end name*⟩`}`

This command is a shortcut for combining an arrow kind with itself and then again.



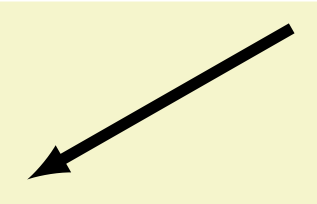
```
\pgfarrowsdeclaretriple{<<<}{>>>}{arcs''}{arcs''}%
\begin{tikzpicture}
  \pgfsetarrows{<<<->>>}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

48.4 Using an Arrow Tip Kind

The following commands install the arrow kind that will be used when stroking is done.

`\pgfsetarrowsstart{`⟨*start arrow kind*⟩`}`

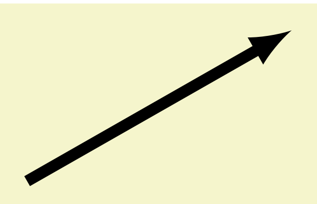
Installs the given ⟨*start arrow kind*⟩ for all subsequent strokes in the in the current TEX-group. If ⟨*start arrow kind*⟩ is empty, no arrow tips will be drawn at the start of the last segment of paths.



```
\begin{tikzpicture}
  \pgfsetarrowsstart{latex}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

`\pgfsetarrowsend{`⟨*start arrow kind*⟩`}`

Like `\pgfsetarrowsstart`, only for the end of the arrow.

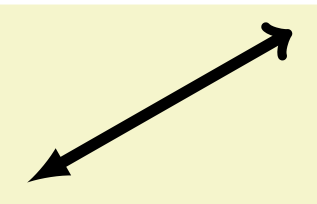


```
\begin{tikzpicture}
  \pgfsetarrowsend{latex}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

Warning: If the compatibility mode is active (which is the default), there also exist old commands called `\pgfsetstartarrow` and `\pgfsetendarrow`, which are incompatible with the meta-arrow management.

`\pgfsetarrows{`⟨*start kind*⟩`-`⟨*end kind*⟩`}`

Calls `\pgfsetarrowsstart` for ⟨*start kind*⟩ and `\pgfsetarrowsend` for ⟨*end kind*⟩.



```
\begin{tikzpicture}
  \pgfsetarrows{latex-to}
  \pgfsetlinewidth{1ex}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3.5cm}{2cm}}
  \pgfusepath{stroke}
  \useasboundingbox (-0.25,-0.25) rectangle (3.75,2.25);
\end{tikzpicture}
```

48.5 Predefined Arrow Tip Kinds

The following arrow tip kinds are always defined:

`stealth-stealth` yields thick and thin

`stealth reversed-stealth reversed` yields thick and thin

`to-to` yields thick and thin

`to reversed-to reversed` yields thick and thin

`latex-latex` yields thick and thin

`latex reversed-latex reversed` yields thick and thin

`|-|` yields thick and thin

For further arrow tips, see page 173.

49 Nodes and Shapes

This section describes the `pgfbaseshapes` package.

```
\usepackage{pgfbaseshapes} % LaTeX
\input pgfbaseshapes.tex   % plain TeX
\usemodule[pgfbaseshapes]  % ConTeXt
```

This package defines commands both for creating nodes and for creating shapes. The package is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

49.1 Overview

PGF comes with a sophisticated set of commands for creating *nodes* and *shapes*. A *node* is a graphical object that consists (typically) of (one or more) text labels and some additional stroked or filled paths. Each node has a certain *shape*, which may be something simple like a `rectangle` or a `circle`, but it may also be something complicated like a `uml class diagram` (this shape is currently not implemented, though). Different nodes that have the same shape may look quite different, however, since shapes (need not) specify whether the shape path is stroked or filled.

49.1.1 Creating and Referencing Nodes

You create a node by calling the macro `\pgfnode` or the more general `\pgfmultipartnode`. These macro takes several parameters and draws the requested shape at a certain position. In addition, it will "remember" the node's position within the current `{pgfpicture}`. You can then, later on, refer to the node's position. Coordinate transformations are "fully supported," which means that if you used coordinate transformations to shift or rotate the shape of a node, the node's position will still be correctly determined by PGF. This is *not* the case if you use canvas transformations, instead.

49.1.2 Anchors

An important property of a node or a shape in general are its *anchors*. Anchors are "important" positions in a shape. For example, the `center` anchor lies at the center of a shape, the `north` anchor is usually "at the top, in the middle" of a shape, the `text` anchor is the lower left corner of the shape's text label (if present), and so on.

Anchors are important both when you create a node and when you reference it. When you create a node, you specify the node's "position" by asking PGF to place the shape in such a way that a certain anchor lies at a certain point. For example, you might ask that the node is placed such that the `north` anchor is at the origin. This will effectively cause the node to be placed below the origin.

When you reference a node, you always reference an anchor of the node. For example, when you request the "`north` anchor of the node just placed" you will get the origin. However, you can also request the "`south` anchor of this node," which will give you a point somewhere below the origin. When a coordinate transformation was in force at the time of creation of a node, all anchors are also transformed accordingly.

49.1.3 Layers of a Shape

The simplest shape, the `coordinate`, has just one anchor, namely the `center`, and a label (which is usually empty). More complicated shapes like the `rectangle` shape also have a *background path*. This is a PGF-path that is defined by the shape. The shape does not prescribe what should happen with the path: When a node is created this path may be stroked (resulting in a frame around the label), filled (resulting in a background color for the text), or just discarded.

Although most shapes consist just of a background path plus some label text, when a shape is drawn, up to seven different layers are drawn:

1. The "behind the background layer." Unlike the background path, which be used in different ways by different nodes, the graphic commands given for this layer will always stroke or always fill the path they construct. They might also insert some text that is "behind everything."

2. The background path layer. How this path is used depends on how the arguments of the `\pgfnode` command.

3. The "before the background path layer." This layer works like the first one, only the commands of this layer are executed after the background path has been used (in whatever way the creator of the node chose).

4. The label layer. This layer inserts the node's text box(es).

5. The "behind the foreground layer." This layer, like the first layer, once more contains graphic commands that are "simply executed."

6. The foreground path layer. This path is treated in the same way as the background path, only it is drawn only after the label text has been drawn.

7. The "before the foreground layer."

Which of these layers are actually used depends on the shape.

49.1.4 Node Parts

A shape typically does not consist only of different background and foreground paths, but it may also have text labels. Indeed, for many shapes the text labels are the more important part of the shape.

Most shapes will have only one text label. In this case, this text label is simply passed as a parameter to the `\pgfnode` command. When the node is drawn, the text label is shifted around such that its lower left corner is at the `text` anchor of the node.

More complicated shapes may have more than one text label. Nodes of such shapes are called *multipart nodes*. The different *node parts* are simply the different text labels. For example, a `uml class` shape might have a `class name` part, a `method` part and an `attributes` part. Indeed, single part nodes are a special case of multipart nodes: They only have one part named `text`.

When a shape is declared, you must specify the node parts. There is a simple command called `\nodeparts` that takes a list of the part names as input. When you create a node of a multipart shape, for each part of the node you must have setup a TEX-box containing the text of the part. For a part named `XYZ` you must setup the box `\pgfnodepartXYZbox`. The box will be placed at the anchor `XYZ`. See the description of `\pgfmultipartnode` for more details.

49.2 Creating Nodes

You create a node using on of the following commands:

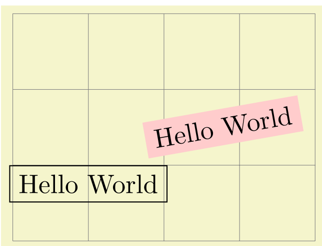
`\pgfnode`{⟨*shape*⟩}{⟨*anchor*⟩}{⟨*label text*⟩}{⟨*name*⟩}{⟨*path usage command*⟩}

This command creates a new node. The ⟨*shape*⟩ of the node must have been declared previously using `\pgfdeclareshape`.

The shape is shifted such that the ⟨*anchor*⟩ is at the origin. In order to place the shape somewhere else, use the coordinate transformation prior to calling this command.

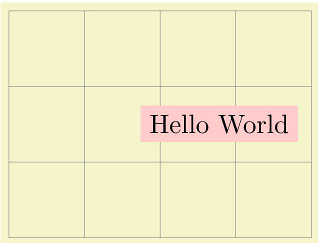
The ⟨*name*⟩ is a name for later reference. If no name is given, nothing will be "saved" for the node, it will just be drawn.

The ⟨*path usage command*⟩ is executed for the background and the foreground path (if the shape defines them).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \pgftransformshift{\pgfpoint{1cm}{1cm}}
    \pgfnode{rectangle}{north}{Hello World}{hellonode}{\pgfusepath{stroke}}
  }
  {
    \color{red!20}
    \pgftransformrotate{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

As can be seen, all coordinate transformations are also applied to the text of the shape. Sometimes, it is desirable that the transformations are applied to the point where the shape will be anchored, but you do not wish the shape itself to the transformed. In this case, you should call `\pgftransformresetnontranslations` prior to calling the `\pgfnode` command.



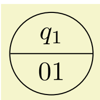

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \color{red!20}
    \pgftransformrotate{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgftransformresetnontranslations
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

The ⟨*label text*⟩ is typeset inside the TeX-box `\pgfnodeparttextbox`. This box is shown at the `text` anchor of the node, if the node has a `text` part. See the description of `\pgfmultipartnode` for details.

`\pgfmultipartnode{`⟨*shape*⟩`}{`⟨*anchor*⟩`}{`⟨*name*⟩`}{`⟨*path usage command*⟩`}`

This command is the more general (and less user-friendly) version of the `\pgfnode` command. While the `\pgfnode` command can only be used for shapes that have a single part (which is the case for most shapes), this command can also be used with multi-part nodes.

When this command is called, for each node part of the node you must have setup one TeX-box. Suppose the shape has two parts: The `text` part and the `lower` part. Then, prior to calling `\pgfmultipartnode`, you must have setup the boxes `\pgfnodeparttextbox` and `\pgfnodepartlowerbox`. These boxes may contain any TeX-text. The shape code will then compute the positions of the shape's anchors based on the sizes of the these shapes. Finally, when the node is drawn, the boxes are placed at the anchor positions `text` and `lower`.




```
\setbox\pgfnodeparttextbox=\hbox{$q_1$}
\setbox\pgfnodepartlowerbox=\hbox{01}
\begin{pgfpicture}
  \pgfmultipartnode{circle split}{center}{my state}{\pgfusepath{stroke}}
\end{pgfpicture}
```

Note: Be careful when using the `\setbox` command inside a `{pgfpicture}` command. You will have to use `\pgfinterruptpath` at the beginning of the box and `\endpgfinterruptpath` at the end of the box to make sure that the box is typeset correctly. In the above example this problem was sidestepped by moving the box construction outside the environment.

Note: It is not necessary to use `\newbox` for every node part name. Although you need a different box for each part of a single shape, two different shapes may very well use the same box even when the names of the parts are different. Suppose you have a `circle split` shape that has an `lower` part and you have a `uml class` shape that has a `methods` part. Then, in order to avoid exhausting TeX's limited number of box registers, you can say

```
\newbox\pgfnodepartlowerbox
\let\pgfnodepartmethodsbox=\pgfnodepartlowerbox
```

Also, when you have a node part name with spaces like `class name`, it may be useful to create an alias:

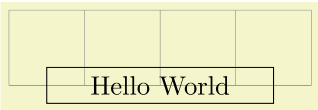
```
\newbox\mybox
\expandafter\let\csname pgfnodepartclass namebox\endcsname=\mybox
```

There are a number of values that have an influence on the size of a node. These parameters can be changed using the following commands:

\pgfsetshapeminwidth{⟨*dimension*⟩}

This command sets the macro \pgfshapeminwidth to ⟨*dimension*⟩. This dimension is the *recommended* minimum width of a shape. Thus, when a shape is drawn and when the shape's width would be smaller than ⟨*dimension*⟩, the shape's width is enlarged by adding some empty space.

Note that this value is just a recommendation. A shape may choose to ignore the value of `\pgfshapeminwidth`.



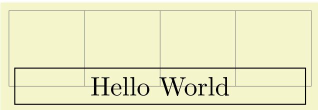

```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfsetshapeminwidth{3cm}
  \pgfnode{rectangle}{center}{Hello World}{}{\pgfusepath{stroke}}
\end{tikzpicture}
```

\pgfsetshapeminheight{⟨*dimension*⟩}

Works like `\pgfsetshapeminwidth`.

\pgfsetshapeinnerxsep{⟨*dimension*⟩}

This command sets the macro \pgfshapeinnerxsep to ⟨*dimension*⟩. This dimension is the *recommended* horizontal inner separation between the label text and the background path. As before, this value is just a recommendation and a shape may choose to ignore the value of `\pgfshapeinnerxsep`.




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfsetshapeinnerxsep{1cm}
  \pgfnode{rectangle}{center}{Hello World}{}{\pgfusepath{stroke}}
\end{tikzpicture}
```

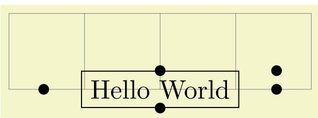
\pgfsetshapeinnerysep{⟨*dimension*⟩}

Works like `\pgfsetshapeinnerysep`.

\pgfsetshapeouterxsep{⟨*dimension*⟩}

This command sets the macro \pgfshapeouterxsep to ⟨*dimension*⟩. This dimension is the recommended horizontal outer separation between the background path and the "outer anchors." For example, if ⟨*dimension*⟩ is `1cm` then the `east` anchor will be 1cm to the right of the right border of the background path.

As before, this value is just a recommendation.




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfsetshapeouterxsep{.5cm}
  \pgfnode{rectangle}{center}{Hello World}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfsetshapeouterysep{⟨*dimension*⟩}

Works like `\pgfsetshapeouterysep`.

\pgfsetshapeaspect{⟨*value*⟩}

This command sets the macro \pgfshapeaspect to ⟨*value*⟩. Furthermore, \pgfshapeaspectinverse is set to the reciprocal of ⟨*value*⟩. The aspect is a recommendation for the quotient of the width and the height of a shape.

49.3 Using Anchors

Each shape defines a set of anchors. We saw already that the anchors are used when the shape is drawn: the shape is placed in such a way that the given anchor is at the origin (which in turn is typically translated somewhere else).

One has to look up the set of anchors of each shape, there is no "default" set of anchors, except for the `center` anchor, which should always be present. Also, most shapes will declare anchors like `north` or `east`, but this is not guaranteed.

49.3.1 Referencing Anchors of Nodes in the Same Picture

Once a node has been defined, you can refer to its anchors using the following commands:

\pgfpointanchor{⟨*node*⟩}{⟨*anchor*⟩}

This command is another "point command" like the commands described in Section 44. It returns the coordinate of the given ⟨*anchor*⟩ in the given ⟨*node*⟩. The command can be used in commands like `\pgfpathmoveto`.


```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

In the above example, you may have noticed something curious: The rotation transformation is still in force when the anchors are invoked, but it does not seem to have an effect. You might expect that the rotation should apply to the already rotated points once more.

However, `\pgfpointanchor` returns a point that takes the current transformation matrix into account: *The inverse transformation to the current coordinate transformation is applied to an anchor point before returning it.*

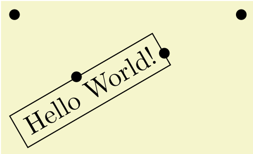
This behavior may seem a bit strange, but you will find it very natural in most cases. If you really want to apply a transformation to an anchor point (for example, to "shift it away" a little bit), you have to invoke `\pgfpointanchor` without any transformations in force. Here is an example:


```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

  {
    \pgftransformreset
    \pgfpointanchor{x}{east}
    \xdef\mycoordinate{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}
  }

  \pgfpathcircle{\mycoordinate}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

A special situation arises when the ⟨*node*⟩ lies in a picture different from the current picture. In this case, if you have not told PGF that the picture should be "remembered," the ⟨*node*⟩ will be treated as if it lied in the current picture. For example, if the ⟨*node*⟩ was at position $(3, 2)$ in the original picture, it is treated as if it lied at position $(3, 2)$ in the current picture. However, if you have told PGF to remember the picture position of the node's picture and also of the current picture, then `\pgfpointanchor` will return a coordinate that corresponds to the position of the node's anchor on the page, transformed into the current coordinate system. For examples and more details see Section 49.3.2.

\pgfpointshapeborder{⟨*node*⟩}{⟨*point*⟩}

This command returns the point on the border of the shape that lies on a straight line from the center of the node to ⟨*point*⟩. For complex shapes it is not guaranteed that this point will actually lie on the border, it may be on the border of a "simplified" version of the shape.




```
\begin{pgfpicture}
  \begin{pgfscope}
    \pgftransformrotate{30}
    \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
  \end{pgfscope}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{2cm}{1cm}}}{2pt}
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}{1cm}}}{2pt}
  \pgfpathcircle{\pgfpoint{-1cm}{1cm}}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

49.3.2 Referencing Anchors of Nodes in Different Pictures

As a picture is typeset, PGF keeps track of the positions of all nodes inside the picture. What PGF does not remember is the position of the picture *itself* on the page. Thus, if you define a node in one picture and then try to reference this node while another picture is typeset, PGF will only know the position of the nodes that you try to typeset inside the original picture, but it will not know where this picture lies. What is missing is the relative positioning of the two pictures.

To overcome this problem, you need to tell PGF that it should remember the position of pictures on a page. If these positions are remembered, then PGF can compute the offset between the pictures and make nodes in different pictures accessible.

Determining the positions of pictures on the page is, alas, not-so-easy. Because of this, PGF does not do so automatically. Rather, you have to proceed as follows:

1. You have to use a backend driver that supports position tracking. pdfTEX is one such drivers, `dvips` currently is not.
2. You have to say `\pgfrememberpicturepositiononpagetrue` somewhere before or inside every picture
   - in which you wish to reference a node and
   - from which you wish to reference a node in another picture.

   The second item is important since PGF does not only need to know the position of the picture in which the node you wish to reference lies, but it also needs to know where the current picture lies.
3. You typically have to run TEX twice (depending on the backend driver) since the position information typically gets written into an external file on the first run and is available only on the second run.
4. You have to switch off automatic bounding bound computations. The reason is that the node in the other picture should not influence the size of the bouding box of the current picture. You should say `\pgfusepath{use as bounding box}` before using a coordinate in another picture.

49.4 Predefined Nodes

There are several special nodes that are always defined and which you should not attempt to redefine.

Predefined node `current bounding box`

This node is of shape `rectangle`. Unlike normal nodes, its size changes constantly and always reflects the size of the bounding box of the current picture. This means that, for instance, that

```
\pgfpointanchor{current bounding box}{south east}
```

returns the lower left corner of the bounding box of the current picture.

Predefined node current path bounding box

This node is also of shape rectangle. Its size is the size of the bounding box of the current path.

Predefined node current page

This node is inside a virtual remembered picture. The size of this node is the size of the current page. This means that if you create a remembered picture and inside this picture you reference an anchor of this node, you reference an absolute position on the page. To demonstrate the effect, the following code puts some text in the lower left corner of the current page. Note that this works only if the backend driver supports it, otherwise the text is inserted right here.

```
\pgfrememberpicturepositiononpagetrue
\begin{pgfpicture}
  \pgfusepath{use as bounding box}
  \pgftransformshift{\pgfpointanchor{current page}{south west}}
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \pgftext[left,base]{
    \textcolor{red}{
      Text absolutely positioned in
      the lower left corner.}
  }
\end{pgfpicture}
```

49.5 Declaring New Shapes

Defining a shape is, unfortunately, a not-quite-trivial process. The reason is that shapes need to be both very flexible (their size will vary greatly according to circumstances) and they need to be constructed reasonably "fast." PGF must be able to handle pictures with several hundreds of nodes and documents with thousands of nodes in total. It would not do if PGF had to compute and store, say, dozens of anchor positions for every node.

49.5.1 What Must Be Defined For a Shape?

In order to define a new shape, you must provide:

- a *shape name*,
- code for computing the *saved anchors* and *saved dimensions*,
- code for computing *anchor* positions in terms of the saved anchors,
- optionally code for the *background path* and *foreground path*,
- optionally code for *things to be drawn before or behind* the background and foreground paths.
- optionally a list of node parts.

49.5.2 Normal Anchors Versus Saved Anchors

Anchors are special places in shape. For example, the north east anchor, which is a normal anchor, lies at the upper right corner of the rectangle shape, as does \northeast, which is a saved anchor. The difference is the following: *saved anchors are computed and stored for each node, anchors are only computed as needed.* The user only has access to the normal anchors, but a normal anchor can just "copy" or "pass through" the location of a saved anchor.

The idea behind all this is that a shape can declare a very large number of normal anchors, but when a node of this shape is created, these anchors are not actually computed. However, this causes a problem: When we wish to reference an anchor of a node at some later time, we must still able to compute the position of the anchor. For this, we may need a lot of information: What was the transformation matrix that was in force when the node was created? What was the size of the text box? What were the values of the different separation dimensions? And so on.

To solve this problem, PGF will always compute the locations of all *saved anchors* and store these positions. Then, when an normal anchor position is requested later on, the anchor position can be given just from knowing where the locations of the saved anchors.

Text absolutely positioned in the lower left corner.

As an example, consider the `rectangle` shape. For this shape two anchors are saved: The `\northeast` corner and the `\southwest` corner. A normal anchor like `north west` can now easily be expressed in terms of these coordinates: Take the x-position of the `\southwest` point and the y-position of the `\northeast` point. The `rectangle` shape currently defines 13 normal anchors, but needs only two saved anchors. Adding new anchors like a `south south east` anchor would not increase the memory and computation requirements of pictures.

All anchors (both saved and normal) are specified in a local *shape coordinate space*. This is also true for the background and foreground paths. The `\pgfnode` macro will automatically apply appropriate transformations to the coordinates so that the shape is shifted to the right anchor or otherwise transformed.

49.5.3 Command for Declaring New Shapes

The following command declares a new shape:

`\pgfdeclareshape{`⟨*shape name*⟩`}{`⟨*shape specification*⟩`}`

This command declares a new shape named ⟨*shape name*⟩. The shape name can later be used in commands like `\pgfnode`.

The ⟨*shape specification*⟩ is some TeX code containing calls to special commands that are only defined inside the ⟨*shape specification*⟩ (similarly to commands like `\draw` that are only available inside the `{tikzpicture}` environment).

Example: Here is the code of the `coordinate` shape:

```
\pgfdeclareshape{coordinate}
{
  \savedanchor\centerpoint{%
    \pgf@x=.5\wd\pgfnodeparttextbox%
    \pgf@y=.5\ht\pgfnodeparttextbox%
    \advance\pgf@y by -.5\dp\pgfnodeparttextbox%
  }
  \anchor{center}{\centerpoint}
  \anchorborder{\centerpoint}
}
```

The special commands are explained next. In the examples given for the special commands a new shape will be constructed, which we might call `simple rectangle`. It should behave like the normal rectangle shape, only without bothering about the fine details like inner and outer separations. The skeleton for the shape is the following.

```
\pgfdeclareshape{simple rectangle}{
  ...
}
```

`\nodeparts{`⟨*list of node parts*⟩`}`

This command declares which parts make up nodes of this shape. A *node part* is a (possibly empty) text label that is drawn when a node of the shape is created.

By default, a shape has just one node part called `text`. However, there can be several node parts. For example, the `circle split` shape has two parts: the `text` part, which shows that upper text, and a `lower` part, which shows the lower text. For the `circle split` shape the `\nodeparts` command was called with the argument `{text,lower}`.

When a multipart node is created, the text labels are drawn in the sequences listed in the ⟨*list of node parts*⟩. For each node part there you must have declared one anchor and the TeX-box of the part is placed at this anchor. For a node part called `XYZ` the TeX-box `\pgfnodepartXYZbox` is placed at anchor `XYZ`.

`\savedanchor{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command declares a saved anchor. The argument ⟨*command*⟩ should be a TeX macro name like `\centerpoint`.

The ⟨*code*⟩ will be executed each time `\pgfnode` (or `\pgfmultipartnode`) is called to create a node of the shape ⟨*shape name*⟩. When the ⟨*code*⟩ is executed, the TeX-boxes of the node parts will contain the text labels of the node. Possibly, these box are void. For example, if there is just a `text` part, the node `\pgfnodeparttextbox` will be setup when the ⟨*code*⟩ is executed.

The ⟨*code*⟩ can use the width, height, and depth of the box(es) to compute the location of the saved anchor. In addition, the ⟨*code*⟩ can take into account the values of dimensions like `\pgfshapeminwidth` or `\pgfshapeinnerxsep`. Furthermore, the ⟨*code*⟩ can take into consideration the values of any further shape-specific variables that are set at the moment when `\pgfnode` is called.

The net effect of the ⟨*code*⟩ should be to set the two TeX dimensions `\pgf@x` and `\pgf@y`. One way to achieve this is to say `\pgfpoint{`⟨*x value*⟩`}{`⟨*y value*⟩`}` at the end of the ⟨*code*⟩, but you can also just set these variables. The values that `\pgf@x` and `\pgf@y` have after the code has been executed, let us call them x and y, will be recorded and stored together with the node that is created by the command `\pgfnode`.

The macro ⟨*command*⟩ is defined to be `\pgfpoint{`x`}{`y`}`. However, the ⟨*command*⟩ is only locally defined while anchor positions are being computed. Thus, it is possible to use very simple names for ⟨*command*⟩, like `\center` or `\a`, without causing a name-clash. (To be precise, very simple ⟨*command*⟩ names will clash with existing names, but only locally inside the computation of anchor positions; and we do not need the normal `\center` command during these computations.)

For our `simple rectangle` shape, we will need only one saved anchor: The upper right corner. The lower left corner could either be the origin or the "mirrored" upper right corner, depending on whether we want the text label to have its lower left corner at the origin or whether the text label should be centered on the origin. Either will be fine, for the final shape this will make no difference since the shape will be shifted anyway. So, let us assume that the text label is centered on the origin (this will be specified later on using the `text` anchor). We get the following code for the upper right corner:

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.5\ht\pgfnodeparttextbox % height of the box, ignoring the depth
  \pgf@x=.5\wd\pgfnodeparttextbox % width of the box
}
```

If we wanted to take, say, the `\pgfshapeminwidth` into account, we could use the following code:

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.\ht\pgfnodeparttextbox % height of the box
  \pgf@x=.\wd\pgfnodeparttextbox % width of the box
  \setlength{\pgf@xa}{\pgfshapeminwidth}
  \ifdim\pgf@x<.5\pgf@xa
    \pgf@x=.5\pgf@xa
  \fi
}
```

Note that we could not have written `.5\pgfshapeminwidth` since the minium width is stored in a "plain text macro," not as a real dimension. So if `\pgfshapeminwidth` depth were 2cm, writing `.5\pgfshapeminwidth` would yield the same as `.52cm`.

In the "real" `rectangle` shape the code is somewhat more complex, but you get the basic idea.

`\saveddimen{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command is similar to `\savedanchor`, only instead of setting ⟨*command*⟩ to `\pgfpoint{`x`}{`y`}`, the ⟨*command*⟩ is set just to (the value of) x.

In the `simple rectangle` shape we might use a saved dimension to store the depth of the shape box.

```
\shapedimen{\depth}{
  \pgf@x=\dp\pgfnodeparttextbox
}
```

`\anchor{`⟨*name*⟩`}{`⟨*code*⟩`}`

This command declares an anchor named ⟨*name*⟩. Unlike for saved anchors, the ⟨*code*⟩ will not be executed each time a node is declared. Rather, the ⟨*code*⟩ is only executed when the anchor is specifically requested; either for anchoring the node during its creation or as a position in the shape referenced later on.

The ⟨*name*⟩ is a quite arbitrary string that is not "passed down" to the system level. Thus, names like `south` or `1` or `::` would all be fine.

A saved anchor is not automatically also a normal anchor. If you wish to give the users access to a saved anchor you must declare a normal anchor that just returns the position of the saved anchor.

When the ⟨*code*⟩ is executed, all saved anchor macros will be defined. Thus, you can reference them in your ⟨*code*⟩. The effect of the ⟨*code*⟩ should be to set the values of `\pgf@x` and `\pgf@y` to the coordinates of the anchor.

Let us consider some example for the `simple rectangle` shape. First, we would like to make the upper right corner publicly available, for example as `north east`:

```
\anchor{north east}{\upperrightcorner}
```

The `\upperrightcorner` macro will set `\pgf@x` and `\pgf@y` to the coordinates of the upper right corner. Thus, `\pgf@x` and `\pgf@y` will have exactly the right values at the end of the anchor's code.

Next, let us define a `north west` anchor. For this anchor, we can negate the `\pgf@x` variable:

```
\anchor{north west}{
  \upperrightcorner
  \pgf@x=-\pgf@x
}
```

Finally, it is a good idea to always define a `center` anchor, which will be the default location for a shape.

```
\anchor{center}{\pgfpointorigin}
```

You might wonder whether we should not take into consideration that the node is not placed at the origin, but has been shifted somewhere. However, the anchor positions are always specified in the shape's "private" coordinate system. The "outer" transformation that has been applied to the shape upon its creation is applied automatically to the coordinates returned by the anchor's ⟨*code*⟩.

Out `simple rectangle` only has one text label (node part) called `text`. This is the default situation, so we need not do anything. For the `text` node part we must setup a `text` anchor. This anchor is used upon creation of a node to determine the lower left corner of the text label (within the private coordinate system of the shape). By default, the `text` anchor is at the origin, but you may change this. For example, we would say

```
\anchor{text}{%
  \upperrightcorner%
  \pgf@x=-\pgf@x%
  \pgf@y=-\pgf@y%
}
```

to center the text label on the origin in the shape coordinate space. Note that we could *not* have written the following:

```
\anchor{text}{\pgfpoint{-.5\wd\pgfnodeparttextbox}{-.5\ht\pgfnodeparttextbox}}
```

Do you see why this is wrong? The problem is that the box `\pgfnodeparttextbox` will most likely not have the correct size when the anchor is computed. After all, the anchor position might be recomputed at a time when several other nodes have been created.

If a shape has several node parts, we would have to define an anchor for each part.

`\anchorborder{`⟨*code*⟩`}`

A *border anchor* is an anchor point on the border of the shape. What exactly is considered as the "border" of the shape depends on the shape.

When the user request a point on the border of the shape using the `\pgfpointshapeborder` command, the ⟨*code*⟩ will be executed to discern this point. When the execution of the ⟨*code*⟩ starts, the dimensions `\pgf@x` and `\pgf@y` will have been set to a location p in the shape's coordinate system. It is now the job of the ⟨*code*⟩ to setup `\pgf@x` and `\pgf@y` such that they specify the point on the shape's border that lies on a straight line from the shape's center to the point p. Usually, this is a somewhat complicated computation, involving many case distinctions and some basic math.

For our `simple rectangle` we must compute a point on the border of a rectangle whose one corner is the origin (ignoring the depth for simplicity) and whose other corner is `\upperrightcorner`. The following code might be used:

```
\anchorborder{%
  % Call a function that computes a border point. Since this
  % function will modify dimensions like \pgf@x, we must move them to
  % other dimensions.
  \@tempdima=\pgf@x
  \@tempdimb=\pgf@y
  \pgfpointborderrectangle{\pgfpoint{\@tempdima}{\@tempdimb}}{\upperrightcorner}
}
```

`\backgroundpath{⟨code⟩}`

This command specifies the path that "makes up" the background of the shape. Note that the shape cannot prescribe what is going to happen with the path: It might be drawn, shaded, filled, or even thrown away. If you want to specify that something should "always" happen when this shape is drawn (for example, if the shape is a stop-sign, we *always* want it to be filled with a red color), you can use commands like `\beforebackgroundpath`, explained below.

When the ⟨*code*⟩ is executed, all saved anchors will be in effect. The ⟨*code*⟩ should contain path construction commands.

For our `simple rectangle`, the following code might be used:

```
\backgroundpath{
  \pgfpathrectanglecorners
    {\upperrightcorner}
    {\pgfpointscale{-1}{\upperrightcorner}}
}
```

As the name suggests, the background path is used "behind" the text labels. Thus, this path is used first, then the text labels are drawn, possibly obscuring part of the path.

`\foregroundpath{⟨code⟩}`

This command works like `\backgroundpath`, only it is invoked after the text labels have been drawn. This means that this path can possibly obscure (part of) the text labels.

`\behindbackgroundpath{⟨code⟩}`

Unlike the previous two commands, ⟨*code*⟩ should not only construct a path, it should also use this path in whatever way is appropriate. For example, the ⟨*code*⟩ might fill some area with a uniform color.

Whatever the ⟨*code*⟩ does, it does it first. This means that any drawing done by ⟨*code*⟩ will be even behind the background path.

Note that the ⟨*code*⟩ is protected with a `{pgfscope}`.

`\beforebackgroundpath{⟨code⟩}`

This command works like `\behindbackgroundpath`, only the ⟨*code*⟩ is executed after the background path has been used, but before the texts label are drawn.

`\behindforegroundpath{⟨code⟩}`

The ⟨*code*⟩ is executed after the text labels have been drawn, but before the foreground path is used.

`\beforeforegroundpath{⟨code⟩}`

This ⟨*code*⟩ is executed at the very end.

`\inheritsavedanchors[from={⟨another shape name⟩}]`

This command allows you to inherit the code for saved anchors from ⟨*another shape name*⟩. The idea is that if you wish to create a new shape that is just a small modification of a another shape, you can recycle the code used for ⟨*another shape name*⟩.

The effect of this command is the same as if you had called `\savedanchor` and `\saveddimen` for each saved anchor or saved dimension declared in ⟨*another shape name*⟩. Thus, it is not possible to "selectively" inherit only some saved anchors, you always have to inherit all saved anchors from another shape. However, you can inherit the saved anchors of more than one shape by calling this command several times.

`\inheritbehindbackgroundpath[from={`⟨*another shape name*⟩`}]`

This command can be used to inherit the code used for the drawings behind the background path from ⟨*another shape name*⟩.

`\inheritbackgroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the background path code from ⟨*another shape name*⟩.

`\inheritbeforebackgroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the before background path code from ⟨*another shape name*⟩.

`\inheritbehindforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the behind foreground path code from ⟨*another shape name*⟩.

`\inheritforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the foreground path code from ⟨*another shape name*⟩.

`\inheritbeforeforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the before foreground path code from ⟨*another shape name*⟩.

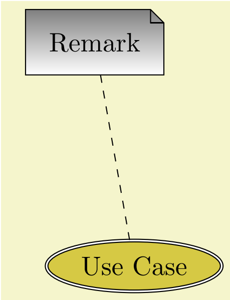
`\inheritanchor[from={`⟨*another shape name*⟩`}]{`⟨*name*⟩`}`

Inherits the code of one specific anchor named ⟨*name*⟩ from ⟨*another shape name*⟩. Thus, unlike saved anchors, which must be inherited collectively, normal anchors can and must be inherited individually.

`\inheritanchorborder[from={`⟨*another shape name*⟩`}]`

Inherits the border anchor code from ⟨*another shape name*⟩.

The following example shows how a shape can be defined that relies heavily on inheritance:




```
\pgfdeclareshape{document}{
  \inheritsavedanchors[from=rectangle] % this is nearly a rectangle
  \inheritanchorborder[from=rectangle]
  \inheritanchor[from=rectangle]{center}
  \inheritanchor[from=rectangle]{north}
  \inheritanchor[from=rectangle]{south}
  \inheritanchor[from=rectangle]{west}
  \inheritanchor[from=rectangle]{east}
  % ... and possibly more
  \backgroundpath{% this is new
    % store lower right in xa/ya and upper right in xb/yb
    \southwest \pgf@xa=\pgf@x \pgf@ya=\pgf@y
    \northeast \pgf@xb=\pgf@x \pgf@yb=\pgf@y
    % compute corner of ''flipped page''
    \pgf@xc=\pgf@xb \advance\pgf@xc by-5pt % this should be a parameter
    \pgf@yc=\pgf@yb \advance\pgf@yc by-5pt
    % construct main path
    \pgfpathmoveto{\pgfpoint{\pgf@xa}{\pgf@ya}}
    \pgfpathlineto{\pgfpoint{\pgf@xa}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@ya}}
    \pgfpathclose
    % add little corner
    \pgfpathmoveto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
 }
}\hskip-1.2cm
\begin{tikzpicture}
  \node[shade,draw,shape=document,inner sep=2ex] (x) {Remark};
  \node[fill=examplefill,draw,ellipse,double]
    at ([shift=(-80:3cm)]x) (y) {Use Case};

  \draw[dashed] (x) -- (y);
\end{tikzpicture}
```

49.6 Predefined Shapes

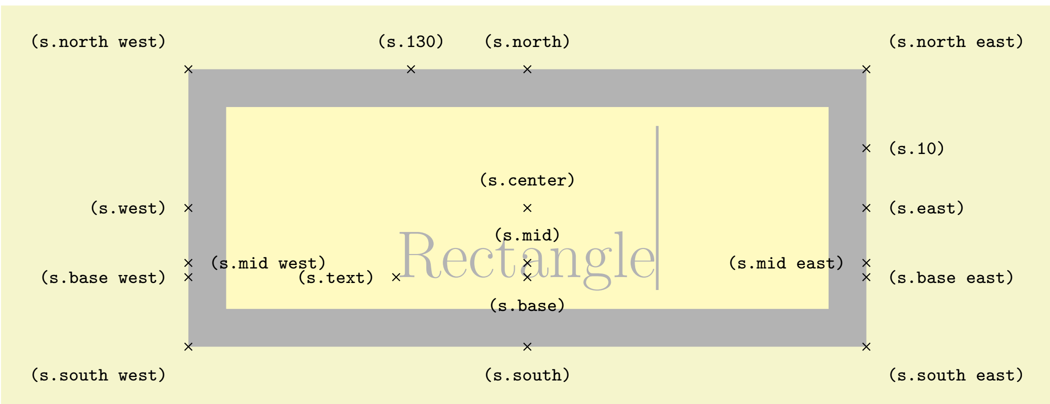
Shape coordinate

The `coordinate` is mainly intended to be used to store locations using the node mechanism. This shape does not have any background path and options like `draw` have no effect on it. Also, it does not have any node parts, so no text is drawn when this shape is used.

Ti*k*Z handles this shape in a special way, see Section 13.13.

Shape rectangle

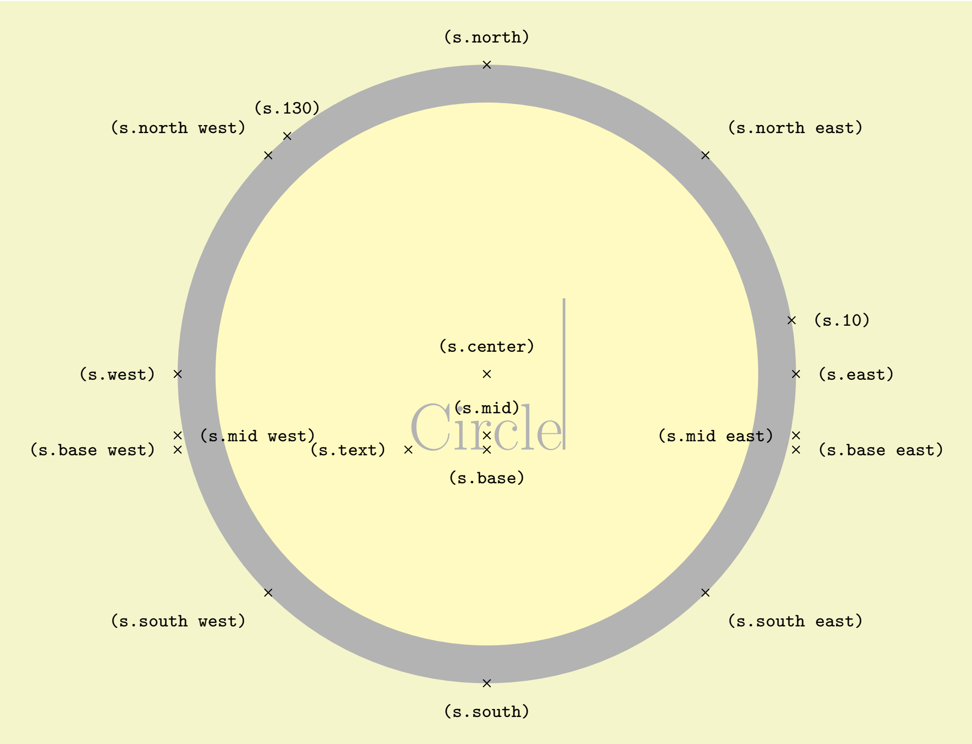
This shape is a rectangle tightly fitting the text box. Use inner or outer separation to increase the distance between the text box and the border and the anchors. The following figure shows the anchors defined by this shape; the anchors `10` and `130` are example of border anchors.



```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rectangle,style=shape example] {Rectangle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates {(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `circle`

This shape is a circle tightly fitting the text box.




```
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle,style=shape example] {Circle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates {(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

50 Matrices

50.1 Overview

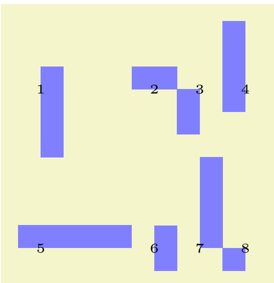
Matrices are a mechanism for aligning several so-called cell pictures horizontally and vertically. The resulting alignment is placed in a normal node and the command for creating matrices, `\pgfmatrix`, takes options very similar to the `\pgfnode` command.

In the following, the basic idea behind the alignment mechanism is explained first. Then the command `\pgfmatrix` is explained. At the end of the section additional ways of modifying the width of columns and rows is discussed.

50.2 Cell Pictures and Their Alignment

A matrix consists of rows of *cells*. Cells are separated using the special command `\pgfmatrixnextcell`, rows are ended using the command `\\`. Each cell contains a *cell picture*, although cell pictures are not complete pictures as they lack layers. However, each cell picture has its own bouding box like a normal picture does. These bounding boxes are important for the alignment as explained in the following.

Each cell picture will have an origin somewhere in the picture (or even outside the picture). The position of these origins is important for the alignment: On each row the origins will be on the same horizontal line and for each column the origins will also be on the same vertical line. These two requirements mean that the cell pictures may need to be shifted around so that the origins wind up on the same lines. The top of a row is given by the top of the cell picture whose bounding box's maximum y-position is largest. Similarly, the bottom of a row is given by the bottom of the cell picture whose bounding box's minimum y-position is the most negative. Similarly, the left end of a row is given by the left end of the cell whose bounding box's x-position is the most negative; and similarly for the right end of a row.




```
\begin{tikzpicture}[x=3mm,y=3mm,fill=blue!50]
  \def\atorig#1{\node[black] at (0,0) {\tiny #1};}

  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{}
    {
      \fill (0,-3)  rectangle (1,1);\atorig1 \pgfmatrixnextcell
      \fill (-1,0)  rectangle (1,1);\atorig2 \pgfmatrixnextcell
      \fill (-1,-2) rectangle (0,0);\atorig3 \pgfmatrixnextcell
      \fill (-1,-1) rectangle (0,3);\atorig4 \\
      \fill (-1,0)  rectangle (4,1);\atorig5 \pgfmatrixnextcell
      \fill (0,-1)  rectangle (1,1);\atorig6 \pgfmatrixnextcell
      \fill (0,0)   rectangle (1,4);\atorig7 \pgfmatrixnextcell
      \fill (-1,-1) rectangle (0,0);\atorig8 \\
    }
\end{tikzpicture}
```

50.3 The Matrix Command

All matrices are typeset using the following command:

`\pgfmatrix`{⟨*shape*⟩}{⟨*anchor*⟩}{⟨*name*⟩}{⟨*usage*⟩}{⟨*shift*⟩}{⟨*pre-code*⟩}{⟨*matrix cells*⟩}

This command creates a node that contains a matrix. The name of the node is ⟨*name*⟩, its shape is ⟨*shape*⟩ and the node is anchored at ⟨*anchor*⟩.

The ⟨*matrix cell*⟩ parameter contains the cells of the matrix. In each cell drawing commands may be given, which create a so-called cell picture. For each cell picture a bounding box is computed and the cells are aligned according to the rules outlined in the previous section.

The resulting matrix is used as the `text` box of the node. As for a normal node, the ⟨*usage*⟩ commands are applied, so that the path(s) of the resulting node are stroked or filled or whatever.

Specifiying the cells and rows. Even though this command uses `\halign` internally, there are two special rules for indicating cells:

1. Cells in the same row must be separated using the macro `\pgfmatrixnextcell` rather than `&`. Using `&` will result in an error message.

   However, you can make `&` an active character and have it expand to `\pgfmatrixnextcell`. This way, it will "look" as if `&` is used.

2. Every row *including the last row* must be ended using the command `\\`.

Both `\pgfmatrixnextcell` and `\\` take an optional argument as explained in the Section 50.4

a b
c d

```
\begin{tikzpicture}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{}
    {
      \node {a}; \pgfmatrixnextcell \node {b}; \\
      \node {c}; \pgfmatrixnextcell \node {d}; \\
    }
\end{tikzpicture}
```

Anchoring matrices at nodes inside the matrix. The parameter ⟨*shift*⟩ is an additional negative shift for the node. Normally, such a shift could be given beforehand (that is, the shift could be preapplied to the current transformation matrix). However, when ⟨*shift*⟩ is evaluated, you can refer to *temporary* positions of nodes inside the matrix. In detail, the following happens: When the matrix has been typeset, all nodes in the matrix temporarily get assigned their positions in the matrix box. The origin of this coordinate system is at the left baseline end of the matrix box, which corresponds to the `text` anchor. The position ⟨*shift*⟩ is then interpreted inside this coordinate system and then used for shifting.

This allows you to use the parameter ⟨*shift*⟩ in the following way: If you use `text` as the ⟨*anchor*⟩ and specify `\pgfpointanchor{inner node}{some anchor}` for the parameter ⟨*shift*⟩, where `inner node` is a node that is created in the matrix, then the whole matrix will be shifted such that `inner node.some anchor` lies at the origin of the whole picture.

Rotations and scaling. The matrix node is never rotated or shifted, because the current coordinate transformation matrix is reset (except for the translational part) at the beginning of `\pgfmatrix`. This is intentional and will not change in the future. If you need to rotate the matrix, you must install an appropriate canvas transformation yourself.

However, nodes and stuff inside the cell pictures can be rotated and scaled normally.

Callbacks. At the beginning and at the end of each cell the special macros `\pgfmatrixbegincode`, `\pgfmatrixendcode` and possibly `\pgfmatrixemptycode` are called. The effect is explained in Section 50.5.

Executing extra code. The parameter ⟨*pre-code*⟩ is executed at the beginning of the outermost TEX-group enclosing the matrix node. It is inside this TEX-group, but outside the matrix itself. It can be used for different purposes:

1. It can be used to simplify the next cell macro. For example, saying `\let\&=\pgfmatrixnextcell` allows you to use `\&` instead of `\pgfmatrixnextcell`. You can also set the catcode of `&` to active.
2. It can be used to issue an `\aftergroup` command. This allows you to regain control after the `\pgfmatrix` command. (If you do not know the `\aftergroup` command, you are probably blessed with a simple and happy life.)

Special considerations concerning macro expansion. As said before, the matrix is typeset using `\halign` internally. This command does a lot of strange and magic things like expanding the first macro of every cell in a most unusual manner. Here are some effects you may wish to be aware of:

- It is not necessary to actually mention `\pgfmatrixnextcell` or `\\` inside the ⟨*matrix cells*⟩. It suffices that the macros inside ⟨*matrix cells*⟩ expand to these macros sooner or later.
- In particular, you can define clever macros that insert columns and rows as needed for special effects.

50.4 Row and Column Spacing

It is possible to control the space between columns and rows rather detailedly. Two commands are important for the row spacing and two commands for the column spacing.

`\pgfsetmatrixcolumnsep{`⟨*sep list*⟩`}`

This macro sets the default separation list for columns. The details of the format of this list are explained in the description of the next command.

`\pgfmatrixnextcell[`⟨*additional sep list*⟩`]`

This command has two puposes: First, it is used to separate cells. Second, by providing the optional argument ⟨*additional sep list*⟩ you can modify the spacing between the columns that are separated by this command.

The optional ⟨*additional sep list*⟩ may only be provided when the `\pgfmatrixnextcell` command starts a new column. Normally, this will only be the case in the first row, but sometimes a later row has more elements than the first row. In this case, the `\pgfmatrixnextcell` commands that start the new columns in the later row may also have the optional argument. Once a column has been started, subsequent uses of this optional argument for the column have no effect.

To determine the space between the two columns the are separated by `\pgfmatrixnextcell`, the following algorithm is executed:

1. Both the default separation list (as setup by `\pgfsetmatrixcolumnsep`) and the ⟨*additional sep list*⟩ are processed, in this order. If the ⟨*additional sep list*⟩ argument is missing, only the default separation list is processed.
2. Both lists may contain dimensions, separated by commas, as well as occurences of the keywords `between origins` and `between borders`.
3. All dimensions occuring in either list are added together to arrive at an dimension d.
4. The last occurence of either of the keywords is located. If neither keyword is present, we proceed as if `between borders` were present.

At the end of the algorithm, a dimension d has been computed and one of the two *modes* `between borders` and `between origins` has been determined. Depending on which mode has been determined, the following happens:

- For the `between borders` mode, an additional horizontal space of d is added between the two columns. Note that d may be negative.
- For the `between origins` mode, the spacing between the two columns is computed differently: Recall that the origins of the cell pictures in both pictures lie on two vertical lines. The spacing between the two columns is setup such that the horizontal distance between these two lines is exactly d.

  This mode may only be used between columns *already introduced in the first row.*

All of the above rules boil down to the following effects:

- A default spacing between columns should be setup using `\pgfsetmatrixcolumnsep`. For example, you might say `\pgfsetmatrixcolumnsep{5pt}` to have columns be spaced apart by `5pt`. You could say

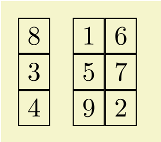
  ```
  \pgfsetmatrixcolumnsep{1cm,between origins}
  ```

  to specify that horizontal space between the origins of cell pictures in adjacent columns should be 1cm by default – regardless of the actual size of the cell pictures.
- You can now use the optional argument of `\pgfmatrixnextcell` to locally overrule the spacing between two columns. By saying `\pgfmatrixnextcell[5pt]` you *add* 5pt to the space between of the two columns, regadless of the mode.

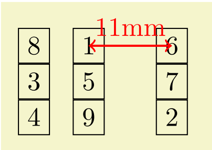
  You can also (locally) change the spacing mode for these two columns. For example, even if the normal spacing mode is `between origins`, you can say

  ```
  \pgfmatrixnextcell[5pt,between borders]
  ```

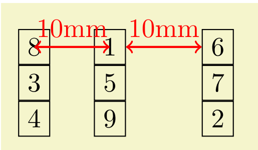
  to locally change the mode for these columns to `between borders`.




```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \pgfsetmatrixcolumnsep{1mm}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node {8}; \&[2mm] \node{1}; \&[-1mm] \node {6}; \\
    \node {3}; \&      \node{5}; \&       \node {7}; \\
    \node {4}; \&      \node{9}; \&       \node {2}; \\
  }
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \pgfsetmatrixcolumnsep{1mm}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node {8}; \&[2mm] \node(a){1}; \&[1cm,between origins] \node(b){6}; \\
    \node {3}; \&      \node   {5}; \&                      \node   {7}; \\
    \node {4}; \&      \node   {9}; \&                      \node   {2}; \\
  }
  \tikzstyle{every node}=[]
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {11mm};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \tikzstyle{every node}=[draw]
  \pgfsetmatrixcolumnsep{1cm,between origins}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node (a) {8}; \& \node (b) {1}; \&[between borders] \node (c) {6}; \\
    \node     {3}; \& \node     {5}; \&                  \node     {7}; \\
    \node     {4}; \& \node     {9}; \&                  \node     {2}; \\
  }
  \tikzstyle{every node}=[]
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm};
  \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway] {10mm};
\end{tikzpicture}
```

The mechanism for the between-row-spacing is the same, only the commands are called differently.

`\pgfsetmatrixrowsep{`⟨*sep list*⟩`}`

This macro sets the default separation list for rows.

`\\[`⟨*additional sep list*⟩`]`

This command ends a line. The optional ⟨*additional sep list*⟩ is used to determine the spacing between the row being ended and the next row. The modes and the computation of d is done in the same way as for columns. For the last row the optional argument has no effect.

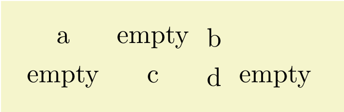
This command is defined in this special way only inside matrices.

50.5 Callbacks

There are three macros that get called at the beginning and end of cells. By redefining these macros, which are empty by default, you can change the appearance of cells in a very general manner.

`\pgfmatrixemptycode`

This macro is executed for empty cells. This means that PGF uses some macro magic to determine whether a cell is empty (it immediatly ends with `\pgfmatrixemptycode` or `\\`) and, if so, put this macro inside the cell.

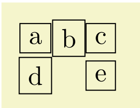



```
\begin{tikzpicture}
  \def\pgfmatrixemptycode{\node{empty};}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node {a}; \&           \& \node {b}; \\
               \& \node{c}; \& \node {d}; \& \\
  }
\end{tikzpicture}
```

As can be seen, the macro is not executed for empty cells at the end of row when columns are added only later on.

`\pgfmatrixbegincode`

This macro is executed at the beginning of non-empty cells. Correspondingly, `\pgfmatrixendcode` is added at the end of every non-empty cell.




```
\begin{tikzpicture}
  \def\pgfmatrixbegincode{\node[draw]\bgroup}
  \def\pgfmatrixendcode{\egroup;}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    a \& b \& c \\
    d \&   \& e \\
  }
\end{tikzpicture}
```

Note that between `\pgfmatrixbegincode` and `\pgfmatrixendcode` there will *not* only be the contents of the cell. Rather, PGF will add some (invisible) commands for book-keeping purposes that involve `\let` and `\gdef`. In particular, it is not a good idea to have `\pgfmatrixbegincode` end with `\csname` and `\pgfmatrixendcode` start with `\endcsname`.

`\pgfmatrixendcode`

See the explanation above.

The following two counters allow you to access the current row and current column in a callback:

`\pgfmatrixcurrentrow`

This counter stores the current row of the current cell of the matrix. Do not even think of changing this counter.

`\pgfmatrixcurrentcolumn`

This counter stores the current column of the current cell of the matrix.

51 Coordinate and Canvas Transformations

51.1 Overview

PGF offers two different ways of scaling, shifting, and rotating (these operations are generally known as *transformations*) graphics: You can apply *coordinate transformations* to all coordinates and you can apply *canvas transformations* to the canvas on which you draw. (The names "coordinate" and "canvas" transformations are not standard, I introduce them only for the purposes of this manual.)

The difference is the following:

- As the name "coordinate transformation" suggests, coordinate transformations apply only to coordinates. For example, when you specify a coordinate like `\pgfpoint{1cm}{2cm}` and you wish to "use" this coordinate—for example as an argument to a `\pgfpathmoveto` command—then the coordinate transformation matrix is applied to the coordinate, resulting in a new coordinate. Continuing the example, if the current coordinate transformation is "scale by a factor of two," the coordinate `\pgfpoint{1cm}{2cm}` actually designates the point $(2\text{cm}, 4\text{cm})$.

  Note that coordinate transformations apply *only* to coordinates. They do not apply to, say, line width or shadings or text.

- The effect of a "canvas transformation" like "scale by a factor of two" can be imagined as follows: You first draw your picture on a "rubber canvas" normally. Then, once you are done, the whole canvas is transformed, in this case stretched by a factor of two. In the resulting image *everything* will be larger: Text, lines, coordinates, and shadings.

In many cases, it is preferable that you use coordinate transformations and not canvas transformations. When canvas transformations are used, PGF looses track of the coordinates of nodes and shapes. Also, canvas transformations often cause undesirable effects like changing text size. For these reasons, PGF makes it easy to setup the coordinate transformation, but a bit harder to change the canvas transformation.

51.2 Coordinate Transformations

51.2.1 How PGF Keeps Track of the Coordinate Transformation Matrix

PGF has an internal coordinate transformation matrix. This matrix is applied to coordinates "in certain situations." This means that the matrix is not always applied to every coordinate "no matter what." Rather, PGF tries to be reasonably smart at when and how this matrix should be applied. The most prominent examples are the path construction commands, which apply the coordinate transformation matrix to their inputs.

The coordinate transformation matrix consists of four numbers a, b, c, and d, and two dimensions s and t. When the coordinate transformation matrix is applied to a coordinate (x, y) the new coordinate $(ax + by + s, cx + dy + t)$ results. For more details on how transformation matrices work in general, please see, for example, the PDF or PostScript reference or a textbook on computer graphics.

The coordinate transformation matrix is equal to the identity matrix at the beginning. More precisely, $a = 1$, $b = 0$, $c = 0$, $d = 1$, $s = 0\text{pt}$, and $t = 0\text{pt}$.

The different coordinate transformation commands will modify the matrix by concatenating it with another transformation matrix. This way the effect of applying several transformation commands will *accumulate*.

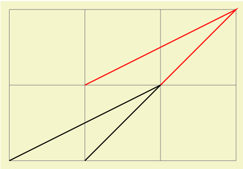
The coordinate transformation matrix is local to the current TEX group (unlike the canvas transformation matrix, which is local to the current `{pgfscope}`). Thus, the effect of adding a coordinate transformation to the coordinate transformation matrix will last only till the end of the current TEX group.

51.2.2 Commands for Relative Coordinate Transformations

The following commands add a basic coordinate transformation to the current coordinate transformation matrix. For all commands, the transformation is applied *in addition* to any previous coordinate transformations.

`\pgftransformshift{`⟨*point*⟩`}`

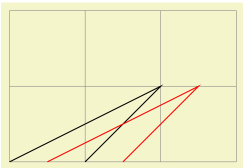
Shifts coordinates by ⟨*point*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformxshift`{⟨*dimensions*⟩}

Shifts coordinates by ⟨*dimension*⟩ along the x-axis.



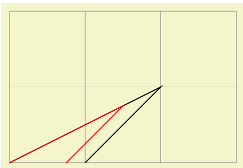
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxshift{.5cm}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformyshift`{⟨*dimensions*⟩}

Like `\pgftransformxshift`, only for the y-axis.

`\pgftransformscale`{⟨*factor*⟩}

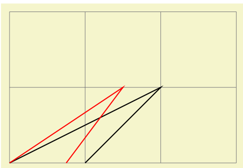
Scales coordinates by ⟨*factor*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformscale{.75}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformxscale`{⟨*factor*⟩}

Scales coordinates by ⟨*factor*⟩ in the x-direction.



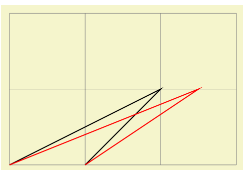
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxscale{.75}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformyscale`{⟨*factor*⟩}

Like `\pgftransformxscale`, only for the y-axis.

`\pgftransformxslant`{⟨*factor*⟩}

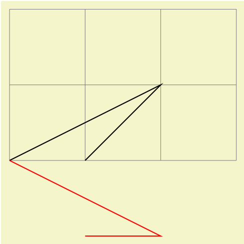
Slants coordinates by ⟨*factor*⟩ in the x-direction. Here, a factor of `1` means 45°.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxslant{.5}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformyslant`{⟨*factor*⟩}

Slants coordinates by ⟨*factor*⟩ in the y-direction.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformyslant{-1}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

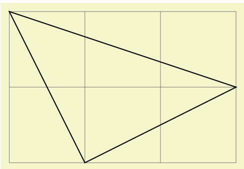
\pgftransformrotate{⟨*degrees*⟩}

Rotates coordinates counterclockwise by ⟨*degrees*⟩.

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformrotate{30}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformtriangle{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}

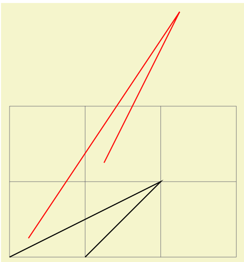
This command transforms the coordinate system in such a way that the triangle given by the points ⟨*a*⟩, ⟨*b*⟩ and ⟨*c*⟩ lies at the coordinates $(0,0)$, $(1\text{pt}, 0\text{pt})$ and $(0\text{pt}, 1\text{pt})$.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformtriangle
  {\pgfpoint{1cm}{0cm}}
  {\pgfpoint{0cm}{2cm}}
  {\pgfpoint{3cm}{1cm}}

  \draw (0,0) -- (1pt,0pt) -- (0pt,1pt) -- cycle;
\end{tikzpicture}
```

\pgftransformcm{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*point*⟩}

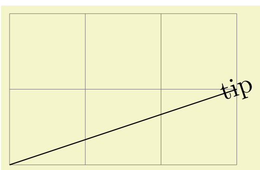
Applies the transformation matrix given by a, b, c, and d and the shift ⟨*point*⟩ to coordinates (in addition to any previous transformations already in force).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformcm{1}{1}{0}{1}{\pgfpoint{.25cm}{.25cm}}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformarrow{⟨*start*⟩}{⟨*end*⟩}

Shift coordinates to the end of the line going from ⟨*start*⟩ to ⟨*end*⟩ with the correct rotation.

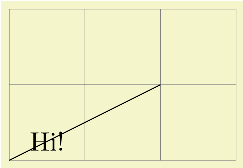



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (3,1);
  \pgftransformarrow{\pgfpointorigin}{\pgfpoint{3cm}{1cm}}
  \pgftext{tip}
\end{tikzpicture}
```

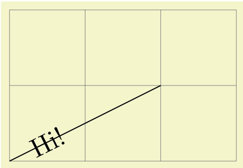
\pgftransformlineattime{⟨*time*⟩}{⟨*start*⟩}{⟨*end*⟩}

Shifts coordinates by a specific point on a line at a specific time. The point by which the coordinate is shifted is calculated by calling `\pgfpointlineattime`, see Section 44.5.2.

In addition to shifting the coordinate, a rotation *may* also be applied. Whether this is the case depends on whether the TeX if `\ifpgfslopedattime` is set to true or not.



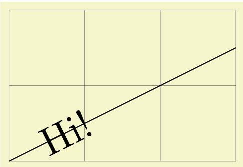

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw       (0,0) -- (2,1);
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```



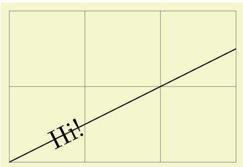

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw       (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

If `\ifpgfslopedattime` is true, another TeX `\if` is important: `\ifpgfallowupsidedowattime`. If this is false, PGF will ensure that the rotation is done in such a way that text is never "upside down."

There is another TeX if that influences this command. If you set `\ifpgfresetnontranslationattime` to true, then, between shifting the coordinate and (possibly) rotating/sloping the coordinate, the command `\pgftransformresetnontranslations` is called. See the description of this command for details.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{1.5}
  \draw       (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgfresetnontranslationattimefalse
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

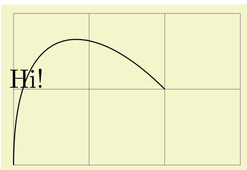



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{1.5}
  \draw       (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgfresetnontranslationattimetrue
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

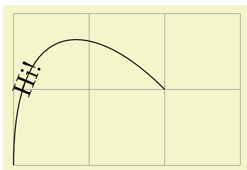
`\pgftransformcurveattime{`⟨*time*⟩`}{`⟨*start*⟩`}{`⟨*first support*⟩`}{`⟨*second support*⟩`}{`⟨*end*⟩`}`

Shifts coordinates by a specific point on a curve at a specific time, see Section 44.5.2 once more.

As for the line-at-time transformation command, `\ifpgfslopedattime` decides whether an additional rotation should be applied. Again, the value of `\ifpgfallowupsidedowattime` is also considered.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw       (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw       (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgfslopedattimetrue
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

The value of `\ifpgfresetnontranslationsattime` is also taken into account.

`\ifpgfslopedattime`

Decides whether the "at time" transformation commands also rotate coordinates or not.

`\ifpgfallowupsidedowattime`

Decides whether the "at time" transformation commands should allow the rotation be down in such a way that "upside-down text" can result.

`\ifpgfresetnontranslationsattime`

Decides whether the "at time" transformation commands should reset the non-translations between shifting and rotating.

51.2.3 Commands for Absolute Coordinate Transformations

The coordinate transformation commands introduced up to now are always applied in addition to any previous transformations. In contrast, the commands presented in the following can be used to change the transformation matrix "absolutely." Note that this is, in general, dangerous and will often produce unexpected effects. You should use these commands only if you really know what you are doing.

`\pgftransformreset`

Resets the coordinate transformation matrix to the identity matrix. Thus, once this command is given no transformations are applied till the end of the scope.

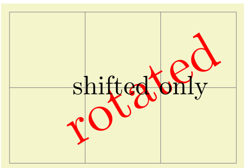
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformreset
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

`\pgftransformresetnontranslations`

This command sets the a, b, c, and d part of the coordinate transformation matrix to $a = 1$, $b = 0$, $c = 0$, and $d = 1$. However, the current shifting of the matrix is not modified.

The effect of this command is that any rotation/scaling/slanting is undone in the current TEX group, but the origin is not "moved back."

This command is mostly useful directly before a `\pgftext` command to ensure that the text is not scaled or rotated.

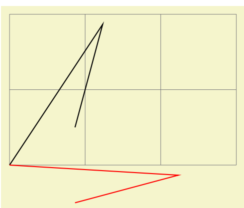



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{2}
  \pgftransformrotate{30}
  \pgftransformxshift{1cm}
  {\color{red}\pgftext{rotated}}
  \pgftransformresetnontranslations
  \pgftext{shifted only}
\end{tikzpicture}
```

`\pgftransforminvert`

Replaces the coordinate transformation matrix by a coordinate transformation matrix that "exactly undoes the original transformation." For example, if the original transformation was "scale by 2 and then shift right by 1cm" the new one is "shift left by 1cm and then scale by 1/2."

This command will produce an error if the determinant of the matrix is too small, that is, if the matrix is near-singular.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransforminvert
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

51.2.4 Saving and Restoring the Coordinate Transformation Matrix

There are two commands for saving and restoring coordinate transformation matrices.

`\pgfgettransform`{⟨*macro*⟩}

This command will (locally) define ⟨*macro*⟩ to a representation of the current coordinate transformation matrix. This matrix can later on be reinstalled using `\pgfsettransform`.

`\pgfsettransform`{⟨*macro*⟩}

Reinstalls a coordinate transformation matrix that was previously saved using `\pgfgettransform`.

51.3 Canvas Transformations

The canvas transformation matrix is not managed by PGF, but by the output format like PDF or PostScript. All the PGF does is to call appropriate low-level `\pgfsys@` commands to change the canvas transformation matrix.

Unlike coordinate transformations, canvas transformations apply to "everything," including images, text, shadings, line thickness, and so on. The idea is that a canvas transformation really stretches and deforms the canvas after the graphic is finished.

Unlike coordinate transformations, canvas transformations are local to the current `{pgfscope}`, not to the current TeX group. This is due to the fact that they are managed by the backend driver, not by TeX or PGF.

Unlike the coordinate transformation matrix, it is not possible to "reset" the canvas transformation matrix. The only way to change it is to concatenate it with another canvas transformation matrix or to end the current `{pgfscope}`.

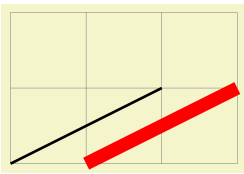
Unlike coordinate transformations, PGF does not "keep track" of canvas transformations. In particular, it will not be able to correctly save the coordinates of shapes or nodes when a canvas transformation is used.

PGF does not offer a whole set of special commands for modifying the canvas transformation matrix. Instead, different commands allow you to concatenate the canvas transformation matrix with a coordinate transformation matrix (and there are numerous commands for specifying a coordinate transformation, see the previous section).

`\pgflowlevelsynccm`

This command concatenates the canvas transformation matrix with the current coordinate transformation matrix. Afterward, the coordinate transformation matrix is reset.

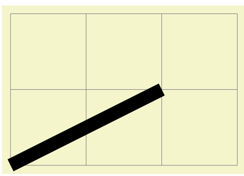
The effect of this command is to "synchronize" the coordinate transformation matrix and the canvas transformation matrix. All transformations that were previously applied by the coordinate transformations matrix are now applied by the canvas transformation matrix.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgftransformscale{5}
  \draw      (0,0) -- (0.4,.2);
  \pgftransformxshift{0.2cm}
  \pgflowlevelsynccm
  \draw[red] (0,0) -- (0.4,.2);
\end{tikzpicture}
```

`\pgflowlevel`{⟨*transformation code*⟩}

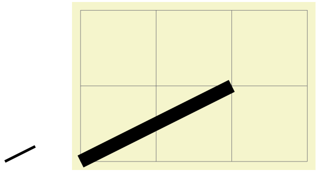
This command concatenates the canvas transformation matrix with the coordinate transformation specified by ⟨*transformation code*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevel{\pgftransformscale{5}}
  \draw      (0,0) -- (0.4,.2);
\end{tikzpicture}
```

\pgflowlevelobj{⟨*transformation code*⟩}{⟨*code*⟩}

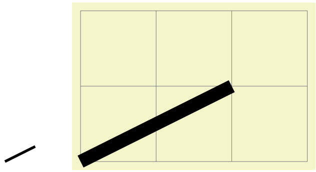
This command creates a local {pgfscope}. Inside this scope, \pgflowlevel is first called with the argument ⟨*transformation code*⟩, then the ⟨*code*⟩ is inserted.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevelobj{\pgftransformscale{5}}    {\draw (0,0) -- (0.4,.2);}
  \pgflowlevelobj{\pgftransformxshift{-1cm}}{\draw (0,0) -- (0.4,.2);}
\end{tikzpicture}
```

\begin{pgflowlevelscope}{⟨*transformation code*⟩}
⟨*environment contents*⟩
\end{pgflowlevelscope}

This environment first surrounds the ⟨*environment contents*⟩ by a {pgfscope}. Then it calls \pgflowlevel with the argument ⟨*transformation code*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \begin{pgflowlevelscope}{\pgftransformscale{5}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
  \begin{pgflowlevelscope}{\pgftransformxshift{-1cm}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
\end{tikzpicture}
```

\pgflowlevelscope{⟨*transformation code*⟩}
⟨*environment contents*⟩
\endpgflowlevelscope

Plain TEX version of the environment.

\startpgflowlevelscope{⟨*transformation code*⟩}
⟨*environment contents*⟩
\stoppgflowlevelscope

ConTEXt version of the environment.

52 Patterns

```
\usepackage{pgfbasepattterns} % LaTeX
\input pgfbasepattterns.tex   % plain TeX
\usemodule[pgfbasepattterns]  % ConTeXt
```

This package provides commands for declaring and using patterns. The package is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

52.1 Overview

There are many ways of filling a path. First, you can fill it using a solid color and this is also the fasted method. Second, you can also fill it using a shading, which means that the color changes smoothly between two (or more) different colors. Third, you can fill it using a tiling pattern and it is explained in the following how this is done.

A tiling pattern can be imagined as a rectangular tile (hence the name) on which a small picture is painted. There is not a single tile, but (conceptually) an infinite number of tiles, all showing the same picture, and these tiles are arranged horizontally and vertically to fill the plane. When you use a tiling pattern to fill a path, what happens is that the path clips out a "window" through which we see part of this infinite plane.

Patterns come in two versions: *inherently colored patterns* and *form-only patterns*. (These are often called "color patterns" and "uncolored patterns," but these names are misleading since uncolored patterns do have a color and the color changes. As I said, the name is misleading...) An inherently colored pattern is just a colored tile like, say, a red star with a black outline. A form-only pattern can be imagined as a tile that is a kind of rubber stamp. When this pattern is used, the stamp is used to print copies of the stamp picture onto the plane, but we can use a different stamp color each time we use a form-only pattern.

PGF provides a special support for patterns. You can declare a pattern and then use it very much like a fill color. PGF directly maps patterns to the pattern facilities of the underlying graphic languages (PostScript, PDF, and SVG). This means that filling a path using a pattern will be nearly as fast as if you used a uniform color.

There are a number of pitfalls and restrictions when using patterns. First, once a pattern has been declared, you cannot change it anymore. In particular, it is not possible to enlarge it or change the line width. Such flexibility would require that the repeating of the pattern were not done by the graphic language, but on the PGF level. This would make patterns orders of magnitude slower to produce and to render.

Second, the phase of patterns is not well-defined, that is, it is not clear where origin of the "first" tile is. To be more precise, PostScript and PDF on the one hand and SVG on the other hand define the origin differently. PostScript and PDF define a fixed origin that is independent of where the path lies. This has the highly desirable effect that if you use the same pattern to fill multiple paths, this has the same effect as if you used the pattern to will a single path that is the union of all the paths. By comparison, SVG uses the upper-left (?) corner of the path to be filled as the origin. However, the SVG specification is a bit vague on this question.

52.2 Declaring a Pattern

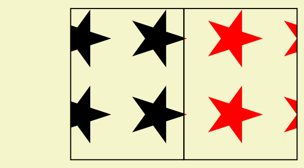
Before a pattern can be used, it must be declared. The following command is used for this:

`\pgfdeclarepatternformonly{`⟨*name*⟩`}{`⟨*lower left*⟩`}{`⟨*upper right*⟩`}{`⟨*tile size*⟩`}{`⟨*code*⟩`}`

This command declares a new form-only pattern. The `{`⟨*name*⟩`}` is a name for later reference. The two parameters `{`⟨*lower left*⟩`}` and `{`⟨*upper right*⟩`}` must describe a bounding box that is large enough to encompass the complete tile.

The size of a tile is given by ⟨*tile size*⟩, that is, a tile is a rectangle whose lower left corner is the origin and whose upper right corner is given by ⟨*tile size*⟩. This might make you wonder why the second and third parameters are needed. First, the bounding box might be smaller than the tile size if the tile is larger than the picture on the tile. Second, the bounding box might be bigger, in which case the picture will "bleed" over the tile.

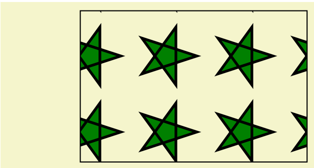
The ⟨*code*⟩ should be PGF code than can be protocolled. It should not contain any color code.



```
\pgfdeclarepatternformonly{stars}
{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
{\pgfpoint{1cm}{1cm}}
{
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{fill}
}
\begin{tikzpicture}
  \filldraw[pattern=stars] (0,0)   rectangle (1.5,2);
  \filldraw[pattern=stars,pattern color=red]
                            (1.5,0) rectangle (3,2);
\end{tikzpicture}
```

`\pgfdeclarepatterninherentlycolored{`⟨*name*⟩`} {`⟨*lower left*⟩`} {`⟨*upper right*⟩`} {`⟨*tile size*⟩`} {`⟨*code*⟩`}`

This command works like `\pgfdeclarepatternuncolored`, only the pattern will have an inherent color. To set the color, you should use PGF's color commands, not the `\color` command, since this fill not be protocolled.



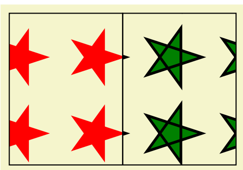
```
\pgfdeclarepatterninherentlycolored{green stars}
{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
{\pgfpoint{1cm}{1cm}}
{
  \pgfsetfillcolor{green!50!black}
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{stroke,fill}
}
\begin{tikzpicture}
  \filldraw[pattern=green stars] (0,0) rectangle (3,2);
\end{tikzpicture}
```

52.3 Setting a Pattern

Once a pattern has been declared, it can be used.

`\pgfsetfillpattern{`⟨*name*⟩`}{`⟨*color*⟩`}`

This command specifies that paths that are filled should be filled with the "color" by the pattern ⟨*name*⟩. For an inherently colored pattern, the ⟨*color*⟩ parameter is ignored. For form-only patterns, the ⟨*color*⟩ parameter specified the color to be used for the pattern.



```
\begin{tikzpicture}
  \pgfsetfillpattern{stars}{red}
  \filldraw (0,0) rectangle (1.5,2);

  \pgfsetfillpattern{green stars}{red}
  \filldraw (1.5,0) rectangle (3,2);
\end{tikzpicture}
```

53 Declaring and Using Images

This section describes the `pgfbaseimage` package.

```
\usepackage{pgfbaseimage} % LaTeX
\input pgfbaseimage.tex   % plain TeX
\usemodule[pgfbaseimage]  % ConTeXt
```

This package offers an abstraction of the image inclusion process. It is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

53.1 Overview

To be quite frank, LaTeX's `\includegraphics` is designed better than `pgfbaseimage`. For this reason, *I recommend that you use the standard image inclusion mechanism of your format.* Thus, LaTeX users are encouraged to use `\includegraphics` to include images.

However, there are reasons why you might need to use the image inclusion facilities of PGF:

- There is no standard image inclusion mechanism in your format. For example, plain TeX does not have one, so PGF's inclusion mechanism is "better than nothing."

  However, this applies only to the `pdftex` backend. For all other backends, PGF currently maps its commands back to the `graphicx` package. Thus, in plain TeX, this does not really help. It might be a good idea to fix this in the future such that PGF becomes independent of LaTeX, thereby providing a uniform image abstraction for all formats.

- You wish to use masking. This is a feature that is only supported by PGF, though I hope that someone will implement this also for the graphics package in LaTeX in the future.

Whatever your choice, you can still use the usual image inclusion facilities of the `graphics` package.

The general approach taken by PGF to including an image is the following: First, `\pgfdeclareimage` declares the image. This must be done prior to the first use of the image. Once you have declared an image, you can insert it into the text using `\pgfuseimage`. The advantage of this two-phase approach is that, at least for PDF, the image data will only be included once in the file. This can drastically reduce the file size if you use an image repeatedly, for example in an overlay. However, there is also a command called `\pgfimage` that declares and then immediately uses the image.

To speedup the compilation, you may wish to use the following class option:

`\usepackage[draft]{pgf}`

In draft mode boxes showing the image name replace the images. It is checked whether the image files exist, but they are not read. If either height or width is not given, 1cm is used instead.

53.2 Declaring an Image

`\pgfdeclareimage[`⟨*options*⟩`]{`⟨*image name*⟩`}{`⟨*filename*⟩`}`

Declares an image, but does not paint anything. To draw the image, use `\pgfuseimage{`⟨*image name*⟩`}`. The ⟨*filename*⟩ may not have an extension. For PDF, the extensions `.pdf`, `.jpg`, and `.png` will automatically tried. For PostScript, the extensions `.eps`, `.epsi`, and `.ps` will be tried.

The following options are possible:

- `height=`⟨*dimension*⟩ sets the height of the image. If the width is not specified simultaneously, the aspect ratio of the image is kept.
- `width=`⟨*dimension*⟩ sets the width of the image. If the height is not specified simultaneously, the aspect ratio of the image is kept.
- `page=`⟨*page number*⟩ selects a given page number from a multipage document. Specifying this option will have the following effect: first, PGF tries to find a file named

  ⟨*filename*⟩`.page`⟨*page number*⟩`.`⟨*extension*⟩

  If such a file is found, it will be used instead of the originally specified filename. If not, PGF inserts the image stored in ⟨*filename*⟩`.`⟨*extension*⟩ and if a recent version of `pdflatex` is used, only the selected page is inserted. For older versions of `pdflatex` and for `dvips` the complete document is inserted and a warning is printed.

- interpolate=⟨*true or false*⟩ selects whether the image should "smoothed" when zoomed. False by default.
- mask=⟨*mask name*⟩ selects a transparency mask. The mask must previously be declared using `\pgfdeclaremask` (see below). This option only has an effect for `pdf`. Not all viewers support masking.

```
\pgfdeclareimage[interpolate=true,height=1cm]{image1}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]{image2}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{brave-gnu-world-logo}
```

`\pgfaliasimage{`⟨*new image name*⟩`}{`⟨*existing image name*⟩`}`

The `{`⟨*existing image name*⟩`}` is "cloned" and the `{`⟨*new image name*⟩`}` can now be used whenever original image is used. This command is useful for creating aliases for alternate extensions and for accessing the last image inserted using `\pgfimage`.

Example: `\pgfaliasimage{image.!30!white}{image.!25!white}`

53.3 Using an Image

`\pgfuseimage{`⟨*image name*⟩`}`

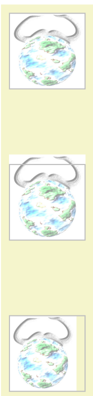
Inserts a previously declared image into the *normal text*. If you wish to use it in a `{pgfpicture}` environment, you must put a `\pgftext` around it.

If the macro `\pgfalternateextension` expands to some nonempty ⟨*alternate extension*⟩, PGF will first try to use the image names ⟨*image name*⟩`.`⟨*alternate extension*⟩. If this image is not defined, PGF will next check whether ⟨*alternate extension*⟩ contains a `!` character. If so, everything up to this exclamation mark and including it is deleted from ⟨*alternate extension*⟩ and the PGF again tries to use the image ⟨*image name*⟩`.`⟨*alternate extension*⟩. This is repeated until ⟨*alternate extension*⟩ no longer contains a `!`. Then the original image is used.

The `xxcolor` package sets the alternate extension to the current color mixin.

```
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]
  {image1}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,width=1cm]{image2}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{brave-gnu-world-logo}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{5cm},left,base]{\pgfuseimage{image1}}
  \pgftext[at=\pgfpoint{1cm}{3cm},left,base]{\pgfuseimage{image2}}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image3}}

  \pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The following example demonstrates the effect of using `\pgfuseimage` inside a color mixin environment.



```
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]
  {image1.!25!white}{brave-gnu-world-logo.25}
\pgfdeclareimage[interpolate=true,width=1cm]
  {image2.25!white}{brave-gnu-world-logo.25}
\pgfdeclareimage[interpolate=true,height=1cm]
  {image3.white}{brave-gnu-world-logo.25}
\begin{colormixin}{25!white}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{5cm},left,base]{\pgfuseimage{image1}}
  \pgftext[at=\pgfpoint{1cm}{3cm},left,base]{\pgfuseimage{image2}}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image3}}

  \pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
\end{colormixin}
```

\pgfalternateextension

You should redefine this command to install a different alternate extension.

Example: `\def\pgfalternateextension{!25!white}`

\pgfimage[⟨*options*⟩]{⟨*filename*⟩}

Declares the image under the name `pgflastimage` and immediately uses it. You can "save" the image for later usage by invoking `\pgfaliasimage` on `pgflastimage`.

```
\begin{colormixin}{25!white}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{5cm},left,base]
    {\pgfimage[interpolate=true,width=1cm,height=1cm]{brave-gnu-world-logo}}
  \pgftext[at=\pgfpoint{1cm}{3cm},left,base]
    {\pgfimage[interpolate=true,width=1cm]{brave-gnu-world-logo}}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]
    {\pgfimage[interpolate=true,height=1cm]{brave-gnu-world-logo}}

  \pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
\end{colormixin}
```

53.4 Masking an Image

\pgfdeclaremask[⟨*options*⟩]{⟨*mask name*⟩}{⟨*filename*⟩}

Declares a transparency mask named ⟨*mask name*⟩ (called a *soft mask* in the PDF specification). This mask is read from the file ⟨*filename*⟩. This file should contain a grayscale image that is as large as the actual image. A white pixel in the mask will correspond to "transparent," a black pixel to "solid," and gray values correspond to intermediate values. The mask must have a single "color channel." This means that the mask must be a "real" grayscale image, not an RGB-image in which all RGB-triples happen to have the same components.

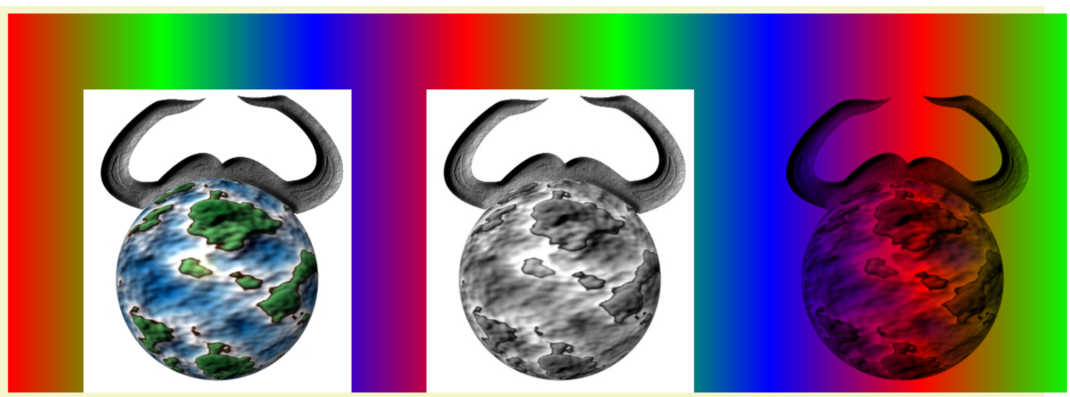
You can only mask images the are in a "pixel format." These are `.jpg` and `.png`. You cannot mask `.pdf` images in this way. Also, again, the mask file and the image file must have the same size.

The following options may be given:

- `matte={`⟨*color components*⟩`}` sets the so-called *matte* of the actual image (strangely, this has to be specified together with the mask, not with the image itself). The matte is the color that has been used to preblend the image. For example, if the image has been preblended with a red background, then ⟨*color components*⟩ should be set to `{1 0 0}`. The default is `{1 1 1}`, which is white in the rgb model.

  The matte is specified in terms of the parent's image color space. Thus, if the parent is a grayscale image, the matte has to be set to `{1}`.

Example:



```
%% Draw a large colorful background
\pgfdeclarehorizontalshading{colorful}{5cm}{color(0cm)=(red);
color(2cm)=(green); color(4cm)=(blue); color(6cm)=(red);
color(8cm)=(green); color(10cm)=(blue); color(12cm)=(red);
color(14cm)=(green)}
\hbox{\pgfuseshading{colorful}\hskip-14cm\hskip1cm
\pgfimage[height=4cm]{brave-gnu-world-logo}\hskip1cm
\pgfimage[height=4cm]{brave-gnu-world-logo-mask}\hskip1cm
\pgfdeclaremask{mymask}{brave-gnu-world-logo-mask}
\pgfimage[mask=mymask,height=4cm,interpolate=true]{brave-gnu-world-logo}}
```

54 Externalizing Graphics

54.1 Overview

There are two fundamentally different ways of inserting graphics into a TEX-document. First, you can create a graphic using some external program like `xfig` or `InDesign` and then include this graphic in your text. This is done using commands like `\includegraphics` or `\pgfimage`. In this case, the graphic file contains all the low-level graphic commands that describe the picture. When such a file is included, all TEX has to worry about is the size of the picture; the internals of the picture are unknown to TEX and it does not care about them.

The second method of creating graphics is to use a special package that transforms TEX-commands like `\draw` or `\psline` into appropriate low-level graphic commands. In this case, TEX has to do all the hard work of "typesetting" the picture and if a picture has a complicated internal structure this may take a lot of time.

While PGF was created to facilitate the second method of creating pictures, there are two main reasons why you may need to employ the first method of image-inclusion, nevertheless:

1. Typesetting a picture using TEX can be a very time-consuming process. If TEX needs a minute to typeset a picture, you do not want to wait this minute when you reTEX your document after having changed a single comma.
2. Some users, especially journal editors, may not be able to process files that contain PGF commands – for the simple reason that the systems of many publishing houses do not have PGF installed.

In both cases, the solution is to "extract" or "externalize" pictures that would normally be typeset every time a document is TEXed. Once the pictures have been extracted into separate graphics files, these graphic files can be reinserted into the text using the first method.

Extracting a graphic from a file is not as easy as it may sound at first since TEX cannot write parts of its output into different files. A bit of trickery is needed and the `pgfbaseimage` package provides macros that simplify the following workflow:

1. You have to tell PGF which files will be used for which pictures. To do so, you enclose each picture that you wish to be "externalized" in a pair of `\beginpgfgraphicnamed` and `\endpgfgraphicnamed` macros.
2. The next step is to generate the extracted graphics. For this you run TEX with the `\jobname` set to the graphic file's name. This will cause `\pgfname` to behave in a very special way: All of your document will simply be thrown away, *except* for the single graphic having the same name as the current jobname.
3. After you have run TEX once for each graphic that your wish to externalize, you can rerun TEX on your document normally. This will have the following effect: Each time a `\beginpgfgraphicnamed` is encountered, PGF checks whether a graphic file of the given name exists (if you did step 2, it will). If this graphic file exists, it will be input and the text till the corresponding `\endpgfgraphicnamed` will be ignored.

In the rest of this section, the above workflow is explained in more detail.

54.2 Workflow Step 1: Naming Graphics

In order to put each graphic in an external file, you first need to tell PGF the names of these files. For this, you use a pair of commands that are declared in the package `pgfbaseimage`.

`\beginpgfgraphicnamed{`⟨*file name prefix*⟩`}`

This command indicates that everything up to the next call of `\endpgfgraphicnamed` is part of a graphic that should be placed in a file named ⟨*file name prefix*⟩`.`⟨*suffix*⟩, where the ⟨*suffix*⟩ depends on your backend driver. Typically, ⟨*suffix*⟩ will be `dvi` or `pdf`.

Here is a typical example of how this command is used:

```
% In file main.tex:
...
As we see in Figure~\ref{fig1}, the world is flat.
\begin{figure}
  \beginpgfgraphicnamed{graphic-of-flat-world}
  \begin{tikzpicture}
    \fill (0,0) circle (1cm);
  \end{tikzpicture}
  \endpgfgraphicnamed
  \caption{The flat world.}
  \label{fig1}
\end{figure}
```

Each graphic that is be externalized should have a unique name. Note that this name will be used as the name of a file in the file system, so it should not contain any funny characters.

This command can have three different effects:

1. The easiest situation arises if there does not yet exist a graphic file called ⟨*file name prefix*⟩`.`⟨*suffix*⟩, where the ⟨*suffix*⟩ is one of the suffixes understood by your current backend driver (so `pdf` or `jpg` if you use `pdftex`, `eps` if you use `dvips`, and so on). In this case, both this command and the `\endpgfgraphicnamed` command simply have no effect.
2. A more complex situation arises when a graphic file named ⟨*file name prefix*⟩`.`⟨*suffix*⟩ *does* exist. In this case, this graphic file is included using the `\includegraphics` command. Furthermore, the text between `\beginpgfgraphicnamed` and `\endpgfgraphicnamed` is ignored.

   When the text is "ignored," what actually happens is that all text up to the next occurrence of `\endpgfgraphicnamed` is thrown away without any macro expansion. This means, in particular, that (a) you cannot put `\endpgfgraphicnamed` inside a macro and (b) the macros used in the graphics need not be defined at all when the graphic file is included.
3. The most complex behaviour arises when current the `\jobname` equals the ⟨*file name prefix*⟩ and, furthermore, the a *real job name* has been declared. The behaviour for this case is explained later.

Note that the `\beginpgfgraphicnamed` does not really have any effect until you have generated the graphic files named. Till then, this command is simply ignored. Also, if you delete the graphics file later on, the graphics are typeset normally once more.

`\endpgfgraphicnamed`

This command just marks the end of the graphic that should be externalized.

54.3 Workflow Step 2: Generating the External Graphics

We have now indicated all the graphics for which we would like graphic files to be generated. In order to generate the files, you now need to modify the `\jobname` appropriately. This is done in two steps:

1. You use the following command to tell PGF the real name of your `.tex` file:

   `\pgfrealjobname{`⟨*name*⟩`}`

   Tells PGF the real name of your job. For instance, if you have a file called `survey.tex` that contains two graphics that you wish to be called `survey-graphic1` and `survey-graphic2`, then you should write the following.

   ```
   % This is file survey.tex
   \documentclass{article}
   ...
   \usepackage{tikz}
   \pgfrealjobname{survey}
   ```

2. You run TeX with the `\jobname` set to the name of the graphic for which you need an external graphic to be generated. To set the `\jobname`, you use the `--jobname=` option of TeX:

   ```
   bash> latex --jobname=survey-graphic1 survey.tex
   ```

The following things will now happen:

1. \pgfrealjobname notices that the \jobname is not the "real" jobname and, thus, must be the name of a graphic that is to be put in an external file.

2. At the beginning of the document, PGF changes the definition of TEX's internal \shipout macro. The new shipout macro simply throws away the output. This means that the document is typeset normally, but no output is produced.

3. When the \beginpgfgraphicnamed{⟨*name*⟩} command is encountered where the ⟨*name*⟩ is the same as the current \jobname, then a TEX-box is started and ⟨*everything*⟩ up to the following \endpgfgraphicnamed command is stored inside this box.

   Note that, typically, ⟨*everything*⟩ will contain just a single {tikzpicture} or {pgfpicture} environment. However, this need not be the case, you use, say, a {pspicture} environment as ⟨*everything*⟩ or even just some normal TEX-text.

4. At the \endpgfgraphicnamed, the box *is* shipped out using the original \shipout command. Thus, unlike everything else, the contents of the graphic is made part of the output.

5. When the box containing the graphic is shipped out, the paper size is modified such that it exactly equal to the height and width of the box.

The net effect of everything described above is that the two commands

```
bash> latex --jobname=survey-graphic1 survey.tex
bash> dvips survey-graphic1
```

produce a file called survey-graphic1.ps that consists of a single page that contains exactly the graphic produced by the code between \beginpgfgraphicnamed{survey-graphic1} and \endpgfgraphicnamed. Furthermore, the size of this single page is exactly the size of the graphic.

If you use pdfTEX, producing the graphic is even simpler:

```
bash> pdflatex --jobname=survey-graphic1 survey.tex
```

produces the single-page pdf-file survey-graphic1.pdf.

54.4 Workflow Step 3: Including the External Graphics

Once you have produced all the pictures in the text, including them into the main document is easy: Simply run TEX again without any modification of the \jobname. In this case the \pgfrealjobname command will notice that the main file is, indeed, the main file. The main file will then be typeset normally and the \beginpgfgraphicnamed commands also behave normally, which means that they will try to include the generated graphic files – which is exactly what you want.

Suppose that you wish to send your survey to a journal that does not have PGF installed. In this case, you now have all the necessary external graphics, but you still need PGF to automatically include them instead of the executing the picture code! One way to solve this problem is to simply delete all of the PGF or Ti*k*Z code from your survey.tex and instead insert appropriate \includegraphics commands "by hand." However, there is a better way: You input the file pgfexternal.tex.

File pgfexternal.tex

This file defines the command \beginpgfgraphicnamed and causes it to have the following effect: It includes the graphic file given as a parameter to it and then gobbles everything up to \endpgfgraphicnamed.

Since \beginpgfgraphicnamed does not do macro expansion as it searches for \endpgfgraphicnamed, it is not necessary to actually include the packages necessary for *creating* the graphics. So the idea is that you comment out things like \usepackage{tikz} and instead say \input pgfexternal.tex.

Indeed, the contents of this file is simply the following line:

```
\long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{\includegraphics{#1}}
```

Instead of \input pgfexternal.tex you could also include this line in your main file.

As a final remark, note that the baseline option does not work together with pictures written to an external graphic file. The simple reason is that there is no way to store this baseline information is an external graphic file.

54.5 A Complete Example

Let us now have a look at a simple, but complete example. We start out with a normal file called `survey.tex` that has the following contents:

```
% This is the file survey.tex
\documentclass{article}

\usepackage{graphics}
\usepackage{tikz}

\begin{document}
In the following figure, we see a circle:
\begin{tikzpicture}
  \fill (0,0) circle (10pt);
\end{tikzpicture}

By comparison, in this figure we see a rectangle:
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\end{document}
```

Now our editor tells us that the publisher will need all figures to be provided in separate PostScript or `.pdf`-files. For this, we enclose all figures in `...graphicnamed`-pairs and we add a call to the `\pgfrealjobname` macro:

```
% This is the file survey.tex
\documentclass{article}

\usepackage{graphics}
\usepackage{tikz}
\pgfrealjobname{survey}

\begin{document}
In the following figure, we see a circle:
\beginpgfgraphicnamed{survey-f1}
\begin{tikzpicture}
  \fill (0,0) circle (10pt);
\end{tikzpicture}
\endpgfgraphicnamed

By comparison, in this figure we see a rectangle:
\beginpgfgraphicnamed{survey-f2}
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\endpgfgraphicnamed
\end{document}
```

After these changes, typesetting the file will still yield the same output as it did before – after all, we have not yet created any external graphics.

To create the external graphics, we run `pdflatex` twice, once for each graphic:

```
bash> pdflatex --jobname=survey-f1 survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
...
) [1] (./survey-f1.aux) )
Output written on survey-f1.pdf (1 page, 1016 bytes).
Transcript written on survey-f1.log.
```

```
bash> pdflatex --jobname=survey-f2 survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
...
(./survey-f2.aux) )
Output written on survey-f2.pdf (1 page, 1002 bytes).
Transcript written on survey-f2.log.
```

We can now send the two generated graphics (`survey-f1.pdf` and `survey-f2.pdf`) to the editor. However, the publisher cannot use our `survey.tex` file, yet. The reason is that it contains the command `\usepackage{tikz}` and they do not have PGF installed.

Thus, we modify the main file `survey.tex` as follows:

```
% This is the file survey.tex
\documentclass{article}

\usepackage{graphics}
\input pgfexternal.tex
% \usepackage{tikz}
% \pgfrealjobname{survey}

\begin{document}
In the following figure, we see a circle:
\beginpgfgraphicnamed{survey-f1}
\begin{tikzpicture}
  \fill (0,0) circle (10pt);
\end{tikzpicture}
\endpgfgraphicnamed

By comparison, in this figure we see a rectangle:
\beginpgfgraphicnamed{survey-f2}
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\endpgfgraphicnamed
\end{document}
```

If we now run pdfLaTeX, then, indeed, PGF is no longer needed:

```
bash> pdflatex survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
Babel <v3.8h> and hyphenation patterns for english, ..., loaded.
(/usr/local/gwTeX/texmf.texlive/tex/latex/base/article.cls
Document Class: article 2005/09/16 v1.4f Standard LaTeX document class
(/usr/local/gwTeX/texmf.texlive/tex/latex/base/size10.clo))
(/usr/local/gwTeX/texmf.texlive/tex/latex/graphics/graphics.sty
(/usr/local/gwTeX/texmf.texlive/tex/latex/graphics/trig.sty)
(/usr/local/gwTeX/texmf.texlive/tex/latex/config/graphics.cfg)
(/usr/local/gwTeX/texmf.texlive/tex/latex/pdftex-def/pdftex.def))
(/Users/tantau/Library/texmf/tex/generic/pgf/generic/pgf/utilities/pgfexternal.
tex) (./survey.aux)
(/usr/local/gwTeX/texmf.texlive/tex/context/base/supp-pdf.tex
[Loading MPS to PDF converter (version 2006.09.02).]
) <survey-f1.pdf, id=1, 23.33318pt x 19.99973pt> <use survey-f1.pdf>
<survey-f2.pdf, id=2, 13.33382pt x 10.00037pt> <use survey-f2.pdf> [1{/Users/ta
ntau/Library/texmf/fonts/map/pdftex/updmap/pdftex.map} <./survey-f1.pdf> <./sur
vey-f2.pdf>] (./survey.aux) )</usr/local/gwTeX/texmf.texlive/fonts/type1/bluesk
y/cm/cmr10.pfb>
Output written on survey.pdf (1 page, 10006 bytes).
Transcript written on survey.log.
```

To our editor, we send the following files:

- The last `survey.tex` shown above.
- The graphic file `survey-f1.pdf`.
- The graphic file `survey-f2.pdf`.
- The file `pgfexternal.tex`, whose contents is simply

  ```
  \long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{\includegraphics{#1}}
  ```

  (Alternatively, we can also directly add this line to our `survey.tex` file).

55 Declaring and Using Shadings

55.1 Overview

A shading is an area in which the color changes smoothly between different colors. Similarly to an image, a shading must first be declared before it can be used. Also similarly to an image, a shading is put into a TEX-box. Hence, in order to include a shading in a `{pgfpicture}`, you have to use `\pgftext` around it.

There are three kinds of shadings: horizontal, vertical, and radial shadings. However, you can rotate and clip shadings like any other graphics object, which allows you to create more complicated shadings. Horizontal shadings could be created by rotating a vertical shading by 90 degrees, but explicit commands for creating both horizontal and vertical shadings are included for convenience.

Once you have declared a shading, you can insert it into text using the command `\pgfuseshading`. This command cannot be used directly in a `{pgfpicture}`, you have to put a `\pgftext` around it. The second command for using shadings, `\pgfshadepath`, on the other hand, can only be used inside `{pgfpicture}` environments. It will "fill" the current path with the shading.

A horizontal shading is a horizontal bar of a certain height whose color changes smoothly. You must at least specify the colors at the left and at the right end of the bar, but you can also add color specifications for points in between. For example, suppose you which to create a bar that is red at the left end, green in the middle, and blue at the end. Suppose you would like the bar to be 4cm long. This could be specified as follows:

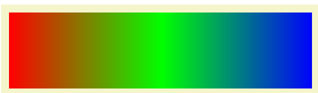
```
rgb(0cm)=(1,0,0); rgb(2cm)=(0,1,0); rgb(4cm)=(0,0,1)
```

This line means that at 0cm (the left end) of the bar, the color should be red, which has red-green-blue (rgb) components (1,0,0). At 2cm, the bar should be green, and at 4cm it should be blue. Instead of `rgb`, you can currently also specify `gray` as color model, in which case only one value is needed, or `color`, in which case you must provide the name of a color in parentheses. In a color specification the individual specifications must be separated using a semicolon, which may be followed by a whitespace (like a space or a newline). Individual specifications must be given in increasing order.

55.2 Declaring Shadings

`\pgfdeclarehorizontalshading[`⟨*color list*⟩`]{`⟨*shading name*⟩`}{`⟨*shading height*⟩`}{`⟨*color specification*⟩`}`

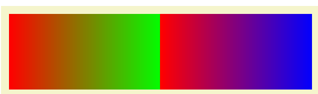
Declares a horizontal shading named ⟨*shading name*⟩ of the specified ⟨*height*⟩ with the specified colors. The length of the bar is deduced automatically from the maximum dimension in the specification.



```
\pgfdeclarehorizontalshading{myshadingA}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(green); color(4cm)=(blue)}
\pgfuseshading{myshadingA}
```

The effect of the ⟨*color list*⟩, which is a comma-separated list of colors, is the following: Normally, when this list is empty, once a shading has been declared, it becomes "frozen." This means that even if you change a color that was used in the declaration of the shading later on, the shading will not change. By specifying a ⟨*color list*⟩ you can specify that the shading should be recalculated whenever one of the colors listed in the list changes (this includes effects like color mixins). Thus, when you specify a ⟨*color list*⟩, whenever the shading is used, PGF first converts the colors in the list to RGB triples using the current values of the colors and taking any mixins and blends into account. If the resulting RGB triples have not yet been used, a new shading is internally created and used. Note that if the option ⟨*color list*⟩ is used, then no shading is created until the first use of `\pgfuseshading`. In particular, the colors mentioned in the shading need not be defined when the declaration is given.

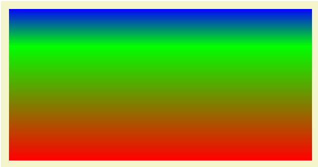
When a shading is recalculated because of a change in the colors mentioned in ⟨*color list*⟩, the complete shading is recalculated. Thus even colors not mentioned in the list will be used with their current values, not with the values they had upon declaration.



```
\pgfdeclarehorizontalshading[mycolor]{myshadingB}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(mycolor)}
\colorlet{mycolor}{green}
\pgfuseshading{myshadingB}
\colorlet{mycolor}{blue}
\pgfuseshading{myshadingB}
```

`\pgfdeclareverticalshading`[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*shading width*⟩}{⟨*color specification*⟩}

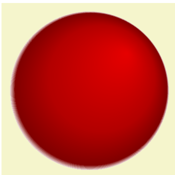
Declares a vertical shading named ⟨*shading name*⟩ of the specified ⟨*width*⟩. The height of the bar is deduced automatically. The effect of ⟨*color list*⟩ is the same as for horizontal shadings.



```
\pgfdeclareverticalshading{myshadingC}
  {4cm}{rgb(0cm)=(1,0,0); rgb(1.5cm)=(0,1,0); rgb(2cm)=(0,0,1)}
\pgfuseshading{myshadingC}
```

`\pgfdeclareradialshading`[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*center point*⟩}{⟨*color specification*⟩}

Declares an radial shading. A radial shading is a circle whose inner color changes as specified by the color specification. Assuming that the center of the shading is at the origin, the color of the center will be the color specified for 0cm and the color of the border of the circle will be the color for the maximum dimension given in the ⟨*color specified*⟩. This maximum will also be the radius of the circle. If the ⟨*center point*⟩ is not at the origin, the whole shading inside the circle (whose size remains exactly the same) will be distorted such that the given center now has the color specified for 0cm. The effect of ⟨*color list*⟩ is the same as for horizontal shadings.

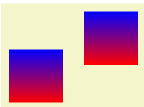


```
\pgfdeclareradialshading{sphere}{\pgfpoint{0.5cm}{0.5cm}}%
  {rgb(0cm)=(0.9,0,0);
   rgb(0.7cm)=(0.7,0,0);
   rgb(1cm)=(0.5,0,0);
   rgb(1.05cm)=(1,1,1)}
\pgfuseshading{sphere}
```

55.3 Using Shadings

`\pgfuseshading`{⟨*shading name*⟩}

Inserts a previously declared shading into the text. If you wish to use it in a `pgfpicture` environment, you should put a `\pgfbox` around it.



```
\begin{pgfpicture}
  \pgfdeclareverticalshading{myshadingD}
    {20pt}{color(0pt)=(red); color(20pt)=(blue)}
  \pgftext[at=\pgfpoint{1cm}{0cm}]  {\pgfuseshading{myshadingD}}
  \pgftext[at=\pgfpoint{2cm}{0.5cm}]{\pgfuseshading{myshadingD}}
\end{pgfpicture}
```

`\pgfshadepath`{⟨*shading name*⟩}{⟨*angle*⟩}

This command must be used inside a `{pgfpicture}` environment. The effect is a bit complex, so let us go over it step by step.

First, PGF will setup a local scope.

Second, it uses the current path to clip everything inside this scope. However, the current path is once more available after the scope, so it can be used, for example, to stroke it.

Now, the ⟨*shading name*⟩ should be a shading whose width and height are 100 bp, that is, 100 big points. PGF has a look at the bounding box of the current path. This bounding box is computed automatically when a path is computed; however, it can sometimes be (quite a bit) too large, especially when complicated curves are involved.

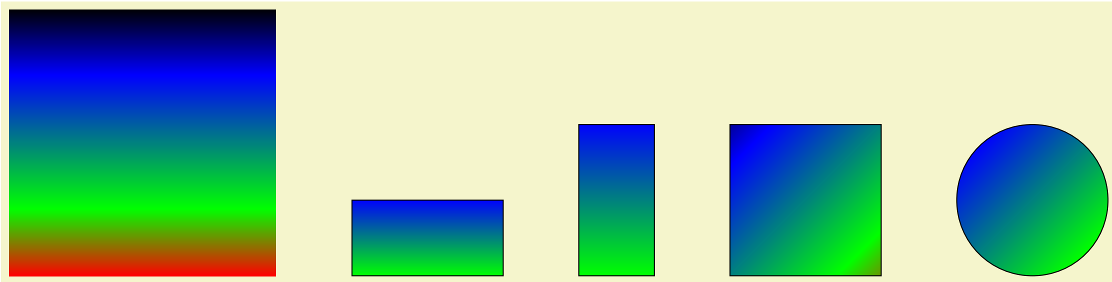
Inside the scope, the low-level transformation matrix is modified. The center of the shading is translated (moved) such that it lies on the center of the bounding box of the path. The low-level coordinate system is also scaled such that the shading "covers" the shading (the details are a bit more complex, see below). Then, the coordinate system is rotated by ⟨*angle*⟩. Finally, if the macro `\pgfsetadditionalshadetransform` has been used, an additional transformation is applied.

After everything has been set up, the shading is inserted. Due to the transformations and clippings, the effect will be that the shading seems to "fill" the path.

If both the path and the shadings were always rectangles and if rotation were never involved, it would be easy to scale shadings such they always cover the path. However, when a vertical shading is rotated, it must obviously be "magnified" so that it still covers the path. Things get worse when the path is not a rectangle itself.

For these reasons, things work slightly differently "in reality." The shading is scaled and translated such that the the point (50bp, 50bp), which is the middle of the shading, is at the middle of the path and such that the the point (25bp, 25bp) is at the lower left corner of the path and that (75bp, 75bp) is at upper right corner.

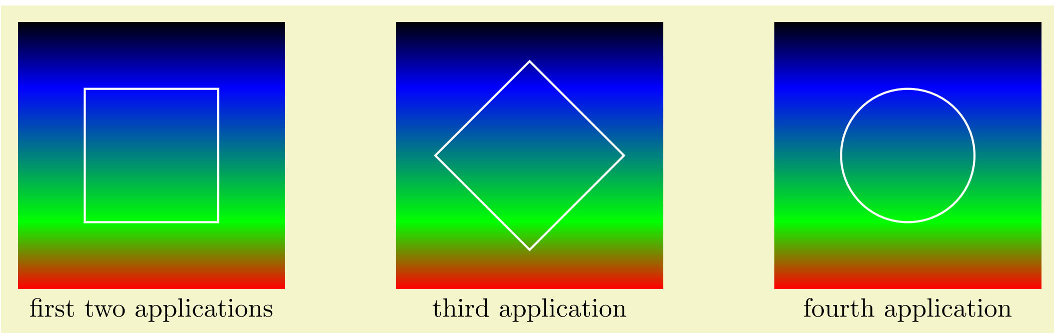
In other words, only the center quarter of the shading will actually "survive the clipping" if the path is a rectangle. If the path is not a rectangle, but, say, a circle, even less is seen of the shading. Here is an example that demonstrates this effect:



```
\pgfdeclareverticalshading{myshadingE}{100bp}
 {color(0bp)=(red); color(25bp)=(green);  color(75bp)=(blue);  color(100bp)=(black)}
\pgfuseshading{myshadingE}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{1cm}{2cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{5cm}{0cm}}{\pgfpoint{2cm}{2cm}}
  \pgfshadepath{myshadingE}{45}
  \pgfusepath{stroke}
  \pgfpathcircle{\pgfpoint{9cm}{1cm}}{1cm}
  \pgfshadepath{myshadingE}{45}
  \pgfusepath{stroke}
\end{pgfpicture}
```

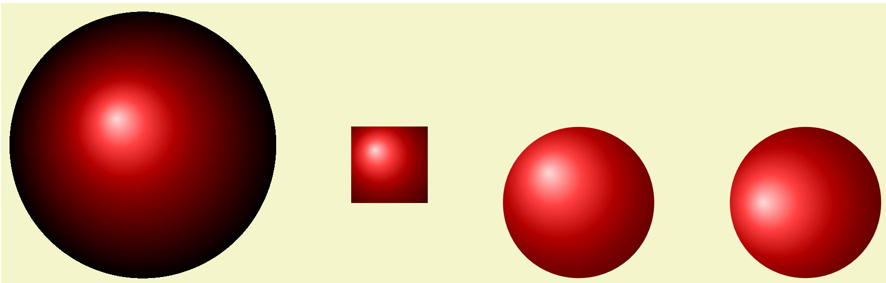
As can be seen above in the last case, the "hidden" part of the shading actually *can* become visible if the shading is rotated. The reason is that it is scaled as if no rotation took place, then the rotation is done.

The following graphics show which part of the shading are actually shown:



```
\pgfdeclareverticalshading{myshadingF}{100bp}
 {color(0bp)=(red); color(25bp)=(green);  color(75bp)=(blue);  color(100bp)=(black)}
\begin{tikzpicture}
  \draw (50bp,50bp) node {\pgfuseshading{myshadingF}};
  \draw[white,thick] (25bp,25bp) rectangle (75bp,75bp);
  \draw (50bp,0bp) node[below] {first two applications};

  \begin{scope}[xshift=5cm]
    \draw (50bp,50bp) node{\pgfuseshading{myshadingF}};
    \draw[rotate around={45:(50bp,50bp)},white,thick] (25bp,25bp) rectangle (75bp,75bp);
    \draw (50bp,0bp) node[below] {third application};
  \end{scope}

  \begin{scope}[xshift=10cm]
    \draw (50bp,50bp) node{\pgfuseshading{myshadingF}};
    \draw[white,thick] (50bp,50bp) circle (25bp);
    \draw (50bp,0bp) node[below] {fourth application};
  \end{scope}
\end{tikzpicture}
```

An advantage of this approach is that when you rotate a radial shading, no distortion is introduced:



```
\pgfdeclareradialshading{ballshading}{\pgfpoint{-10bp}{10bp}}
 {color(0bp)=(red!15!white); color(9bp)=(red!75!white);
 color(18bp)=(red!70!black); color(25bp)=(red!50!black); color(50bp)=(black)}
\pgfuseshading{ballshading}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{3cm}{0cm}}{1cm}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{6cm}{0cm}}{1cm}
  \pgfshadepath{ballshading}{45}
  \pgfusepath{}
\end{pgfpicture}
```

If you specify a rotation of 90° and if the path is not a square, but an elongated rectangle, the "desired" effect results: The shading will exactly vary between the colors at the 25bp and 75bp boundaries. Here is an example:

```
\pgfdeclareverticalshading{myshadingG}{100bp}
 {color(0bp)=(red); color(25bp)=(green);  color(75bp)=(blue);  color(100bp)=(black)}
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingG}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingG}{90}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{6cm}{0cm}}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingG}{45}
  \pgfusepath{stroke}
\end{pgfpicture}
```

As a final example, let us define a "rainbow spectrum" shading for use with Ti*k*Z.


```
\pgfdeclareverticalshading{rainbow}{100bp}
 {color(0bp)=(red); color(25bp)=(red); color(35bp)=(yellow);
  color(45bp)=(green); color(55bp)=(cyan); color(65bp)=(blue);
  color(75bp)=(violet); color(100bp)=(violet)}
\begin{tikzpicture}[shading=rainbow]
  \shade (0,0) rectangle node[white] {\textsc{pride}} (2,1);
  \shade[shading angle=90] (3,0) rectangle +(1,2);
\end{tikzpicture}
```

Note that rainbow shadings are *way* to colorful in almost all applications.

`\pgfsetadditionalshadetransform{`⟨*transformation*⟩`}`

This command allows you to specify an additional transformation that should be applied to shadings when the `\pgfshadepath` command is used. The ⟨*transformation*⟩ should be transformation code like `\pgftransformrotate{20}`.

56 Creating Plots

This section describes the `pgfbaseplot` package.

```
\usepackage{pgfbaseplot} % LaTeX
\input pgfbaseplot.tex   % plain TeX
\usemodule[pgfbaseplot]  % ConTeXt
```

This package provides a set of commands that are intended to make it reasonably easy to plot functions using PGF. It is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

56.1 Overview

There are different reasons for using PGF for creating plots rather than some more powerful program such as GNUPLOT or MATHEMATICA, as discussed in Section 16.1. So, let us assume that – for whatever reason – you wish to use PGF for generating a plot.

PGF (conceptually) uses a two-stage process for generating plots. First, a *plot stream* must be produced. This stream consists (more or less) of a large number of coordinates. Second a *plot handler* is applied to the stream. A plot handler "does something" with the stream. The standard handler will issue line-to operations to the coordinates in the stream. However, a handler might also try to issue appropriate curve-to operations in order to smooth the curve. A handler may even do something else entirely, like writing each coordinate to another stream, thereby duplicating the original stream.

Both for the creation of streams and the handling of streams different sets of commands exist. The commands for creating streams start with `\pgfplotstream`, the commands for setting the handler start with `\pgfplothandler`.

56.2 Generating Plot Streams

56.2.1 Basic Building Blocks of Plot Streams

A *plot stream* is a (long) sequence of the following three commands:

1. `\pgfplotstreamstart`,
2. `\pgfplotstreampoint`, and
3. `\pgfplotstreamend`.

Between calls of these commands arbitrary other code may be called. Obviously, the stream should start with the first command and end with the last command. Here is an example of a plot stream:

```
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\newdimen\mydim
\mydim=2cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\advance \mydim by 3cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\pgfplotstreamend
```

`\pgfplotstreamstart`

This command signals that a plot stream starts. The effect of this command is to call the internal command `\pgf@plotstreamstart`, which is set by the current plot handler to do whatever needs to be done at the beginning of the plot.

`\pgfplotstreampoint{⟨point⟩}`

This command adds a ⟨*point*⟩ to the current plot stream. The effect of this command is to call the internal command `\pgf@plotstreampoint`, which is also set by the current plot handler. This command should now "handle" the point in some sensible way. For example, a line-to command might be issued for the point.

`\pgfplotstreamend`

This command signals that a plot stream ends. It calls `\pgf@plotstreamend`, which should now do any necessary "cleanup."

Note that plot streams are not buffered, that is, the different points are handled immediately. However, using the recording handler, it is possible to record a stream.

56.2.2 Commands That Generate Plot Streams

Plot streams can be created "by hand" as in the earlier example. However, most of the time the coordinates will be produced internally by some command. For example, the `\pgfplotxyfile` reads a file and converts it into a plot stream.

`\pgfplotxyfile{`⟨*filename*⟩`}`

This command will try to open the file ⟨*filename*⟩. If this succeeds, it will convert the file contents into a plot stream as follows: A `\pgfplotstreamstart` is issued. Then, each nonempty line of the file should start with two numbers separated by a space, such as `0.1 1` or `100 -.3`. Anything following the numbers is ignored.

Each pair ⟨x⟩ and ⟨y⟩ of numbers is converted into one plot stream point in the xy-coordinate system. Thus, a line like

```
2 -5 some text
```

is turned into

```
\pgfplotstreampoint{\pgfpointxy{2}{-5}}
```

The two characters `%` and `#` are also allowed in a file and they are both treated as comment characters. Thus, a line starting with either of them is empty and, hence, ignored.

When the file has been read completely, `\pgfplotstreamend` is called.

`\pgfplotxyzfile{`⟨*filename*⟩`}`

This command works like `\pgfplotxyfile`, only *three* numbers are expected on each non-empty line. They are converted into points in the xyz-coordinate system. Consider, the following file:

```
% Some comments
# more comments
2 -5  1 first entry
2 -.2 2 second entry
2 -5  2 third entry
```

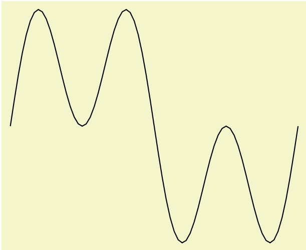
It is turned into the following stream:

```
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{1}}
\pgfplotstreampoint{\pgfpointxyz{2}{-.2}{2}}
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{2}}
\pgfplotstreamend
```

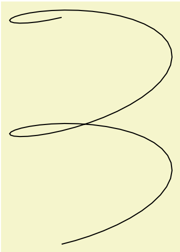
Currently, there is no command that can decide automatically whether the xy-coordinate system should be used or whether the xyz-system should be used. However, it would not be terribly difficult to write a "smart file reader" that parses coordinate files a bit more intelligently.

`\pgfplotfunction{`⟨*variable*⟩`}{`⟨*sample list*⟩`}{`⟨*point*⟩`}`

This command will produce coordinates by iterating the ⟨*variable*⟩ over all values in ⟨*sample list*⟩, which should be a list in the `\foreach` syntax. For each value of ⟨*variable*⟩, the ⟨*point*⟩ is evaluated and the resulting coordinate is inserted into the plot stream.



```
\begin{tikzpicture}[x=3.8cm/360]
  \pgfplothandlerlineto
  \pgfplotfunction{\x}{0,5,...,360}{\pgfpointxy{\x}{sin(\x)+sin(3*\x)}}
  \pgfusepath{stroke}
\end{tikzpicture}
```



```
\begin{tikzpicture}[y=3cm/360]
  \pgfplothandlerlineto
  \pgfplotfunction{\y}{0,5,...,360}{\pgfpointxyz{sin(2*\y)}{\y}{cos(2*\y)}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

Be warnded that if the expressions that need to evaluated for each point are complex, then this command can be very slow.

`\pgfplotgnuplot`[⟨*prefix*⟩]{⟨*function*⟩}

This command will "try" to call the GNUPLOT program to generate the coordinates of the ⟨*function*⟩. In detail, the following happens:

This command works with two files: ⟨*prefix*⟩`.gnuplot` and ⟨*prefix*⟩`.table`. If the optional argument ⟨*prefix*⟩ is not given, it is set to `\jobname`.

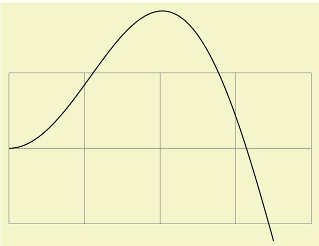
Let us start with the situation where none of these files exists. Then PGF will first generate the file ⟨*prefix*⟩`.gnuplot`. In this file it writes

```
set terminal table; set output "#1.table"; set format "%.5f"
```

where `#1` is replaced by ⟨*prefix*⟩. Then, in a second line, it writes the text ⟨*function*⟩.

Next, PGF will try to invoke the program `gnuplot` with the argument ⟨*prefix*⟩`.gnuplot`. This call may or may not succeed, depending on whether the `\write18` mechanism (also known as shell escape) is switched on and whether the `gnuplot` program is available.

Assuming that the call succeeded, the next step is to invoke `\pgfplotxyfile` on the file ⟨*prefix*⟩`.table`; which is exactly the file that has just been created by `gnuplot`.



```
\begin{tikzpicture}
  \draw[help lines] (0,-1) grid (4,1);
  \pgfplothandlerlineto
  \pgfplotgnuplot[plots/pgfplotgnuplot-example]{plot [x=0:3.5] x*sin(x)}
  \pgfusepath{stroke}
\end{tikzpicture}
```

The more difficult situation arises when the `.gnuplot` file exists, which will be the case on the second run of TeX on the TeX file. In this case PGF will read this file and check whether it contains exactly what PGF "would have written" into this file. If this is not the case, the file contents is overwritten with what "should be there" and, as above, `gnuplot` is invoked to generate a new `.table` file. However, if the file contents is "as expected," the external `gnuplot` program is *not* called. Instead, the ⟨*prefix*⟩`.table` file is immediately read.

As explained in Section 16.6, the net effect of the above mechanism is that `gnuplot` is called as little as possible and that when you pass along the `.gnuplot` and `.table` files with your `.tex` file to someone else, that person can TeX the `.tex` file without having `gnuplot` installed and without having the `\write18` mechanism switched on.

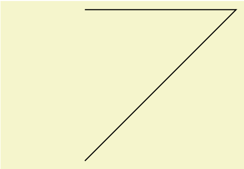
56.3 Plot Handlers

A *plot handler* prescribes what "should be done" with a plot stream. You must set the plot handler before the stream starts. The following commands install the most basic plot handlers; more plot handlers are defined in the file `pgflibraryplothandlers`, which is documented in Section 28.

All plot handlers work by setting redefining the following three macros: `\pgf@plotstreamstart`, `\pgf@plotstreampoint`, and `\pgf@plotstreamend`.

`\pgfplothandlerlineto`

This handler will issue a `\pgfpathlineto` command for each point of the plot, *except* possibly for the first. What happens with the first point can be specified using the two commands described below.



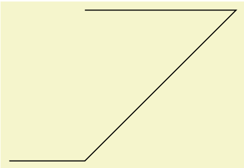
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetmovetofirstplotpoint`

Specifies that the line-to plot handler (and also some other plot handlers) should issue a move-to command for the first point of the plot instead of a line-to. This will start a new part of the current path, which is not always, but often, desirable. This is the default.

`\pgfsetlinetofirstplotpoint`

Specifies that plot handlers should issue a line-to command for the first point of the plot.



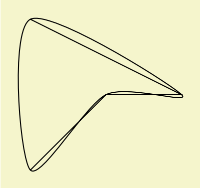
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfsetlinetofirstplotpoint
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfplothandlerdiscard`

This handler will simply throw away the stream.

`\pgfplothandlerrecord{`⟨*macro*⟩`}`

When this handler is installed, each time a plot stream command is called, this command will be appended to ⟨*macros*⟩. Thus, at the end of the stream, ⟨*macro*⟩ will contain all the commands that were issued on the stream. You can then install another handler and invoke ⟨*macro*⟩ to "replay" the stream (possibly many times).



```
\begin{pgfpicture}
  \pgfplothandlerrecord{\mystream}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfplothandlerlineto
  \mystream
  \pgfplothandlerclosedcurve
  \mystream
  \pgfusepath{stroke}
\end{pgfpicture}
```

57 Layered Graphics

```
\usepackage{pgfbaselayers} % LaTeX
\input pgfbaselayers.tex   % plain TeX
\usemodule[pgfbaselayers]  % ConTeXt
```

This package provides a commands and environments for composing a picture from multiple layers. The package is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

57.1 Overview

PGF provides a layering mechanism for composing graphics from multiple layers. (This mechanism is not be confused with the conceptual "software layers" the PGF system is composed of.) Layers are often used in graphic programs. The idea is that you can draw on the different layers in any order. So you might start drawing something on the "background" layer, then something on the "foreground" layer, then something on the "middle" layer, and then something on the background layer once more, and so on. At the end, no matter in which ordering you drew on the different layers, the layers are "stacked on top of each other" in a fixed ordering to produce the final picture. Thus, anything drawn on the middle layer would come on top of everything of the background layer.

Normally, you do not need to use different layers since you will have little trouble "ordering" your graphic commands in such a way that layers are superfluous. However, in certain situations you only "know" what you should draw behind something else after the "something else" has been drawn.

For example, suppose you wish to draw a yellow background behind your picture. The background should be as large as the bounding box of the picture, plus a little border. If you know the size of the bounding box of the picture at its beginning, this is easy to accomplish. However, in general this is not the case and you need to create a "background" layer in addition to the standard "main" layer. Then, at the end of the picture, when the bounding box has been established, you can add a rectangle of the appropriate size to the picture.

57.2 Declaring Layers

In PGF layers are referenced using names. The standard layer, which is a bit special in certain ways, is called `main`. If nothing else is specified, all graphic commands are added to the `main` layer. You can declare a new layer using the following command:

`\pgfdeclarelayer{`⟨*name*⟩`}`

This command declares a layer named ⟨*name*⟩ for later use. Mainly, this will setup some internal bookkeeping.

The next step toward using a layer is to tell PGF which layers will be part of the actual picture and which will be their ordering. Thus, it is possible to have more layers declared than are actually used.

`\pgfsetlayers{`⟨*layer list*⟩`}`

This command, which should be used *outside* a `{pgfpicture}` environment, tells PGF which layers will be used in pictures. They are stacked on top of each other in the order given. The layer `main` should always be part of the list. Here is an example:

```
\pgfdeclarelayer{background}
\pgfdeclarelayer{foreground}
\pgfsetlayers{background,main,foreground}
```

57.3 Using Layers

Once the layers of your picture have been declared, you can start to "fill" them. As said before, all graphics commands are normally added to the `main` layer. Using the `{pgfonlayer}` environment, you can tell PGF that certain commands should, instead, be added to the given layer.

`\begin{pgfonlayer}{`⟨*layer name*⟩`}`
⟨*environment contents*⟩

\end{pgfonlayer}

The whole ⟨*environment contents*⟩ is added to the layer with the name ⟨*layer name*⟩. This environment can be used anywhere inside a picture. Thus, even if it is used inside a {pgfscope} or a TeX group, the contents will still be added to the "whole" picture. Using this environment multiple times inside the same picture will cause the ⟨*environment contents*⟩ to accumulate.

Note: You can *not* add anything to the main layer using this environment. The only way to add anything to the main layer is to give graphic commands outside all {pgfonlayer} environments.


```
\pgfdeclarelayer{background layer}
\pgfdeclarelayer{foreground layer}
\pgfsetlayers{background layer,main,foreground layer}
\begin{tikzpicture}
  % On main layer:
  \fill[blue] (0,0) circle (1cm);

  \begin{pgfonlayer}{background layer}
    \fill[yellow] (-1,-1) rectangle (1,1);
  \end{pgfonlayer}

  \begin{pgfonlayer}{foreground layer}
    \node[white] {foreground};
  \end{pgfonlayer}

  \begin{pgfonlayer}{background layer}
    \fill[black] (-.8,-.8) rectangle (.8,.8);
  \end{pgfonlayer}

  % On main layer again:
  \fill[blue!50] (-.5,-1) rectangle (.5,1);
\end{tikzpicture}
```

\pgfonlayer{⟨*layer name*⟩}
  ⟨*environment contents*⟩
\endpgfonlayer

This is the plain TeX version of the environment.

\startpgfonlayer{⟨*layer name*⟩}
  ⟨*environment contents*⟩
\stoppgfonlayer

This is the ConTeXt version of the environment.

58 Quick Commands

This section explains the "quick" commands of PGF. These commands are executed more quickly than the normal commands of PGF, but offer less functionality. You should use these commands only if you either have a very large number of commands that need to be processed or if you expect your commands to be executed very often.

58.1 Quick Coordiante Commands

`\pgfqpoint{`$\langle x\rangle$`}{`$\langle y\rangle$`}`

This command does the same as `\pgfpoint`, but $\langle x\rangle$ and $\langle y\rangle$ must be simple dimensions like `1pt` or `1cm`. Things like `2ex` or `2cm+1pt` are not allowed.

58.2 Quick Path Construction Commands

The difference between the quick and the normal path commands is that the quick path commands

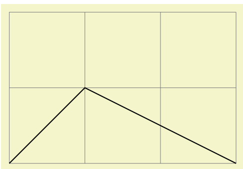
- do not keep track of the bounding boxes,
- do not allow you to arc corners,
- do not apply coordinate transformations.

However, they do use the soft-path subsystem (see Section 61 for details), which allows you to mix quick and normal path commands arbitrarily.

All quick path construction commands start with `\pgfpathq`.

`\pgfpathqmoveto{`$\langle x\ dimension\rangle$`}{`$\langle y\ dimension\rangle$`}`

Either starts a path or starts a new part of a path at the coordinate ($\langle x\ dimension\rangle$, $\langle y\ dimension\rangle$). The coordinate is *not* transformed by the current coordinate transformation matrix. However, any low-level transformations apply.



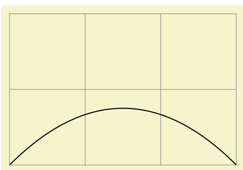
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformxshift{1cm}
  \pgfpathqmoveto{0pt}{0pt} % not transformed
  \pgfpathqlineto{1cm}{1cm} % not transformed
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfpathqlineto{`$\langle x\ dimension\rangle$`}{`$\langle y\ dimension\rangle$`}`

The quick version of the line-to operation.

`\pgfpathqcurveto{`$\langle s_x^1\rangle$`}{`$\langle s_y^1\rangle$`}{`$\langle s_x^2\rangle$`}{`$\langle s_y^2\rangle$`}{`$\langle t_x\rangle$`}{`$\langle t_y\rangle$`}`

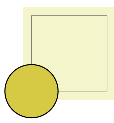
The quick version of the curve-to operation. The first support point is (s_x^1, s_y^1), the second support point is (s_x^2, s_y^2), and the target is (t_x, t_y).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathqmoveto{0pt}{0pt}
  \pgfpathqcurveto{1cm}{1cm}{2cm}{1cm}{3cm}{0cm}
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfpathqcircle{`$\langle radius\rangle$`}`

Adds a radius around the origin of the given $\langle radius\rangle$. This command is orders of magnitude faster than `\pgfcircle{\pgfpointorigin}{`$\langle radius\rangle$`}`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (1,1);
  \pgfpathqcircle{10pt}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{stroke,fill}
\end{tikzpicture}
```

58.3 Quick Path Usage Commands

The quick path usage commands perform similar tasks as `\pgfusepath`, but they

- do not add arrows,
- do not modify the path in any way, in particular,
- ends are not shortened,
- corners are not replaced by arcs.

Note that you *have to* use the quick versions in the code of arrow tip definitions since, inside these definition, you obviously do not want arrows to be drawn.

`\pgfusepathqstroke`

Strokes the path without further ado. No arrows are drawn, no corners are arced.

```
\begin{pgfpicture}
  \pgfpathqcircle{5pt}
  \pgfusepathqstroke
\end{pgfpicture}
```

`\pgfusepathqfill`

Fills the path without further ado.

`\pgfusepathqfillstroke`

Fills and then strokes the path without further ado.

`\pgfusepathqclip`

Clips all subsequent drawings against the current path. The path is not processed.

58.4 Quick Text Box Commands

`\pgfqbox{`⟨*box number*⟩`}`

This command inserts a TEX box into a `{pgfpicture}` by "escaping" to TEX, inserting the box number ⟨*box number*⟩ at the origin, and then returning to the typesetting the picture.

`\pgfqboxsynced{`⟨*box number*⟩`}`

This command works similarly to the `\pgfqbox` command. However, before inserting the text in ⟨*box number*⟩, the current coordinate transformation matrix is applied to the current canvas transformation matrix (is it "synced" with this matrix, hence the name).

Thus, this command basically has the same effect as if you first called `\pgflowlevelsynccm` followed by `\pgfqbox`. However, this command will use `\hskip` and `\raise` commands for the "translational part" of the coordinate transformation matrix, instead of adding the translational part to the current canvas transformation matrix directly. Both methods have the same effect (box ⟨*box number*⟩ is translated where it should), but the method used by `\pgfqboxsynced` ensures that hyperlinks are placed correctly. Note that scaling and rotation will not (cannot, even) apply to hyperlinks.

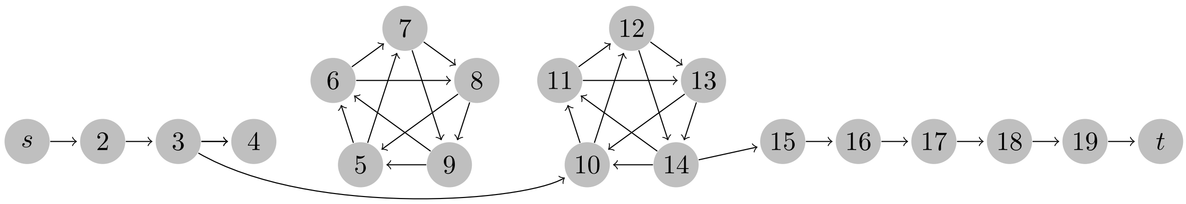
Part VIII
The System Layer

by Till Tantau

This part describes the low-level interface of PGF, called the *system layer*. This interface provides a complete abstraction of the internals of the underlying drivers.

Unless you intend to port PGF to another driver or unless you intend to write your own optimized frontend, you need not read this part.

In the following it is assumed that you are familiar with the basic workings of the `graphics` package and that you know what TeX-drivers are and how they work.




```
\begin{tikzpicture}[shorten >=1pt,->]
  \tikzstyle{vertex}=[circle,fill=black!25,minimum size=17pt,inner sep=0pt]

  \foreach \name/\x in {s/1, 2/2, 3/3, 4/4, 15/11, 16/12, 17/13, 18/14, 19/15, t/16}
    \node[vertex] (G-\name) at (\x,0) {$\name$};

  \foreach \name/\angle/\text in {P-1/234/5, P-2/162/6, P-3/90/7, P-4/18/8, P-5/-54/9}
    \node[vertex,xshift=6cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \name/\angle/\text in {Q-1/234/10, Q-2/162/11, Q-3/90/12, Q-4/18/13, Q-5/-54/14}
    \node[vertex,xshift=9cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \from/\to in {s/2,2/3,3/4,3/4,15/16,16/17,17/18,18/19,19/t}
    \draw (G-\from) -- (G-\to);

  \foreach \from/\to in {1/2,2/3,3/4,4/5,5/1,1/3,2/4,3/5,4/1,5/2}
    { \draw (P-\from) -- (P-\to); \draw (Q-\from) -- (Q-\to); }

  \draw (G-3) .. controls +(-30:2cm) and +(-150:1cm) .. (Q-1);
  \draw (Q-5) -- (G-15);
\end{tikzpicture}
```

59 Design of the System Layer

59.1 Driver Files

The PGF system layer mainly consists of a large number of commands starting with `\pgfsys@`. These commands will be called *system commands* in the following. The higher layers "interface" with the system layer by calling these commands. The higher layers should never use `\special` commands directly or even check whether `\pdfoutput` is defined. Instead, all drawing requests should be "channeled" through the system commands.

The system layer is loaded and setup by the following package:

```
\usepackage{pgfsys} % LaTeX
\input pgfsys.tex   % plain TeX
\usemodule[pgfsys]  % ConTeXt
```

This file provides "default implementations" of all system commands, but most simply produce a warning that they are not implemented. The actual implementations of the system commands for a particular driver like, say, `pdftex` reside in files called `pgfsys-xxxx.sty`, where `xxxx` is the driver name. These will be called *driver files* in the following.

When `pgfsys.sty` is loaded, it will try to determine which driver is used by loading `pgf.cfg`. This file should setup the macro `\pgfsysdriver` appropriately. The, `pgfsys.sty` will input the appropriate `pgfsys-`⟨*drivername*⟩`.sty`.

`\pgfsysdriver`

This macro should expand to the name of the driver to be used by `pgfsys`. The default from `pgf.cfg` is `pgfsys-\Gin@driver`. This is very likely to be correct if you are using LaTeX. For plain TeX, the macro will be set to `pgfsys-pdftex.def` if `pdftex` is used and to `pgfsys-dvips.def` otherwise.

File `pgf.cfg`

This file should setup the command `\pgfsysdriver` correctly. If `\pgfsysdriver` is already set to some value, the driver normally should not change it. Otherwise, it should make a "good guess" at which driver will be appropriate.

The currently supported backend drivers are discussed in Section 7.2.

59.2 Common Definition Files

Some drivers share many `\pgfsys@` commands. For the reason, files defining these "common" commands are available. These files are *not* usable alone.

File `pgfsys-common-postscript`

This file defines some `\pgfsys@` commands so that they produce appropriate PostScript code.

File `pgfsys-common-pdf`

This file defines some `\pgfsys@` commands so that they produce appropriate PDF code.

60 Commands of the System Layer

60.1 Beginning and Ending a Stream of System Commands

A "user" of the PGF system layer (like the basic layer or a frontend) will interface with the system layer by calling a stream of commands starting with `\pgfsys@`. From the system layer's point of view, these commands form a long stream. Between calls to the system layer, control goes back to the user.

The driver files implement system layer commands by inserting `\special` commands that implement the desired operation. For example, `\pgfsys@stroke` will be mapped to `\special{pdf: S}` by the driver file for `pdftex`.

For many drivers, when such a stream of specials starts, it is necessary to install an appropriate transformation and perhaps perform some more bureaucratic tasks. For this reason, every stream will start with a `\pgfsys@beginpicture` and will end with a corresponding ending command.

`\pgfsys@beginpicture`

Called at the beginning of a `{pgfpicture}`. This command should "setup things."

Most drivers will need to implement this command.

`\pgfsys@endpicture`

Called at the end of a pgfpicture.

Most drivers will need to implement this command.

`\pgfsys@typesetpicturebox{`⟨*box*⟩`}`

Called *after* a `{pgfpicture}` has been typeset. The picture will have been put in box ⟨*box*⟩. This command should insert the box into the normal text. The box ⟨*box*⟩ will still be a "raw" box that contains only the `\special`'s that make up the description of the picture. The job of this command is to resize and shift ⟨*box*⟩ according to the baseline shift and the size of the box.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@beginpurepicture`

This version of the `\pgfsys@beginpicture` picture command can be used for pictures that are guaranteed not to contain any escaped boxes (see below). In this case, a driver might provide a more compact version of the command.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@endpurepicture`

Called at the end of a "pure" `{pgfpicture}`.

This command has a default implementation and need not be implemented by a driver file.

Inside a stream it is sometimes necessary to "escape" back into normal typesetting mode; for example to insert some normal text, but with all of the current transformations and clippings being in force. For this escaping, the following command is used:

`\pgfsys@hbox{`⟨*box number*⟩`}`

Called to insert a (horizontal) TeX box inside a `{pgfpicture}`.

Most drivers will need to (re-)implement this command.

`\pgfsys@hboxsynced{`⟨*box number*⟩`}`

Called to insert a (horizontal) TeX box inside a `{pgfpicture}`, but with the current coordiante transformation matrix synced with the canvas transformation matrix.

This command should do the same as if you used `\pgflowlevelsynccm` followed by `\pgfsys@hbox`. However, the default implementation of this command will use a "TeX-translation" for the translation part of the transformation matrix. This will ensure that hyperlinks "survive" at least translations. On the other hand, a driver may choose to revert to a simpler implementation. This is done, for example, for the SVG implementation, where a TeX-translation makes no sense.

60.2 Path Construction System Commands

\pgfsys@moveto{⟨x⟩}{⟨y⟩}

This command is used to start a path at a specific point (x, y) or to move the current point of the current path to (x, y) without drawing anything upon stroking (the current path is "interrupted").

Both ⟨x⟩ and ⟨y⟩ are given as TeX dimensions. It is the driver's job to transform these to the coordinate system of the backend. Typically, this means converting the TeX dimension into a dimensionless multiple of $\frac{1}{72}$in. The function `\pgf@sys@bp` helps with this conversion.

Example: Draw a line from (10pt, 10pt) to the origin of the picture.

```
\pgfsys@moveto{10pt}{10pt}
\pgfsys@lineto{0pt}{0pt}
\pgfsys@stroke
```

This command is protocoled, see Section 62.

\pgfsys@lineto{⟨x⟩}{⟨y⟩}

Continue the current path to (x, y) with a straight line.

This command is protocoled, see Section 62.

\pgfsys@curveto{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}{⟨x_3⟩}{⟨y_3⟩}

Continue the current path to (x_3, y_3) with a Bézier curve that has the two control points (x_1, y_1) and (x_2, y_2).

Example: Draw a good approximation of a quarter circle:

```
\pgfsys@moveto{10pt}{0pt}
\pgfsys@curveto{10pt}{5.55pt}{5.55pt}{10pt}{0pt}{10pt}
\pgfsys@stroke
```

This command is protocoled, see Section 62.

\pgfsys@rect{⟨x⟩}{⟨y⟩}{⟨width⟩}{⟨height⟩}

Append a rectangle to the current path whose lower left corner is at (x, y) and whose width and height in big points are given by ⟨*width*⟩ and ⟨*height*⟩.

This command can be "mapped back" to `\pgfsys@moveto` and `\pgfsys@lineto` commands, but it is included since PDF has a special, quick version of this command.

This command is protocoled, see Section 62.

\pgfsys@closepath

Close the current path. This results in joining the current point of the path with the point specified by the last `\pgfsys@moveto` operation. Typically, this is preferable over using `\pgfsys@lineto` to the last point specified by a `\pgfsys@moveto`, since the line starting at this point and the line ending at this point will be smoothly joined by `\pgfsys@closepath`.

Example: Consider

```
\pgfsys@moveto{0pt}{0pt}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@closepath
\pgfsys@stroke
```

and

```
\pgfsys@moveto{0bp}{0bp}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@lineto{0bp}{0bp}
\pgfsys@stroke
```

The difference between the above will be that in the second triangle the corner at the origin will be wrong; it will just be the overlay of two lines going in different directions, not a sharp pointed corner.

This command is protocoled, see Section 62.

60.3 Canvas Transformation System Commands

`\pgfsys@transformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨e⟩}{⟨f⟩}`

Perform a concatenation of the canvas transformation matrix with the matrix given by the values ⟨*a*⟩ to ⟨*f*⟩, see the PDF or PostScript manual for details. The values ⟨*a*⟩ to ⟨*d*⟩ are dimensionless factors, ⟨*e*⟩ and ⟨*f*⟩ are TEX dimensions

Example: `\pgfsys@transformcm{1}{0}{0}{1}{1cm}{1cm}`.

This command is protocoled, see Section 62.

`\pgfsys@transformshift{⟨x displacement⟩}{⟨y displacement⟩}`

This command will change the origin of the canvas to (x, y).

This command has a default implementation and need not be implemented by a driver file.

This command is protocoled, see Section 62.

`\pgfsys@transformxyscale{⟨x scale⟩}{⟨y scale⟩}`

This command will scale the canvas (and everything that is drawn) by a factor of ⟨*x scale*⟩ in the x-direction and ⟨*y scale*⟩ in the y-direction. Note that this applies to everything, including lines. So a scaled line will have a different width and may even have a different width when going along the x-axis and when going along the y-axis, if the scaling is different in these directions. Usually, you do not want this.

This command has a default implementation and need not be implemented by a driver file.

This command is protocoled, see Section 62.

60.4 Stroking, Filling, and Clipping System Commands

`\pgfsys@stroke`

Stroke the current path (as if it were drawn with a pen). A number of graphic state parameters influence this, which can be set using appropriate system commands described later.

Line width The "thickness" of the line. A width of 0 is the thinnest width renderable on the device. On a high-resolution printer this may become invisible and should be avoided. A good choice is 0.4pt, which is the default.

Stroke color This special color is used for stroking. If it is not set, the current color is used.

Cap The cap describes how the endings of lines are drawn. A round cap adds a little half circle to these endings. A butt cap ends the lines exactly at the end (or start) point without anything added. A rectangular cap ends the lines like the butt cap, but the lines protrude over the endpoint by the line thickness. (See also the PDF manual.) If the path has been closed, no cap is drawn.

Join This describes how a bend (a join) in a path is rendered. A round join draws bends using small arcs. A bevel join just draws the two lines and then fills the join minimally so that it becomes convex. A miter join extends the lines so that they form a single sharp corner, but only up to a certain miter limit. (See the PDF manual once more.)

Dash The line may be dashed according to a dashing pattern.

Clipping area If a clipping area is established, only those parts of the path that are inside the clipping area will be drawn.

In addition to stroking a path, the path may also be used for clipping after it has been stroked. This will happen if the `\pgfsys@clipnext` is used prior to this command, see there for details.

This command is protocoled, see Section 62.

`\pgfsys@closestroke`

This command should have the same effect as first closing the path and then stroking it.

This command has a default implementation and need not be implemented by a driver file.

This command is protocoled, see Section 62.

\pgfsys@fill

This command fills the area surrounded by the current path. If the path has not yet been closed, it is closed prior to filling. The path itself is not stroked. For self-intersecting paths or paths consisting of multiple parts, the nonzero winding number rule is used to determine whether a point is inside or outside the path, except if \ifpgfsys@eorule holds – in which case the even-odd rule should be used. (See the PDF or PostScript manual for details.)

The following graphic state parameters influence the filling:

Interior rule If \ifpgfsys@eorule is set, the even-odd rule is used, otherwise the non-zero winding number rule.

Fill color If the fill color is not especially set, the current color is used.

Clipping area If a clipping area is established, only those parts of the filling area that are inside the clipping area will be drawn.

In addition to filling the path, the path will also be used for clipping if \pgfsys@clipnext is used prior to this command.

This command is protocoled, see Section 62.

\pgfsys@fillstroke

First, the path is filled, then the path is stroked. If the fill and stroke colors are the same (or if they are not specified and the current color is used), this yields almost the same as a \pgfsys@fill. However, due to the line thickness of the stroked path, the fill-stroked area will be slightly larger.

In addition to stroking and filling the path, the path will also be used for clipping if \pgfsys@clipnext is used prior to this command.

This command is protocoled, see Section 62.

\pgfsys@discardpath

Normally, this command should "throw away" the current path. However, after \pgfsys@clipnext has been called, the current path should subsequently be used for clipping. See \pgfsys@clipnext for details.

This command is protocoled, see Section 62.

\pgfsys@clipnext

This command should be issued after a path has been constructed, but before it has been stroked and/or filled or discarded. When the command is used, the next stroking/filling/discarding command will first be executed normally. Then, afterwards, the just-used path will be used for subsequent clipping. If there has already been a clipping region, this region is intersected with the new clipping path (the clipping cannot get bigger). The nonzero winding number rule is used to determine whether a point is inside or outside the clipping area or the even-odd rule, depending on whether \ifpgfsys@eorule holds.

60.5 Graphic State Option System Commands

\pgfsys@setlinewidth{⟨*width*⟩}

Sets the width of lines, when stroked, to ⟨*width*⟩, which must be a TeX dimension.

This command is protocoled, see Section 62.

\pgfsys@buttcap

Sets the cap to a butt cap. See \pgfsys@stroke.

This command is protocoled, see Section 62.

\pgfsys@roundcap

Sets the cap to a round cap. See \pgfsys@stroke.

This command is protocoled, see Section 62.

\pgfsys@rectcap

Sets the cap to a rectangular cap. See \pgfsys@stroke.

This command is protocoled, see Section 62.

`\pgfsys@miterjoin`

Sets the join to a miter join. See `\pgfsys@stroke`.

This command is protocoled, see Section 62.

`\pgfsys@setmiterlimit{`⟨*factor*⟩`}`

Sets the miter limit of lines to ⟨*factor*⟩. See the PDF or PostScript for details on what the miter limit is.

This command is protocoled, see Section 62.

`\pgfsys@roundjoin`

Sets the join to a round join. See `\pgfsys@stroke`.

This command is protocoled, see Section 62.

`\pgfsys@beveljoin`

Sets the join to a bevel join. See `\pgfsys@stroke`.

This command is protocoled, see Section 62.

`\pgfsys@setdash{`⟨*pattern*⟩`}{`⟨*phase*⟩`}`

Sets the dashing patter. ⟨*pattern*⟩ should be a list of TEX dimensions lengths separated by commas. ⟨*phase*⟩ should be a single dimension.

Example: `\pgfsys@setdash{3pt,3pt}{0pt}`

The list of values in ⟨*pattern*⟩ is used to determine the lengths of the "on" phases of the dashing and of the "off" phases. For example, if ⟨*pattern*⟩ is `3bp,4bp`, then the dashing pattern is "3bp on followed by 4bp off, followed by 3bp on, followed by 4bp off, and so on." A pattern of `.5pt,4pt,3pt,1.5pt` means ".5pt on, 4pt off, 3pt on, 1.5pt off, .5pt on, ..." If the number of entries is odd, the last one is used twice, so `3pt` means "3pt on, 3pt off, 3pt on, 3pt off, ..." An empty list means "always on."

The second argument determines the "phase" of the pattern. For example, for a pattern of `3bp,4bp` and a phase of `1bp`, the pattern would start: "2bp on, 4bp off, 3bp on, 4bp off, 3bp on, 4bp off, ..."

This command is protocoled, see Section 62.

`\ifpgfsys@eorule`

Determines whether the even odd rule is used for filling and clipping or not.

`\pgfsys@stroke@opacity{`⟨*value*⟩`}`

Sets the opacity of stroking operations.

`\pgfsys@fill@opacity{`⟨*value*⟩`}`

Sets the opacity of filling operations.

60.6 Color System Commands

The PGF system layer provides a number of system commands for setting colors. These command coexist with commands from the `color` and `xcolor` package, which perform similar functions. However, the `color` package does not support having two different colors for stroking and filling, which is a useful feature that is supported by PGF. For this reason, the PGF system layer offers commands for setting these colors separatedly. Also, plain TEX profits from the fact that PGF can set colors.

For PDF, implementing these color commands is easy since PDF supports different stroking and filling colors directly. For PostScript, a more complicated approach is needed in which the colors need to be stored in special PostScript variables that are set whenever a stroking or a filling operation is done.

`\pgfsys@color@rgb{`⟨*red*⟩`}{`⟨*green*⟩`}{`⟨*blue*⟩`}`

Sets the color used for stroking and filling operations to the given red/green/blue tuple (numbers between 0 and 1).

This command is protocoled, see Section 62.

`\pgfsys@color@rgb@stroke{⟨red⟩}{⟨green⟩}{⟨blue⟩}`

Sets the color used for stroking operations to the given red/green/blue tuple (numbers between 0 and 1).

Example: Make stroked text dark red: `\pgfsys@color@rgb@stroke{0.5}{0}{0}`

The special stroking color is only used if the stroking color has been set since the last `\color` or `\pgfsys@color@xxx` command. Thus, each `\color` command will reset both the stroking and filling colors by calling `\pgfsys@color@reset`.

This command is protocoled, see Section 62.

`\pgfsys@color@rgb@fill{⟨red⟩}{⟨green⟩}{⟨blue⟩}`

Sets the color used for filling operations to the given red/green/blue tuple (numbers between 0 and 1). This color may be different from the stroking color.

This command is protocoled, see Section 62.

`\pgfsys@color@cmyk{⟨cyan⟩}{⟨magenta⟩}{⟨yellow⟩}{⟨black⟩}`

Sets the color used for stroking and filling operations to the given cymk tuple (numbers between 0 and 1).

This command is protocoled, see Section 62.

`\pgfsys@color@cmyk@stroke{⟨cyan⟩}{⟨magenta⟩}{⟨yellow⟩}{⟨black⟩}`

Sets the color used for stroking operations to the given cymk tuple (numbers between 0 and 1).

This command is protocoled, see Section 62.

`\pgfsys@color@cmyk@fill{⟨cyan⟩}{⟨magenta⟩}{⟨yellow⟩}{⟨black⟩}`

Sets the color used for filling operations to the given cymk tuple (numbers between 0 and 1).

This command is protocoled, see Section 62.

`\pgfsys@color@cmy{⟨cyan⟩}{⟨magenta⟩}{⟨yellow⟩}`

Sets the color used for stroking and filling operations to the given cym tuple (numbers between 0 and 1).

This command is protocoled, see Section 62.

`\pgfsys@color@cmy@stroke{⟨cyan⟩}{⟨magenta⟩}{⟨yellow⟩}`

Sets the color used for stroking operations to the given cym tuple (numbers between 0 and 1).

This command is protocoled, see Section 62.

`\pgfsys@color@cmy@fill{⟨cyan⟩}{⟨magenta⟩}{⟨yellow⟩}`

Sets the color used for filling operations to the given cym tuple (numbers between 0 and 1).

This command is protocoled, see Section 62.

`\pgfsys@color@gray{⟨black⟩}`

Sets the color used for stroking and filling operations to the given black value, where 0 means black and 1 means white.

This command is protocoled, see Section 62.

`\pgfsys@color@gray@stroke{⟨black⟩}`

Sets the color used for stroking operations to the given black value, where 0 means black and 1 means white.

This command is protocoled, see Section 62.

`\pgfsys@color@gray@fill{⟨black⟩}`

Sets the color used for filling operations to the given black value, where 0 means black and 1 means white.

This command is protocoled, see Section 62.

\pgfsys@color@reset

This command will be called when the \color command is used. It should purge any internal settings of stroking and filling color. After this call, till the next use of a command like \pgfsys@color@rgb@fill, the current color installed by the \color command should be used.

If the TEX-if \pgfsys@color@reset@inorder is set to true, this command may "assume" that any call to a color command that sets the fill or stroke color came "before" the call to this command and may try to optimize the output accordingly.

An example of an incorrect "out of order" call would be using \pgfsys@color@reset at the beginning of a box that is constructed using \setbox. Then, when the box is constructed, no special fill or stroke color might be in force. However, when the box is later on inserted at some point, a special fill color might already have been set. In this case, this command is not guaranteed to reset the color correctly.

\pgfsys@color@reset@inordertrue

Sets the optimized "in order" version of the color resetting. This is the default.

\pgfsys@color@reset@inorderfalse

Switches off the optimized color resetting.

\pgfsys@color@unstacked{⟨*LaTeX color*⟩}

This slightly obscure command causes the color stack to be tricked. When called, this command should set the current color to ⟨*LaTeX color*⟩ without causing any change in the color stack.

Example: \pgfsys@color@unstacked{red}

60.7 Pattern System Commands

\pgfsys@declarepattern{⟨*name*⟩}{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩} {⟨*x step*⟩}{⟨*y step*⟩}{⟨*code*⟩}{⟨*flag*⟩}

This command declares a new colored or uncolored pattern, depending on whether ⟨*flag*⟩ is 0, which means uncolored, or 1, which means colored. Uncolored patterns have no inherent color, the color is provided when they are set. Colored patters have an inherent color.

The ⟨*name*⟩ is a name for later use when the pattern is to be shown. The pairs (x_1, y_1) and (x_2, y_2) must describe a bounding box of the pattern ⟨*code*⟩.

The tiling step of the pattern is given by ⟨*x step*⟩ and ⟨*y step*⟩.

Example:

```
\pgfsys@declarepattern{hori}{-.5pt}{0pt}{.5pt}{3pt}{3pt}{3pt}
{\pgfsys@moveto{0pt}{0pt}\pgfsys@lineto{0pt}{3pt}\pgfsys@stroke}
{0}
```

\pgfsys@setpatternuncolored{⟨*name*⟩}{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the fill color to the pattern named ⟨*name*⟩. This pattern must previously have been declared with ⟨*flag*⟩ set to 0. The color of the pattern is given in the parameters ⟨*red*⟩, ⟨*green*⟩, and ⟨*blue*⟩ in the usual way.

The fill color "pattern" will persist till the next color command that modifies the fill color.

\pgfsys@setpatterncolored{⟨*name*⟩}

Sets the fill color to the pattern named ⟨*name*⟩. This pattern must have been declared with the 1 flag.

60.8 Scoping System Commands

The scoping commands are used to keep changes of the graphics state local.

\pgfsys@beginscope

Saves the current graphic state on a graphic state stack. All changes to the graphic state parameters mentioned for \pgfsys@stroke and \pgfsys@fill will be local to the current graphic state and the old values will be restored after \pgfsys@endscope is used.

Warning: PDF and PostScript differ with respect to the question of whether the current path is part of the graphic state or not. For this reason, you should never use this command unless the path is currently empty. For example, it might be a good idea to use `\pgfsys@discardpath` prior to calling this command.

This command is protocoled, see Section 62.

`\pgfsys@endscope`

Restores the last saved graphic state.

This command is protocoled, see Section 62.

60.9 Image System Commands

The system layer provides some commands for image inclusion.

`\pgfsys@imagesuffixlist`

This macro should expand to a list of suffixes, separated by ':', that will be tried when searching for an image.

Example: `\def\pgfsys@imagesuffixlist{eps:epsi:ps}`

`\pgfsys@defineimage`

Called, when an image should be defined.

This command does not take any parameters. Instead, certain macros will be preinstalled with appropriate values when this command is invoked. These are:

- `\pgf@filename` File name of the image to be defined.
- `\pgf@imagewidth` Will be set to the desired (scaled) width of the image.
- `\pgf@imageheight` Will be set to the desired (scaled) height of the image.

  If this macro and also the height macro are empty, the image should have its "natural" size.

  If exactly only of them is specified, the undefined value the image is scaled so that the aspect ratio is kept.

  If both are set, the image is scaled in both directions independently, possibly changing the aspect ratio.

The following macros presumable mostly make sense for drivers that can handle PDF:

- `\pgf@imagepage` The desired page number to be extracted from a multi-page "image."
- `\pgf@imagemask` If set, it will be set to `/SMask x 0 R` where `x` is the PDF object number of a soft mask to be applied to the image.
- `\pgf@imageinterpolate` If set, it will be set to `/Interpolate true` or `/Interpolate false`, indicating whether the image should be interpolated in PDF.

The command should now setup the macro `\pgf@image` such that calling this macro will result in typesetting the image. Thus, `\pgf@image` is the "return value" of the command.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@definemask`

This command declares a mask for usage with images. It works similar to `\pgfsys@defineimage`: Certain macros are set when the command is called. The result should be to set the macro `\pgf@mask` to a pdf object count that can subsequently be used as a soft mask. The following macros will be set when this command is invoked:

- `\pgf@filename` File name of the mask to be defined.
- `\pgf@maskmatte` The so-called matte of the mask (see the PDF documentation for details). The matte is a color specification consisting of 1, 3 or 4 numbers between 0 and 1. The number of numbers depends on the number of color channels in the image (not in the mask!). It will be assumed that the image has been preblended with this color.

60.10 Shading System Commands

`\pgfsys@horishading`{⟨*name*⟩}{⟨*height*⟩}{⟨*specification*⟩}

Declares a horizontal shading for later use. The effect of this command should be the definition of a macro called `\@pgfshading`⟨*name*⟩`!` (or `\csname @pdfshading`⟨*name*⟩`!\endcsname`, to be precise). When invoked, this new macro should insert a shading at the current position.

⟨*name*⟩ is the name of the shading, which is also used in the output macro name. ⟨*height*⟩ is the height of the shading and must be given as a TeX dimension like `2cm` or `10pt`. ⟨*specification*⟩ is a shading color specification as specified in Section 55. The shading specification implicitly fixes the width of the shading.

When `\@pgfshading`⟨*name*⟩`!` is invoked, it should insert a box of height ⟨*height*⟩ and the width implicit in the shading declaration.

`\pgfsys@vertshading`{⟨*name*⟩}{⟨*width*⟩}{⟨*specification*⟩}

Like the horizontal version, only for vertical shadings. This time, the height of the shading is implicit in ⟨*specification*⟩ and the width is given as ⟨*width*⟩.

`\pgfsys@radialshading`{⟨*name*⟩}{⟨*starting point*⟩}{⟨*specification*⟩}

Declares a radial shading. Like the previous macros, this command should setup the macro `\@pgfshading`⟨*name*⟩`!`, which upon invocation should insert a radial shading whose size is implicit in ⟨*specification*⟩.

The parameter ⟨*starting point*⟩ is a PGF point specifying the inner starting point of the shading.

60.11 Reusable Objects System Commands

`\pgfsys@invoke`{⟨*literals*⟩}

This command gets protocoled literals and should insert them into the `.pdf` or `.dvi` file using an appropriate `\special`.

`\pgfsys@defobject`{⟨*name*⟩}{⟨*lower left*⟩}{⟨*upper right*⟩}{⟨*code*⟩}

Declares an object for later use. The idea is that the object can be precached in some way and then be rendered more quickly when used several times. For example, an arrow head might be defined and prerendered in this way.

The parameter ⟨*name*⟩ is the name for later use. ⟨*lower left*⟩ and ⟨*upper right*⟩ are PGF points specifying a bounding box for the object. ⟨*code*⟩ is the code for the object. The code should not be too fancy.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@useobject`{⟨*name*⟩}{⟨*extra code*⟩}

Renders a previously declared object. The first parameter is the name of the the object. The second parameter is extra code that should be executed right *before* the object is rendered. Typically, this will be some transformation code.

This command has a default implementation and need not be implemented by a driver file.

60.12 Invisibility System Commands

All drawing or stroking or text rendering between calls of the following commands should be suppressed. A similar effect can be achieved by clipping against an empty region, but the following commands do not open a graphics scope and can be opened and closed "orthogonally" to other scopes.

`\pgfsys@begininvisible`

Between this command and the closing `\pgfsys@endinvisible` all output should be suppressed. Nothing should be drawn at all, which includes all paths, images and shadings. However, no groups (neither TEX groups nor graphic state groups) should be opened by this command.

This command has a default implementation and need not be implemented by a driver file.

This command is protocoled, see Section 62.

\pgfsys@endinvisible

Ends the invisibility section, unless invisibility blocks have been nested. In this case, only the "last" one restores visibility.

This command has a default implementation and need not be implemented by a driver file.

This command is protocoled, see Section 62.

60.13 Position Tracking Commands

The following commands are used to determine the position of text on a page. This is a rather complicated process in general since at the moment when the text is read by TEX the final position cannot be determined, yet. For example, the text might be put in a box which is later put in the headline or perhaps in the footline or perhaps even on a different page.

For these reasons, position tracking is typically a two-stage process. In a first stage you indicate that a certain position is of interest by *marking* it. This will (depending on the details of the backend driver) cause page coordinates or this position to be written to a `.aux` file when the page is shipped. Possibly, the position might also be determined at an even later stage. Then, on a second run of TEX, the position is read from the `.aux` file and can be used.

\pgfsys@markposition{⟨*name*⟩}

Marks a position on the page. This command should be given while normal typesetting is done such as in

```
The value of $x$ is \pgfsys@markposition{here}important.
```

It causes the position `here` to be saved when the page is shipped out.

\pgfsys@getposition{⟨*name*⟩}{⟨*macro*⟩}

This command retrieves a position that has been marked on an earlier run of TEX on the current file. The ⟨*macro*⟩ must be a macro name such as `\mymarco`. It will redefined such that it is

- either just `\relax` or
- a `\pgfpoint...` command.

The first case will happen when the position has not been marked at all or when the file is typeset for the first time, when the coordinates are not yet available.

In the second case, executing ⟨*macro*⟩ yields the position on the page that is to be interpreted as follows: A coordinate like `\pgfpoint{2cm}{3cm}` means "2cm to the right and 3cm up from the origin of the page." The position of the origin of the page is not guaranteed to be at the lower left corner, it is only guaranteed that all pictures on a page use the same origin.

To determine the lower left corner of a page, you can call `\pgfsys@getposition` with ⟨*name*⟩ set to the special name `pgfpageorigin`. By shifting all positions by the amount returned by this call you can position things absolutely on a page.

Example: Referencing a point or the page:

```
The value of $x$ is \pgfsys@markposition{here}important.

Lots of text.

\hbox{\pgfsys@markposition{myorigin}%
\begin{pgfpicture}
  % Switch of size protocol
  \pgfpathmoveto{\pgfpointorigin}
  \pgfusepath{use as bounding box}

  \pgfsys@getposition{here}{\hereposition}
  \pgfsys@getposition{myorigin}{\thispictureposition}

  \pgftransformshift{\pgfpointscale{-1}{\thispictureposition}}
  \pgftransformshift{\hereposition}

  \pgfpathcircle{\pgfpointorigin}{1cm}
  \pgfusepath{draw}
\end{pgfpicture}}
```

60.14 Internal Conversion Commands

The system commands take TEX dimensions as input, but the dimensions that have to be inserted into PDF and PostScript files need to be dimensionless values that are interpreted as multiples of $\frac{1}{72}$in. For example, the TEX dimension $2bp$ should be inserted as `2` into a PDF file and the TEX dimension 10pt as `9.9626401`. To make this conversion easier, the following command may be useful:

`\pgf@sys@bp{`⟨*dimension*⟩`}`

Inserts how many multiples of $\frac{1}{72}$in the ⟨*dimension*⟩ is into the current protocol stream (buffered).

Example: `\pgf@sys@bp{\pgf@x}` or `\pgf@sys@bp{1cm}`.

Note that this command is *not* a system command that can/needs to be overwritten by a driver.

61 The Soft Path Subsystem

This section describes a set of commands for creating *soft paths* as opposed to the commands of the previous section, which created *hard paths*. A soft path is a path that can still be "changed" or "molded." Once you (or the PGF system) is satisfied with a soft path, it is turned into a hard path, which can be inserted into the resulting `.pdf` or `.ps` file.

Note that the commands described in this section are "high-level" in the sense that they are not implemented in driver files, but rather directly by the PGF-system layer. For this reason, the commands for creating soft paths do not start with `\pgfsys@`, but rather with `\pgfsyssoftpath@`. On the other hand, as a user you will never use these commands directly, so they are described as part of the low-level interface.

61.1 Path Creation Process

When the user writes a command like `\draw (0bp,0bp) -- (10bp,0bp);` quite a lot happens behind the scenes:

1. The frontend command is translated by Ti*k*Z into commands of the basic layer. In essence, the command is translated to something like

   ```
   \pgfpathmoveto{\pgfpoint{0bp}{0bp}}
   \pgfpathlineto{\pgfpoint{10bp}{0bp}}
   \pgfusepath{stroke}
   ```

2. The `\pgfpathxxxx` command do *not* directly call "hard" commands like `\pgfsys@xxxx`. Instead, the command `\pgfpathmoveto` invokes a special command called `\pgfsyssoftpath@moveto` and `\pgfpathlineto` invokes `\pgfsyssoftpath@lineto`.

   The `\pgfsyssoftpath@xxxx` commands, which are described below, construct a soft path. Each time such a command is used, special tokens are added to the end of an internal macro that stores the soft path currently being constructed.

3. When the `\pgfusepath` is encountered, the soft path stored in the internal macro is "invoked." Only now does a special macro iterate over the soft path. For each line-to or move-to operation on this path it calls an appropriate `\pgfsys@moveto` or `\pgfsys@lineto` in order to, finally, create the desired hard path, namely, the string of literals in the `.pdf` or `.ps` file.

4. After the path has been invoked, `\pgfsys@stroke` is called to insert the literal for stroking the path.

Why such a complicated process? Why not have `\pgfpathlineto` directly call `\pgfsys@lineto` and be done with it? There are two reasons:

1. The PDF specification requires that a path is not interrupted by any non-path-construction commands. Thus, the following code will result in a corrupted `.pdf`:

   ```
   \pgfsys@moveto{0}{0}
   \pgfsys@setlinewidth{1}
   \pgfsys@lineto{10}{0}
   \pgfsys@stroke
   ```

   Such corrupt code is *tolerated* by most viewers, but not always. It is much better to create only (reasonably) legal code.

2. A soft path can still be changed, while a hard path is fixed. For example, one can still change the starting and end points of a soft path or do optimizations on it. Such transformations are not possible on hard paths.

61.2 Starting and Ending a Soft Path

No special action must be taken in order to start the creation of a soft path. Rather, each time a command like `\pgfsyssoftpath@lineto` is called, a special token is added to the (global) current soft path being constructed.

However, you can access and change the current soft path. In this way, it is possible to store a soft path, to manipulate it, or to invoke it.

`\pgfsyssoftpath@getcurrentpath{`⟨*macro name*⟩`}`

This command will store the current soft path in ⟨*macro name*⟩.

`\pgfsyssoftpath@setcurrentpath{`⟨*macro name*⟩`}`

This command will set the current soft path to be the path stored in ⟨*macro name*⟩. This macro should store a path that has previously been extracted using the `\pgfsyssoftpath@getcurrentpath` command and has possibly been modified subsequently.

`\pgfsyssoftpath@invokecurrentpath`

This command will turn the current soft path in a "hard" path. To do so, it iterates over the soft path and calls an appropriate `\pgfsys@xxxx` command for each element of the path. Note that the current soft path is *not changed* by this command. Thus, in order to start a new soft path after the old one has been invoked and is no longer needed, you need to set the current soft path to be empty. This may seems strange, but it is often useful to immediately use the last soft path again.

`\pgfsyssoftpath@flushcurrentpath`

This command will invoke the current soft path and then set it to be empty.

61.3 Soft Path Creation Commands

`\pgfsyssoftpath@moveto{`⟨x⟩`}{`⟨y⟩`}`

This command appends a "move-to" segment to the current soft path. The coordinates ⟨x⟩ and ⟨y⟩ are given as normal TEX dimensions.

Example: One way to draw a line:

```
\pgfsyssoftpath@moveto{0pt}{0pt}
\pgfsyssoftpath@lineto{10pt}{10pt}
\pgfsyssoftpath@flushcurrentpath
\pgfsys@stroke
```

`\pgfsyssoftpath@lineto{`⟨x⟩`}{`⟨y⟩`}`

Appends a "line-to" segment to the current soft path.

`\pgfsyssoftpath@curveto{`⟨a⟩`}{`⟨b⟩`}{`⟨c⟩`}{`⟨d⟩`}{`⟨x⟩`}{`⟨y⟩`}`

Appends a "curve-to" segment to the current soft path with controls (a, b) and (c, d).

`\pgfsyssoftpath@rect{`⟨*lower left x*⟩`}{`⟨*lower left y*⟩`}{`⟨*width*⟩`}{`⟨*height*⟩`}`

Appends a rectangle segment to the current soft path.

`\pgfsyssoftpath@closepath`

Appends a "close-path" segment to the current soft path.

61.4 The Soft Path Data Structure

A soft path is stored in a standardized way, which makes it possible to modify it before it becomes "hard." Basically, a soft path is a long sequence of triples. Each triple starts with a *token* that identifies what is going on. This token is followed by two dimensions in braces. For example, the following is a soft path that means "the path starts at (0bp, 0bp) and then continues in a straight line to (10bp, 0bp)."

```
\pgfsyssoftpath@movetotoken{0bp}{0bp}\pgfsyssoftpath@linetotoken{10bp}{0bp}
```

A curve-to is hard to express in this way since we need six numbers to express it, not two. For this reasons, a curve-to is expressed using three triples as follows: The command

```
\pgfsyssoftpath@curveto{1bp}{2bp}{3bp}{4bp}{5bp}{6bp}
```

results in the following three triples:

```
\pgfsyssoftpath@curvetosupportatoken{1bp}{2bp}
\pgfsyssoftpath@curvetosupportbtoken{3bp}{4bp}
\pgfsyssoftpath@curvetotoken{5bp}{6bp}
```

These three triples must always "remain together." Thus, a lonely `supportbtoken` is forbidden.

In details, the following tokens exist:

- \pgfsyssoftpath@movetotoken indicates a move-to operation. The two following numbers indicate the position to which the current point should be moved.
- \pgfsyssoftpath@linetotoken indicates a line-to operation.
- \pgfsyssoftpath@curvetosupportatoken indicates the first control point of a curve-to operation. The triple must be followed by a \pgfsyssoftpath@curvetosupportbtoken.
- \pgfsyssoftpath@curvetosupportbtoken indicates the second control point of a curve-to operation. The triple must be followed by a \pgfsyssoftpath@curvetotoken.
- \pgfsyssoftpath@curvetotoken indicates the target of a curve-to operation.
- \pgfsyssoftpath@rectcornertoken indicates the corner of a rectangle on the soft path. The triple must be followed by a \pgfsyssoftpath@rectsizetoken.
- \pgfsyssoftpath@rectsizetoken indicates the size of a rectangle on the soft path.
- \pgfsyssoftpath@closepath indicates that the subpath begun with the last move-to operation should be closed. The parameter numbers are currently not important, but if set to anything different from {0pt}{0pt}, they should be set to the coordinate of the original move-to operation to which the path "returns" now.

62 The Protocol Subsystem

This section describes commands for *protocolling* literal text created by PGF. The idea is that some literal text, like the string of commands used to draw an arrow head, will be used over and over again in a picture. It is then much more efficient to compute the necessary literal text just once and to quickly insert it "in a single sweep."

When protocolling is "switched on," there is a "current protocol" to which literal text gets appended. Once all commands that needed to be protocoled have been issued, the protocol can be obtained and stored using `\pgfsysprotocol@getcurrentprotocol`. At any point, the current protocol can be changed using a corresponding setting command. Finally, `\pgfsysprotocol@invokecurrentprotocol` is used to insert the protocoled commands into the `.pdf` or `.dvi` file.

Only those `\pgfsys@` commands can be protocolled that use the command `\pgfsysprotocol@literal` interally. For example, the definition of `\pgfsys@moveto` in `pgfsys-common-pdf.def` is

```
\def\pgfsys@moveto#1#2{\pgfsysprotocol@literal{#1 #2 m}}
```

All "normal" system-level commands can be protocolled. However, commands for creating or invoking shadings, images, or whole pictures require special `\special`'s and cannot be protocolled.

`\pgfsysprotocol@literalbuffered{`⟨*literal text*⟩`}`

Adds the ⟨*literal text*⟩ to the current protocol, after it has been "`\edef`ed." This command will always protocol.

`\pgfsysprotocol@literal{`⟨*literal text*⟩`}`

First calls `\pgfsysprotocol@literalbuffered` on ⟨*literal text*⟩. Then, if protocolling is currently switched off, the ⟨*literal text*⟩ is passed on to `\pgfsys@invoke`.

`\pgfsysprotocol@bufferedtrue`

Turns on protocolling. All subsequent calls of `\pgfsysprotocol@literal` will append their argument to the current protocol.

`\pgfsysprotocol@bufferedfalse`

Turns off protocolling. Subsequent calls of `\pgfsysprotocol@literal` directly insert their argument into the current `.pdf` or `.ps`.

Note that if the current protocol is not empty when protocolling is switched off, the next call to `\pgfsysprotocol@literal` will first flush the current protocol, that is, insert it into the file.

`\pgfsysprotocol@getcurrentprotocol{`⟨*macro name*⟩`}`

Stores the current protocol in ⟨*macro name*⟩ for later use.

`\pgfsysprotocol@setcurrentprotocol{`⟨*macro name*⟩`}`

Sets the current protocol to ⟨*macro name*⟩.

`\pgfsysprotocol@invokecurrentprotocol`

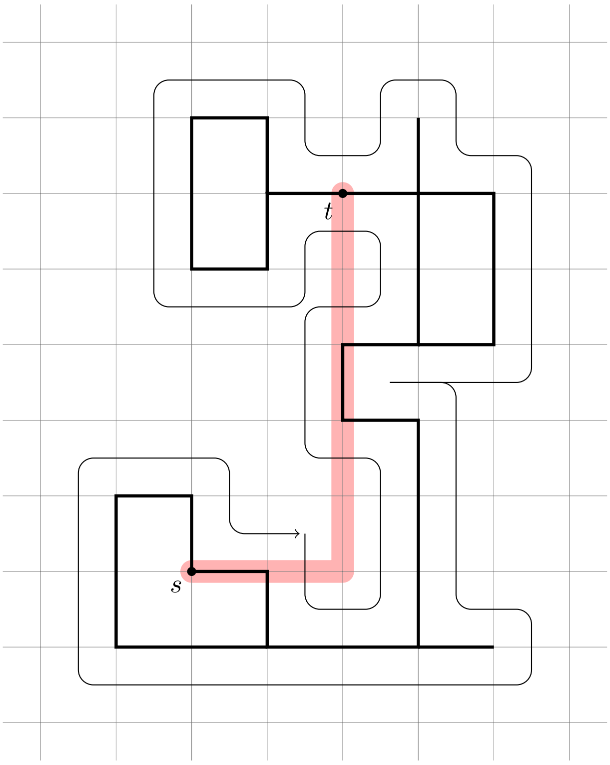
Inserts the text stored in the current protocol into the `.pdf` or `.dvi` file. This does *not* change the current protocol.

`\pgfsysprotocol@flushcurrentprotocol`

First inserts the current protocol, then sets the current protocol to the empty string.

Part IX

References and Index




```
\begin{tikzpicture}
  \draw[line width=0.3cm,color=red!30,cap=round,join=round] (0,0)--(2,0)--(2,5);
  \draw[help lines] (-2.5,-2.5) grid (5.5,7.5);
  \draw[very thick] (1,-1)--(-1,-1)--(-1,1)--(0,1)--(0,0)--
    (1,0)--(1,-1)--(3,-1)--(3,2)--(2,2)--(2,3)--(3,3)--
    (3,5)--(1,5)--(1,4)--(0,4)--(0,6)--(1,6)--(1,5)
    (3,3)--(4,3)--(4,5)--(3,5)--(3,6)
    (3,-1)--(4,-1);
  \draw[below left] (0,0) node(s){$s$};
  \draw[below left] (2,5) node(t){$t$};
  \fill (0,0) circle (0.06cm) (2,5) circle (0.06cm);
  \draw[->,rounded corners=0.2cm,shorten >=2pt]
    (1.5,0.5)-- ++(0,-1)-- ++(1,0)-- ++(0,2)-- ++(-1,0)-- ++(0,2)-- ++(1,0)--
    ++(0,1)-- ++(-1,0)-- ++(0,-1)-- ++(-2,0)-- ++(0,3)-- ++(2,0)-- ++(0,-1)--
    ++(1,0)-- ++(0,1)-- ++(1,0)-- ++(0,-1)-- ++(1,0)-- ++(0,-3)-- ++(-2,0)--
    ++(1,0)-- ++(0,-3)-- ++(1,0)-- ++(0,-1)-- ++(-6,0)-- ++(0,3)-- ++(2,0)--
    ++(0,-1)-- ++(1,0);
\end{tikzpicture}
```

Index

This index only contains automatically generated entries. A good index should also contain carefully selected keywords. This index is not a good index.