BEGGINING

PDL

Version 0.1

Xavier Calbet
xcalbet@yahoo.es

Contents

Part I

Perl Quickie

Chapter 1

Introduction

1.1 Introduction

Perl is short for "Practical Extraction and Report Language". It is what is called a scripting language, one in which there is no need to compile the written program or "script". In practice the program is really compiled partly at the beggining of its execution. It has been designed to "make easy things easy and hard things possible".

Some advantages of Perl include:

- It is a good "glue" language. You can put together several programs in an easy way to make them achieve a given goal. This is specially important for Windows users, which normally lack a good scripting language.
- It runs quite fast for a "scripting" language.
- It is available in many different platforms and operating systems. Actually it works under many flavours of UNIX, Linux and all flavours of Windows. A program, if it is written with compatibility in mind, can be written in one platform and executed in another.
- Application development is very fast.
- There is a huge collection of modules that can be attached to any Perl script. They are all available under CPAN (see below).
- Perl is free. More than that, it is "Free Software". Here the term "free" is as in free speech, not free beer. This means that the source code is available for anyone to see and modify and, what is more important, it will **always** be. Even if you never intend to make any changes to the code, it is important that you do have the ability to do so, you can always hire or ask someone else to do it and in the case there is a bug, it should always be possible to correct it.
- It gives the programmers a fair amount of freedom to do the program as they choose. As the Perl slogan goes "There is more than one way to do it".

Some disadvantages of Perl are:

- Is is slow for some applications, like low level programming, writing a driver application or running a numerical model. Although in this last case we will see how you can insert FORTRAN or C subroutines into Perl having the best of both worlds with not too much hazzle.
- The freedom for the programmer might mean writting an unreadable program. If it is not carefully written it can be difficult to read. In fact there is an obfuscated Perl contest.
- Is uses quite a "large" number of computer resources. This means it is not as light as a C program, but it still is very slim for common computer processing power available these days.

1.2 Where to get it, how to install it

It is available on the internet at http://www.perl.com.

It is available today as default on many Linux distributions, so if you are running Linux chances are that you already have it installed.

To install it follow the guidelines that come with the program.

1.3 Comprehensive Perl Archive Network: CPAN

Maybe the most important aspect that makes Perl a very useful programming language is the huge number of modules made by programmers all over the world to extend Perl's capabilities. These modules are collected at the Comprehensive Perl Archive Network or CPAN for short. It is available at

http://www.perl.com/CPAN.

1.4 Getting help

Perl is very well documented, there are plenty of books on Perl and there is also a very good help facility. Typing `man perl` or `perldoc perl` gives you an overview of the available help pages. Perhaps the most useful one is `perlfunc` which lists the functions of Perl, you get it by typing `man perlfunc` or `perldoc perlfunc`. You can also get help from CPAN http://www.perl.com/CPAN.

Chapter 2

Our first Perl program

2.1 The "Hello, World" program

Get hold of a good text processor and write down your first Perl program:

```
#!/usr/bin/perl -w
print "Hello, world\n";
```

The first line tells us that this is a Perl program, it also tells the machine that the following lines must be interpreted as Perl commands. This line must be the first one in the file and it should be written as is shown. The path that appears in this line must be the location where the Perl executable is located, in this example the Perl executable is located at `/usr/bin/perl`. Your system might have your Perl executable in another location.

In this line we can add options to the plain Perl interpreter. In this case we have added the `-w` option which indicates the interpreter that we want to be warned as much as possible of any mistakes we can make. It is advisable to add this option **always**.

The second line tells the computer to print a message to the screen. It consists of a string, ended with a <NEWLINE> character, which is represented in Perl in the usual C convention as `\n`. We see here that all Perl commands must end with a semicolon, just like in C.

Perl programs are normally called by a name followed by a `.pl` extension.

2.2 Comments

In perl, comments can be inserted by pre-fixing a `#`. That is, anything following the `#` is ignored.

Exercise *comment.pl*

Add a comment to the previous example and run it.

2.3 Solution to the exercises

-

Exercise *comment.pl*

```
#!/usr/bin/perl -w

# This line is ignored
# the following line prints a message

print "Hello, world\n";
```

Chapter 3

Variable types

3.1 Introduction

There are three type of data in Perl: scalars, arrays and hashes.

3.2 Scalars

Scalar variables are the ones that just hold one value. This value can be a number or a string. Scalar variables start with a `$` before the variable name. Here is an example:

```
$value=1;
```

Note that implicitly in the program there is no distinction between a number or string variable. They are distinguished by Perl automatically.

3.2.1 Numerical variables

Numerical variables can be added, substracted, divided, etc.. There are also the typical C increment, `++`, and decrement, `--`, operators.

They can be compared with the usual C operators, `>`, `<`, `>=`, `<=`, `!=` and `==`. More on this later.

Exercise *increment.pl*

Write a short program in which you give a numerical value to a variable and then increment it in one.

3.2.2 String variables

Strings are just a set of alphanumeric characters. They can be stated as single-quoted strings and double-quoted strings.

3.2.2.1 Single-quoted strings

When a string is written down between single quotes, the meaning of the string is taken literally. For example, the variable `$mesg`:

```
$mesg='Hello, how are you doing?';
```

will have the value exactly as it is shown in its definition.

3.2.2.2 Double-quoted strings

If the same string is stated within double quotes, some special characters in that string will be interpreted.

The most notables of these special characters are `\n` which includes a <NEWLINE> character in the string and `$` which interprets whatever comes next as a variable and substitutes its value. For example, the variable `mesg`:

```
$mesg="Hello\n How are you doing $name \n";
```

will have the value of "Hello" followed by a <NEWLINE>, then "How are you doing " and whatever the value of `$name` is followed by another <NEWLINE>.

3.2.2.3 Manipulation of string variables

String variables are easily manipulated in Perl. They can be concatenated with `.` and they can be compared with FORTRAN like operators `eq`, `ne`, `lt`, `gt`, `le`, `ge`.

3.2.3 Printing the value of variables

This is done with the `print` statement followed by a space character and then whatever we want to print separated by commas:

```
# Printing something literally
print 'This is a way to print something\n';
# This next line prints the same text but followed
# by a <NEWLINE>
print "This is another way to print something\n";

# Printing the value of a variable
print 'The mesg variable contains the ',$mesg,' value',"\n";
# This prints the same thing
print "The mesg variable contains the $mesg value\n";
# and this too
print 'The mesg variable contains the'.$mesg."value\n";
```

Exercise *incprint.pl*

Write a short program in which you give a numerical value to two variables, then print them, then increment one of them by one and then print it again. Afterwards, print the product of the two.

Exercise *concat.pl*

Print the values of two string variables, then join the two in a third variable and print it.

3.3 Getting values from the keyboard

To get the value of a variable from the keyboard use the `<STDIN>` symbol, which will replace itself with whatever you type in the keyboard until you hit the <NEWLINE> key. Note that the trailing <NEWLINE> character is also included in the string.

A useful function is `chomp` which chops the trailing <NEWLINE> character that comes with the input.

For example:

```
$val=<STDIN>;
chomp($val);
print "You just typed $val\n'';
```

Exercise *fractal1.pl*

Write a program in a filename called *fractal1.pl* in which you ask for two complex number and then you calculate the product of the two, printing them on the screen. Keep this program in a safe place, because we will be extending it until we have a complete beautiful script.

3.4 Arrays

Arrays are variables that hold one or more value. In Perl they begin with an `@`.

To assign them a value we use values enclosed in parentheses and separated by commas. They can also be printed with `print`:

```
#!/usr/bin/perl -w

@artop=(8,5,4,"job","cheese hello");
print "@artop\n";
```

You can access the value of one element of the array referring to it with the index number enclosed in `[]`.

```
print $artop[3],"\n";
print $artop[4],"\n";
```

Note the `$` sign instead of the `@` sign, the reason for this is that this value is not an array any more, but a scalar. Indices, like in C, start at cero.

3.5 References and two dimensional arrays

Is it important to note that arrays are unidimensional, no matter what we do they are always unidimensional arrays of scalars. To overcome this, we can use references. A reference is a scalar variable that points to another variable, very much like in C.

If we define a variable, we can get a reference from it by pre-fixing a \ sign.

```
# The variable
$val=5;
# The reference to the variable
$ref_val=\$val;
```

It is advisable to denote references in a different way than normal variables, like beggining their name with `$ref_`.

We can now retrieve the original value from the reference de-referencing it adding another `$` sign to to name of the variable:

```
print "The orginal value is ",${$ref_val},"\n";
```

In this way we can include arrays in arrays like this

```
@array1=(2,3,4,5);
@array2=(5,6,7);
@array3=(7,8,9,1,2,3,4);

@ar2d=(\@array1,\@array2,\@array3);
```

To get the values back we simply do:

```
print "@{$ar2d[0]}\n";
```

this gives us back `@array1`. We can also get a single element of that referenced array, for example, element one from `@array1`,

```
print $ar2d[0]->[1];
# Or equivalently
print $ar2d[0][1];
```

This last form can also be used to give values to an array like this:

```
$pp[0][0]=1;
$pp[0][1]=3;
$pp[0][2]=4;
$pp[1][0]=6;
$pp[1][1]=9;
$pp[1][2]=10;
```

with no previous allocation of memory like in other languages. Perl makes space as needed. This last method is the recomended one for persons used to numerical calculations since it resembles very much the written mathematical notation.

Finally we can use what is called anonymous arrays which is just a reference to an array but with no variable name, we do this using `[]`:

```
$yes=[1,2,3,4,5];
print $$yes[0]," ",$$yes[1],"\n";
```

3.6 Functions to operate with arrays

The following functions are useful to work with arrays

- `push` and `pop` adds or removes an element from the end of the array:

  ```
  push(@list,$newvalue);
  $oldvalue=pop(@list);
  ```

- `shift` and `unshift` removes or adds an element at the beggining of an array:

  ```
  unshift(@list,$newvalue);
  $oldvalue=shift(@list);
  ```

- `sort` sorts the elements of a list

  ```
  @sorted_list=sort(@list);
  ```

  If this command is used as shown it will sort the list alphanumerically. To make a numerical sort use

  ```
  @sorted_numerical_list=sort {$a <=> $b} @numerical_list;
  ```

  See the `perlfunc` man page for more details.

- The `$#array` gives the last index of `@array`, so it is useful to find the size of arrays.

3.6.1 Exercise

Make a program that accepts several numbers from the keyboard and stores them in a list. In the end print the result.

3.7 Hashes

Hashes are very similar to arrays, but instead of having a numerical index they have an alphanumeric key and are not stored in an ordered way. A certain value is accesed like this:

```
$hh{"hello"}="bye";
```

The entire hash is named with `%hh`.
Anonymous hashes are created and accesed like this:

```
$hashref={
  'key'   => 'value',
  'Adam'  => 'Eve',
  'Clyde' => 'Bonnie',
};

print $$hasref{'Adam'};
```

See the `perlvar` man page for more details.

3.8 Solution to the exercises

- Exercise *increment.pl*

```
#!/usr/bin/perl -w

$val=0;

$val++;
```

- Exercise *incprint.pl*

```
#!/usr/bin/perl -w

$val=0;
print "Value of val is $val\n";
$val2=$val;
$val2++;
print "Value of val2 is $val2\n";
print "The product of the two is ",$val*$val2,"\n";
```

- Exercise *concat.pl*
- Exercise **??**

```
#!/usr/bin/perl -w
print "Real part of first number?\n";
$re_num1=<STDIN>;
chomp($re_num1);

print "Imaginary part of first number?\n";
$im_num1=<STDIN>;
chomp($im_num1);

print "Real part of second number?\n";
$re_num2=<STDIN>;
chomp($re_num2);

print "Imaginary part of second number?\n";
$im_num2=<STDIN>;
chomp($im_num2);

$re_product=$re_num1*$re_num2-$im_num1*$im_num2;
$im_product=$im_num1*$re_num2+$re_num1*$im_num2;
print "Solution $re_product + $im_product i\n";
```

- Exercise 3.6.1

```
#!/usr/bin/perl -w

$var1=<STDIN>;
push(@list,$var1);
$var2=<STDIN>;
push(@list,$var2);
$var3=<STDIN>;
push(@list,$var3);

print "@list\n";
```

Chapter 4

Control structures

4.1 Statement blocks

A statement block is a sequence of statements enclosed in curly braces:

```
{
   $val=1;
   $array[3]=5;
}
```

4.2 The if/unless statement

It serves to modify execution flow, it can take the following forms:

```
# This is the most simple one
if ($val == 3) {
       print "The value is $val\n";
}

# This is a more sofisticated one
if ($val > 3) {
       print "Big value\n";
} else {
       print "Small value\n";
}

# This is a longer one
if ($val < 3) {
       print "Small value\n";
} elsif ($val >= 3 && $val < 6) {
       print "Intermediate value\n";
} else {
       print "Big value\n";
}
```

The `unless` statement works just like the `if` statement except that the condition is reversed.

4.3 Conditionals

The conditionals between parentheses are first evaluated whether they are true or not. A value of zero, an empty string or an undefined variable is false, everything else is true.

Several conditionals can be concatenated together with a logical or, expresed with `||` or `or`, or with a logical and, expresed with `&&` or `and`. A logical not is expressed as `!` or `not`. The difference between the various flavours of logical operators is in the precedence. For more information see the `perlop` man page.

4.4 The while/until statement

It works like any other `while` statement from other languages. It executes whatever is between curly braces repeated times while the condition between parenthesis is true.

```
while ($val < 5) {
     print "Another round\n";
     $val++;
}
```

The `until` is the same as the `while` statement except that the condition is reversed.

4.5 The do while/until statement

Very similar to the `while` statement except that whatever is enclosed whithin braces is executed at least one time:

```
do {
     print "Hello\n";
     $val++;
} while ($val < 5);
```

4.6 The for statement

It is very similar to the `for` statement from C:

```
# for ( initialization; condition to be met;
#       command executed after each;
#       iteration)
for ($val=0;$val<5;$val++) {
     print "val = $val\n";
}
```

4.7 The foreach statement

This statement takes a list of values and asigns them one at a time to a scalar variable, executing a block of code with each successive assigment.

```
foreach $name ($name_list) {
     print "A name in the list is $name\n";
}
```

4.8 Other control structures

We will briefly describe the `last` and `next` statements. For more information please see the `perlfunc` manual page.

4.8.1 The last statement

It is equivalent to the `break` statement in C. The `last` statement breaks out of the innermost loop before finishing processing all iterations of the loop.

4.8.2 The next statement

This command causes execution to skip past the rest of the innermost enclosing looping block without terminating the block.

4.9 Excercise

Write a program in which you ask for several names and then print them in a sorted order.

4.10 Solution to the exercises

- Excercise 4.9

```
#!/usr/bin/perl -w

$val="";
while($val ne 'end') {
      $val=<STDIN>;
      chomp($val);
      push(@values,$val);
}

@sorted_values=sort(@values);
print "@sorted_values\n";
```

Chapter 5

Input/output

5.1 Opening a file for reading/writing

The way to open a file is with the `open` statement. This statement accepts what is known as a filehandle, which is a named reference of the file for future use in the program, and a string, which contains the name of the file. If the string starts with `>` it is an output file, if it starts with `<` it is an input file and if it starts with `>>` it is an output file in which we will append more data.

```
# Open for output
open(OUTPUT,"> output.file");
# Open for input
open(INPUT,"< input.file");
# Open for appending
open(FILE,">> app.file");
```

5.2 Closing a file

To close a file just type

```
close(FILE);
```

5.3 Reading from a file

The most common way to read a file is reading it line by line. To read just one line from a file we use:

```
$variable=<FILE>;
```

If we want to read a whole file line by line:

```
open(FILE,"< file.txt");
while (<FILE>) {
$line=$_;
chomp($line);
```

```
print $line;
}
close(FILE);
```

It can also be read the file as a whole using an array, having one line per list element:

```
open(FILE,"< file.txt");
@array_file=<FILE>;
close(FILE);
```

5.4 List and scalar context

We will make a slight detour here to explain array and list context. As we have seen when reading files, a command that returns a value, like the `<A_FILE>` command, will have a different behaviour whether the expresion to the left of the equal sign is a scalar or an array. Some commands, like the one just shown, have more or less similar behaviours in both contexts, but others can mean quite different things.

5.5 Writing to a file

To write to a file just use the print statement like this:

```
open(FILE,"> file.txt");
print FILE "This goes into the file $val\n";
close(FILE);
```

If you want formatted output you can use a similar function to C

```
printf FILE "The value is %d\n",$a;
```

There is also a formatted print redirected to a string

```
$string=sprintf "This goes into the string $val\n";
```

5.5.1 Exercise

Write a program that accepts several input parameters from the keyboard and then prints the square of those values to a file.

5.6 Here documents

To avoid typing too many `print` statements when writing to a file, it is sometimes convenient to use the here-doc syntax. It is used like this

```
print FILE <<EOF;
This line goes into FILE
This one too
EOF
```

What it does is it prints literally the lines after the `print` command until the `EOF` string is found at the beggining of the line.

If we want the lines to be interpreted as if they where in double quotes we can use the following

```
print FILE <<EOF;
This line goes into FILE
In this way we can print a variable $val
And again $var
EOF
```

For more information read the `perldata` man page.

5.7 Reading and writing binary files

This is done in exactly the same way as ASCII files, except that the content of the variables we write is binary.

To fill a variable with binary content use the `pack` function:

```
$three_int=pack('iii',4,3,2);
```

this transforms the three numbers 4, 3, 2 into binary integer format, specified by the `'iii'` string. We can then print this variable to a file in the common way.

To read the data back from the binary file we use the `read` function like this,

```
read(FILE,$var,6);
```

this reads 6 bytes from `FILE` into variable `$var`.

To turn these binary values into normal Perl values use the `unpack` function

```
@values=unpack('iii',$var);
```

The format is the same as the `pack` function.

A complete example is shown below:

```
#!/usr/bin/perl -w

# Writing binary file
open(FILE,"> data.bin");
$bin_val=pack('iii',3,2,1);
print FILE $bin_val;
close(FILE);

# Reading binary file
open(FILE,"< data.bin");
read(FILE,$var,12);
@values=unpack('iii',$var);
print "The values from the file are @values\n";
close(FILE);
```

For more information see the `perlfunc` man page.

5.8 Solution to the exercises

- Exercise 5.5.1

```
#!/usr/bin/perl -w

$val='0';
while($val ne 'end') {
   print "Please enter a number (end to finish)\n";
   $val=<STDIN>;
   chomp($val);
   push(@values,$val);
}

pop(@values);

open(FILE,"> squares.txt");
foreach $val (@values) {
   print FILE "The square of $val is ",$val*$val,"\n";
}
close(FILE);
```

Chapter 6

Functions

To structure your programs, the most convenient way to do it is using functions or equivalently subroutines. They can have several scalar input parameters and several scalar output parameters. To define a function it is done like this:

```
sub func_name {
     # Accepting input parameters
     ($input1,$input2,$input3,$input4)=@_;

     # Using them
     $sum=$input1+$input2+$input3+$input4;
     $product=$input1+$input2+$input3+$input4;

     # Sending output parameters
     return ($sum,$product);
}
```

It should be called in this way:

```
($s1,$p1)=&func_name(2,1,6,3);
```

we can also ommit the `&` in front of the function like this

```
($s1,$p1)=func_name(2,1,6,3);
```

Note that you cannot transfer directly several arrays or hashes, this has to be done by reference.

6.1 References to functions

We can also define references of functions with the \ symbol like this

```
$ref_function=\&func_name;
```

we can then use it somewhere else in the program like this

```
&$ref_function(2,1,6,3);
```

We can make anonymous references to functions using `sub` without a name:

```
$ref_function=sub {
     print "Hello\n";
}
```

6.2 Localizing your variables

Until now all variables have been global, when a program grows in size it is important to structure it more and keep the variables local to the function. This is done with the `my` modifier like this:

```
sub func_name {
     # Accepting input parameters
     # Now this variables are only known in the function
     my ($input1,$input2,$input3,$input4)=@_;

     # Using them
     # The same with these two
     my $sum=$input1+$input2+$input3+$input4;
     my $product=$input1+$input2+$input3+$input4;

     # Sending output parameters
     return ($sum,$product);
}
```

These variables are restricted to the surrounding curly braces. Note that if you want to declare with `my` more than one variable in the same line you should enclose them in parenthesis.

Perl also allows us to force us to define all variables, this is done writing

```
use strict;
```

at the beggining of the program. If we still want certain variables to be global we should declare them with `my` at the very beggining of the program before any left curly brace. To declare variables inside the main body of the program we should surround the main body of the program with curly braces and put our `my` declarations inside them. If we want a variable to be re-defined several times so it uses a new memory space each time we use we should include `my` in its assigment. Here is an example:

```
#!/usr/bin/perl -w

# Whether you write down this or not does not matter
# in the scope of the variables, it just enforces
# us to declare all variables with \verb/my/
use strict;

#Now the global variables
my $glob1;
my ($glob2,$glob3,$glob4);
```

```
# Starting the main body of the program
{
   # These variables are local to the main body
   my ($fact1,$fact2);
   my ($i);

   $glob1=3;
   $glob2=4;
   $glob3=5;
   $glob4=10;

   ($fact1,$fact2)=pol($glob1,$glob2,$glob3);

   print "The result is $fact1  $fact2\n";

   for ($i=0;$i<6;$i++) {
# This variable is re-defined over and over again
        # So we completely forget past values and allocate
        # a new memory space
my $var=$fact1*$fact2;

print "This is the product $var\n";
   }

}

# Definition of a subroutine
sub pol {

   # Variables local to a subroutine
   my ($loc1,$loc2)=@_;

   return ($loc1+$loc2**2,$loc1**2+$loc2);
}
```

In a certain part of the program, we can stop using this condition, again, restricted to the surrounding curly braces like this

```
{
  # Here strict is not valid
  no strict;
  $pepe=0;
}
```

6.3 Exercise

Take the previous frac.pl program that calculates the product of two complex numbers and convert the product algorithm into a separate function.

6.4 Solution to the exercises

- Exercise 6.3

```
#!/usr/bin/perl -w
print "Real part of first number?\n";
$re_num1=<STDIN>;
chomp($re_num1);

print "Imaginary part of first number?\n";
$im_num1=<STDIN>;
chomp($im_num1);

print "Real part of second number?\n";
$re_num2=<STDIN>;
chomp($re_num2);

print "Imaginary part of second number?\n";
$im_num2=<STDIN>;
chomp($im_num2);

($re_product,$im_product)=comp_prod($re_num1,$im_num1,
                                    $re_num2,$im_num2);

print "Solution $re_product + $im_product i\n";

sub comp_prod {
    my ($re1,$im1,$re2,$im2)=@_;

    return($re1*$re2-$im1*$im2,
           $im1*$re2+$re1*$im2);
}
```

Chapter 7

A detour: writing a complete program

We now have all the elements necessary to write a complete program. It will be a numerical computation program based on the Newton root finding method. We will keep it for future reference, since we will time it with other equivalent programs written in other languages or in a different way with Perl.

7.1 The Newton root finding method

There is a method to find the root of an equation based on using the slope of the curve that defines the equation.

Suppose you want to find the roots of the following equation:

$$f(x) = 0 \tag{7.1}$$

What we do is we start at a point which we think is close to the root. We find its derivative and plot a tangent line from that point to the x axis (see Fig. 7.1). Wherever this line cuts the x axis is our new solution which we will use again iteratively to get very close to the real solution.

The equation of a line tangent to the initial guessed point, x_i is

$$y = f(x_i) - f'(x_i) + f'(x_i)x. \tag{7.2}$$

To find where it cuts the x axis we make $y = 0$. The new x value, x_{i+1} will be our new estimation of the root, so

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{7.3}$$

7.2 An example

As an example we will study the solutions to

$$x^4 = 1 \tag{7.4}$$

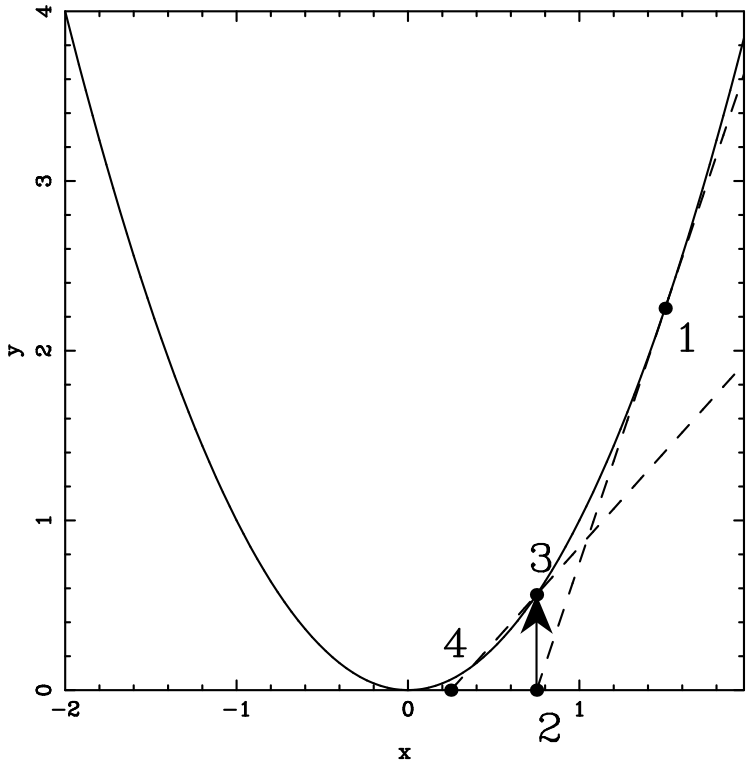

Figure 7.1: Root finding with the Newton method. We start with a point close to the root of the function (1). A downhill linear extrapolation is made with the slope of the curve at that point until the x-axis is met (2). This point, (2), is the first aproximation to the root. We move with constant x to the curve (3) and repeat the process. Point (4) is the second approximation.

We know that this equation has solutions in the complex plane $x = 1, -1, i, -i$
The Newton method applied to this example gives:

$$x_{i+1} = x_i - \frac{x^4 - 1}{4x^3} \tag{7.5}$$

7.2.1 Exercise

Write a subroutine that divides two imaginary numbers.

7.2.2 Exercise

Write a script that starts at a given point in the complex plane and then iterates five times with the Newton method to find the root of $x^4 = 1$. Print the resulting root.

7.2.3 Exercise

Start with a 100x100 square on the complex plane between (-1,1) and (i,-i) as initial guessing points. Run the Newton method solution for all these initial values with just 5 iterarions each. If the program is too slow make the size of the square smaller.

7.2.4 Exercise

Do the same thing as the last exercise. Label each one of the known exact solutions, $x = 1, -1, i, -i$, with an integer number from 0 to 3. Find out the closest exact solution to the numerical computed solution and store its label on a matrix the size of the initial 100x100 square. Write this matrix to a file.

7.3 Timing programs

To compare the speed of different languages it is important to note the amount of CPU time the programs use. To do this we can use the `time` command, which gives the time a program takes to run. It is used by typing `time command_to_time`.

We will use as a reference the program equivalent to the one from Exercise 7.2.4 but written in C. The time this program takes to execute will be a reference which we would like to be as close to. The C program is shown below:

```
#include <stdio.h>
#include <math.h>

main()
{

FILE *fp2;

double rz,iz;
double in_rez,fin_rez,in_imz,fin_imz;
__complex__ double z;
long int lonpint,sobra;
unsigned short int i,j,k,kk,zeroe;
unsigned short int mat[1200][1200];
unsigned long int count;

double pasore,pasoim;

double mindisol;
__complex__ double sol[4];
double disol;
double rdz,idz;


zeroe=0;

lonpint=100;
```

```
in_rez=-1;
fin_rez=1;
in_imz=-1;
fin_imz=1;


pasore=(fin_rez-in_rez)/( (double)lonpint );
pasoim=(fin_imz-in_imz)/( (double)lonpint );

sol[0]=1;
sol[1]=-1;
sol[2]=1i;
sol[3]=-1i;


for (i=0;i<lonpint;i++) {
  //printf("i = %d\n",i);
  for (j=0;j<lonpint;j++) {
    mindisol=700000000;
    kk=4;
    rz=in_rez+( (double)i )*pasore;
    iz=in_imz+( (double)j )*pasoim;
    z=rz+iz*1i;
    for (count=0;count<5;count++) {
      z=z-(z*z*z*z-1)/(4*z*z*z);
    }
    for (k=0;k<4;k++) {
      disol=(z-sol[k])*~(z-sol[k]);
      if (disol < mindisol) {
mindisol=disol;
kk=k;
      }
    }
    mat[i][j]=kk;
    //printf("i j mat %d %d %d\n",i,j,mat[i][j]);
  }
}

fp2=fopen("salida.txt","w");

for (i=0;i<lonpint;i++) {
  for (j=0;j<lonpint;j++) {
    fprintf(fp2,"%u ",mat[i][j]);
  }
  fprintf(fp2,"\n");
}

fclose(fp2);
```

Language	CPU time (secs.)
C	0.33
Perl	32.50

Table 7.1: Benchmark of the frac program from Exercise 7.2.4 written in C and Perl

```
}
```

Benchmarks were taken for the C and Perl programs with five iterations on each point and a 100x100 square on a Pentium 100 PC. The results are the following:

We can see that regular Perl for numerical calculations is very aproximately 100 times slower than the fastest high level programming language, C.

7.3.1 Exercise

Time your program to compare in the future with the same program written in a more efficient manner.

7.4 Solution to the exercises

- Exercise 7.2.1

```
sub cdiv {
    my ($re1,$im1,$re2,$im2)=@_;

    return ( ($re1*$re2+$im1+$im2)/($re2**2+$im2**2),
             ($im1*$re2-$re1*$im2)/($re2**2+$im2**2) );
}
```

- Exercise 7.2.2

```
#!/usr/bin/perl -w

$re_ini=1.3;
$im_ini=0.5;

$re_x=$re_ini;
$im_x=$im_ini;

for ($count=0;$count<1000;$count++) {
    ($re_x2,$im_x2)=cpro($re_x ,$im_x ,$re_x,$im_x);
    ($re_x3,$im_x3)=cpro($re_x2,$im_x2,$re_x,$im_x);
    ($re_x4,$im_x4)=cpro($re_x3,$im_x3,$re_x,$im_x);

    ($re_dx,$im_dx)=cdiv($re_x4 - 1, $im_x4,
 4 * $re_x3, 4 * $im_x3);
```

```
    $re_x=$re_x-$re_dx;
    $im_x=$im_x-$im_dx;
}

print "Initial value  $re_ini + $im_ini i\n";
print "Final   value  $re_x + $im_x i\n";

sub cpro {
    my ($re1,$im1,$re2,$im2)=@_;

    return($re1*$re2-$im1*$im2,
           $im1*$re2+$re1*$im2);
}

sub cdiv {
    my ($re1,$im1,$re2,$im2)=@_;

    return ( ($re1*$re2+$im1+$im2)/($re2**2+$im2**2),
             ($im1*$re2-$re1*$im2)/($re2**2+$im2**2) );
}
```

- Exercise 7.2.3

```
#!/usr/bin/perl -w

use strict;

my ($long_sq);
my ($re_ini_sq,$re_end_sq,$im_ini_sq,$im_end_sq);
my ($re_step,$im_step);
my ($i,$j);
my ($re_x,$im_x);
my ($re_x2,$im_x2);
my ($re_x3,$im_x3);
my ($re_x4,$im_x4);
my ($re_dx,$im_dx);
my ($count);

$long_sq=100;

$re_ini_sq=-1;
$re_end_sq=1;
$im_ini_sq=-1;
$im_end_sq=1;


$re_step=($re_end_sq-$re_ini_sq)/$long_sq;
$im_step=($im_end_sq-$im_ini_sq)/$long_sq;
```

```
for ($i=0;$i<$long_sq;$i++) {
  print "i = $i\n";
  for ($j=0;$j<$long_sq;$j++) {
    $re_x=$re_ini_sq+$i*$re_step;
    $im_x=$im_ini_sq+$j*$im_step;

    for ($count=0;$count<5;$count++) {
($re_x2,$im_x2)=cpro($re_x ,$im_x ,$re_x,$im_x);
($re_x3,$im_x3)=cpro($re_x2,$im_x2,$re_x,$im_x);
($re_x4,$im_x4)=cpro($re_x3,$im_x3,$re_x,$im_x);

($re_dx,$im_dx)=cdiv($re_x4 - 1, $im_x4,
     4 * $re_x3, 4 * $im_x3);

$re_x=$re_x-$re_dx;
$im_x=$im_x-$im_dx;
    }

  }
}



sub cpro {
    my ($re1,$im1,$re2,$im2)=@_;

    return($re1*$re2-$im1*$im2,
           $im1*$re2+$re1*$im2);
}

sub cdiv {
    my ($re1,$im1,$re2,$im2)=@_;

    if ($re2**2+$im2**2 == 0) { return(0,0); }

    return ( ($re1*$re2+$im1+$im2)/($re2**2+$im2**2),
             ($im1*$re2-$re1*$im2)/($re2**2+$im2**2) );
}
```

- Exercise 7.2.4

```
#!/usr/bin/perl -w

use strict;

my ($long_sq);
my ($re_ini_sq,$re_end_sq,$im_ini_sq,$im_end_sq);
my ($re_step,$im_step);
my ($i,$j);
my ($re_x,$im_x);
```

```
my ($re_x2,$im_x2);
my ($re_x3,$im_x3);
my ($re_x4,$im_x4);
my ($re_dx,$im_dx);
my ($count);


my (@re_sol,@im_sol);
my ($disol,$mindisol);
my (@mat);
my ($kk,$k);

$long_sq=100;

$re_ini_sq=-1;
$re_end_sq=1;
$im_ini_sq=-1;
$im_end_sq=1;

$re_sol[0]=1;
$im_sol[0]=0;
$re_sol[1]=-1;
$im_sol[1]=0;
$re_sol[2]=0;
$im_sol[2]=1;
$re_sol[3]=0;
$im_sol[3]=-1;


$re_step=($re_end_sq-$re_ini_sq)/$long_sq;
$im_step=($im_end_sq-$im_ini_sq)/$long_sq;


for ($i=0;$i<$long_sq;$i++) {
  print "i = $i\n";
  for ($j=0;$j<$long_sq;$j++) {
    $mindisol=7000000;
    $kk=4;
    $re_x=$re_ini_sq+$i*$re_step;
    $im_x=$im_ini_sq+$j*$im_step;

    for ($count=0;$count<5;$count++) {
($re_x2,$im_x2)=cpro($re_x ,$im_x ,$re_x,$im_x);
($re_x3,$im_x3)=cpro($re_x2,$im_x2,$re_x,$im_x);
($re_x4,$im_x4)=cpro($re_x3,$im_x3,$re_x,$im_x);

($re_dx,$im_dx)=cdiv($re_x4 - 1, $im_x4,
     4 * $re_x3, 4 * $im_x3);

$re_x=$re_x-$re_dx;
```

```
$im_x=$im_x-$im_dx;
    }

    for ($k=0;$k<4;$k++) {
$disol=($re_x-$re_sol[$k])**2+($im_x-$im_sol[$k])**2;
if ($disol < $mindisol) {
    $mindisol=$disol;
    $kk=$k;
}
    }
    $mat[$i][$j]=$kk;

  }
}

open(FILE,"> output.txt");
for ($i=0;$i<$long_sq;$i++) {
  for ($j=0;$j<$long_sq;$j++) {
      print FILE " $mat[$i][$j]";
  }
  print FILE "\n";
}

close(FILE);

sub cpro {
    my ($re1,$im1,$re2,$im2)=@_;

    return($re1*$re2-$im1*$im2,
           $im1*$re2+$re1*$im2);
}

sub cdiv {
    my ($re1,$im1,$re2,$im2)=@_;

    if ($re2**2+$im2**2 == 0) { return(0,0); }

    return ( ($re1*$re2+$im1*$im2)/($re2**2+$im2**2),
             ($im1*$re2-$re1*$im2)/($re2**2+$im2**2) );
}
```

Chapter 8

Directories and files

8.1 Directories

8.1.1 Accessing directories

Perl can be used to access directories directly. The most common way to access them is with directory handles. They are very similar to file handles, but instead of reading lines from a file, it reads filenames from a directory. To open a directory the `opendir` command is used. To access it, the `readdir` command is used. The command `closedir` is used to close it. Here is an example:

```
#!/usr/bin/perl -w

# Opening directory
opendir(DIR,"/home/pepe");
# Reading filenames from direcotory
while ($name=readdir(DIR)) {
    print "$name\n";
}
closedir(DIR);
```

8.1.2 Changing directories

To change the current working directory to another one use the command `chdir` like this

```
chdir("/home/pepe/bin");
```

8.1.3 Globbing

You can also use the equivalent to UNIX command line wildcards to access some given filenames. This is done with the `glob` command. It can be used like this

```
@files=glob("/etc/passwd*");
```

8.1.4 Making and removing directories

This is done with the `mkdir` and `rmdir` commands.

To obtain more information on how to use these commands consult the `perlfunc` manual page.

8.2 Files

8.2.1 Removing and renaming files

This is done with the `unlink` and `rename` commands. For more information see the `perlfunc` man page.

8.2.2 File tests

The properties of a file can be known from inside a Perl script. This is done with the -x file tests.

For example, we can test if a given file is executable by typing:

```
if (-x $filename) {
print "File $filename is executable\n";
}
```

There are a lot of other Perl tests like these. See the `perlfunc` manpage under the `-X` header.

Chapter 9

Processes

9.1 Running external programs

A way to run an external program from Perl is to use the `system` command. This command runs a program just as if it were typed from the command line shell and returns the return status of the program. For example:

```
system("ls");
\begin{verbatim}

If we want to capture the standard output
of the program in a variable use the backquotes:

\begin{verbatim}
$output=`ls`;
```

If we want the external program to be run and the Perl script to die we can use the `exec` command:

```
exec("ls -l");
```

9.1.1 Using processes as filehandles

Regular programs can also be used as regular filehandles. For example, we can open the UNIX `ls` command for reading like this:

```
open(LSOUT,"ls |");
while (<LSOUT>) {
    print $_;
}
```

or for writing

```
open(LPR,"| lpr");
print LPR "Print this in the default printer\n";
```

A process can also be opened as a file and its input and output processed. This feature has to be handled with care since it can lead to a deadlock.

Here is an example:

```
#!/usr/bin/perl -w

use IPC::Open2;

open2(*README,*WRITEME, $program);
print WRITEME "Writing to the program\n";
$reading_from_program=<README>;
close(WRITEME);
close(README);
```

9.1.2 Exercise

We know that we can numerically solve the following differential equations

$$\frac{dx}{dt} = -sx + sy \tag{9.1}$$
$$\frac{dy}{dt} = -xz + rx - y \tag{9.2}$$
$$\frac{dz}{dt} = xy - bz \tag{9.3}$$
$$\tag{9.4}$$

with the following `octave` or `matlab` program

```
function xdot=f(x,t)
  xdot=zeros(3,1);
  xdot(1)=-s*x(1)+s*x(2);
  xdot(2)=-x(1)*x(3)+r*x(1)-x(2);
  xdot(3)=x(1)*x(2)-b*x(3);
endfunction
x0=[0.5;0.5;0.5];
t=linspace(0,50,10000);
y=lsode("f",x0,t);
printf("%g,",t);
printf("\n");
printf("%g,",y(:,1));
printf("\n");
printf("%g,",y(:,2));
printf("\n");
printf("%g,",y(:,3));
printf("\n");
```

but we want to control the program from Perl. We should be able to change the parameters s, r, b from Perl, then calculate the solution to these equations and later on get the results. As an example set $s = 10$, $b = 8/3$ and $r = 28$.

9.2 Solution to the exercises

- Exercise 9.1.2

```
#!/usr/bin/perl -w

use IPC::Open2;

$s=10;
$b=8/3;
$r=28;

open2(*README,*WRITEME,'octave -q');

print WRITEME <<'EOF';
function xdot=f(x,t)
  xdot=zeros(3,1);
EOF

print WRITEME <<"EOF";
  xdot(1)=-$s*x(1)+$s*x(2);
  xdot(2)=-x(1)*x(3)+$r*x(1)-x(2);
  xdot(3)=x(1)*x(2)-$b*x(3);
EOF

print WRITEME <<'EOF';
endfunction
x0=[0.5;0.5;0.5];
t=linspace(0,50,10000);
y=lsode("f",x0,t);
printf("%g,",t);
printf("\n");
printf("%g,",y(:,1));
printf("\n");
printf("%g,",y(:,2));
printf("\n");
printf("%g,",y(:,3));
printf("\n");
EOF

$t=<README>;
$x=<README>;
$y=<README>;
$z=<README>;

chomp($t,$x,$y,$z);

close(WRITEME);
close(README);

print "t = $t\n";
print "x = $x\n";
print "y = $y\n";
print "z = $z\n";
```

Chapter 10

Regular expressions

10.1 General concepts

A regular expression is a pattern, a template, to be matched againts a string. We can check whether a particular regular expression exists in a string, or change a regular expression for another one, or change the case of the string.

10.2 Simple regular expressions

Regular expressions are denoted by enclosing them between slashes. Normally letters or numbers are represented by themselves in regular expressions. This is a valid regular expression

```
/hola/
```

which matches any string that has `hola` in it.

These "normal" characters can be modified. If you add a `*` after a character it means that character repeated cero or more times, so

```
/ho*la/
```

matches the string `hla`, `hola`, `hoola`, `hooola` and so on.

Another interesting modifier is `+` which means repeat the last character one or more times, so

```
/ho+la/
```

matches `hola`, `hoola`, `hooola` and so on, but not `hla`.

There are many more modifiers. Regular expressions can be a very powerful way to parse text files. To see more have a look at the `perlre` man page.

10.3 Testing the match of a regular expression

We can see if a given string matches a regular expression like this

```
if ($str =~ /hola/) {
    print 'The string $str contains the string hola';
}
```

To test the opossite type

```
if ($str !~ /hola/) {
    print 'The string $str does NOT contains the string hola';
}
```

10.4 Substitutions

You can also substitute a regular expression with another one, this is made by using

```
$str='This is the content';

print "$str\n";
# Now we substitute all 'i's with 'a'
$str =~ s/i/a/g;
print "$str\n";
```

This substitution can be made case insensitive. You can also change from lowercase letters to uppercase, etc. See the `perlre` man page for more details.

10.5 Split and join commands

10.5.1 Split

The `split` function splits a string into fields separated by a given regular expression. These fields are given as elements of an array. For example,

```
@fields=split(/,/,$line);
```

splits the content of `$line` in fields separated by commas.
To separate fields within whitespaces use `split(' ', $line)`.

10.5.2 Join

The `join` function does the opossite to `split`, it joins the elements of an array into a string with another string as delimiter. For example this joins the elements of an array into a string separating them with :

```
$str=join(":",@array);
```

10.5.3 Exercise

Make a program that reads a text file with fields separated by commas and change them with whitespces using `split` and `join`.

10.6 Solution to the exercises

- Exercise 10.5.3

```
#!/usr/bin/perl -w

open(FILE,"< infile.txt");
open(OUT,"> outfile.txt");
while (<FILE>) {
   @arr=split(/:/,$_);
   $line=join(' ',@arr);
   print OUTFILE $line;
}
close(FILE);
close(OUT);
```

Chapter 11

Modules

11.1 Using modules

As already stated, one of the stregths of Perl is its enormous amount of modules which are available on http://www.perl.com/CPAN. To use one of such modules, if it is not available in the regular Perl distribution it must be installed on your local computer. Once this is accomplished it is called by using a `use` statement at the beggining of the program,

```
#!/usr/bin/perl -w

use Module;

# Rest of the program
```

For example, there is a modules to compute complex numbers within Perl which is called `Math::Complex`

For more information about this package type `man Math::Complex`.

11.2 Making modules

To make your own modules just type your subroutines in regular Perl and pre-pend them with the `package` statement. When this statement is encountered it means that a new namespace is created with its own global and local variables until another `package` statement is found or the end of file. For example to make a package with a subroutine that adds two numbers we can do

```
package Sum;

$global_var=15;

sub add {
    my ($a,$b)=@_;

    return $a+$b;
}
```

To access the subroutines or global variables of this package from another package or from the main program we must pre-pend the name of the package like so,

```
print "$a plus $b is equal to ",Sum::add($a,$b),"\n";
print "The global variable from  package Sum is ",
       $Sum::global_var,"\n";
```

The package can be separated in a different file with a `.pm` suffix and the invoked in the program with the `use` statement.

11.2.1 Exercise

Write the previous Newton root finding method program including the subroutines in a package.

11.2.2 Exercise

Separate the main program and the package in two separate files.

11.3 Importing subroutines names

Sometimes it is desirable to to know the subroutines in a package outside from it, without pre-pending the package name. This is done adding three more lines after the `package` directive like this

```
package Sum;
use Exporter;
@ISA=('Exporter');
@EXPORT=('add'); # Put more subroutines here
```

and then using the `use` statement specifying the routines we want to access

```
use Sum;
```

For more information on modules see the `perlmod` man page.

11.3.1 Exercise

Write again the program from the previous exercise without pre-pending `Complex::` in the calling funtcions.

11.4 Solution to the exercises

- Exercise 11.2.1

  ```
  #!/usr/bin/perl -w

  use strict;

  my ($long_sq);
  ```

```
my ($re_ini_sq,$re_end_sq,$im_ini_sq,$im_end_sq);
my ($re_step,$im_step);
my ($i,$j);
my ($re_x,$im_x);
my ($re_x2,$im_x2);
my ($re_x3,$im_x3);
my ($re_x4,$im_x4);
my ($re_dx,$im_dx);
my ($count);


my (@re_sol,@im_sol);
my ($disol,$mindisol);
my (@mat);
my ($kk,$k);

$long_sq=100;

$re_ini_sq=-1;
$re_end_sq=1;
$im_ini_sq=-1;
$im_end_sq=1;

$re_sol[0]=1;
$im_sol[0]=0;
$re_sol[1]=-1;
$im_sol[1]=0;
$re_sol[2]=0;
$im_sol[2]=1;
$re_sol[3]=0;
$im_sol[3]=-1;


$re_step=($re_end_sq-$re_ini_sq)/$long_sq;
$im_step=($im_end_sq-$im_ini_sq)/$long_sq;


for ($i=0;$i<$long_sq;$i++) {
  #print "i = $i\n";
  for ($j=0;$j<$long_sq;$j++) {
    $mindisol=7000000;
    $kk=4;
    $re_x=$re_ini_sq+$i*$re_step;
    $im_x=$im_ini_sq+$j*$im_step;

    for ($count=0;$count<5;$count++) {
($re_x2,$im_x2)=Complex::cpro($re_x ,$im_x ,$re_x,$im_x);
($re_x3,$im_x3)=Complex::cpro($re_x2,$im_x2,$re_x,$im_x);
($re_x4,$im_x4)=Complex::cpro($re_x3,$im_x3,$re_x,$im_x);
```

```
($re_dx,$im_dx)=Complex::cdiv($re_x4 - 1, $im_x4,
     4 * $re_x3, 4 * $im_x3);

$re_x=$re_x-$re_dx;
$im_x=$im_x-$im_dx;
    }

    for ($k=0;$k<4;$k++) {
$disol=($re_x-$re_sol[$k])**2+($im_x-$im_sol[$k])**2;
if ($disol < $mindisol) {
    $mindisol=$disol;
    $kk=$k;
}
    }
    $mat[$i][$j]=$kk;

  }
}

open(FILE,"> output.txt");
for ($i=0;$i<$long_sq;$i++) {
  for ($j=0;$j<$long_sq;$j++) {
      print FILE " $mat[$i][$j]";
  }
  print FILE "\n";
}

close(FILE);



package Complex;

sub cpro {
    my ($re1,$im1,$re2,$im2)=@_;

    return($re1*$re2-$im1*$im2,
           $im1*$re2+$re1*$im2);
}

sub cdiv {
    my ($re1,$im1,$re2,$im2)=@_;

    if ($re2**2+$im2**2 == 0) { return(0,0); }

    return ( ($re1*$re2+$im1*$im2)/($re2**2+$im2**2),
             ($im1*$re2-$re1*$im2)/($re2**2+$im2**2) );
}
```

- Exercise 11.2.2

Program file

```
#!/usr/bin/perl -w

use strict;
use Complex;

my ($long_sq);
my ($re_ini_sq,$re_end_sq,$im_ini_sq,$im_end_sq);
my ($re_step,$im_step);
my ($i,$j);
my ($re_x,$im_x);
my ($re_x2,$im_x2);
my ($re_x3,$im_x3);
my ($re_x4,$im_x4);
my ($re_dx,$im_dx);
my ($count);


my (@re_sol,@im_sol);
my ($disol,$mindisol);
my (@mat);
my ($kk,$k);

$long_sq=100;

$re_ini_sq=-1;
$re_end_sq=1;
$im_ini_sq=-1;
$im_end_sq=1;

$re_sol[0]=1;
$im_sol[0]=0;
$re_sol[1]=-1;
$im_sol[1]=0;
$re_sol[2]=0;
$im_sol[2]=1;
$re_sol[3]=0;
$im_sol[3]=-1;


$re_step=($re_end_sq-$re_ini_sq)/$long_sq;
$im_step=($im_end_sq-$im_ini_sq)/$long_sq;


for ($i=0;$i<$long_sq;$i++) {
  #print "i = $i\n";
  for ($j=0;$j<$long_sq;$j++) {
    $mindisol=7000000;
    $kk=4;
```

```
    $re_x=$re_ini_sq+$i*$re_step;
    $im_x=$im_ini_sq+$j*$im_step;

    for ($count=0;$count<5;$count++) {
($re_x2,$im_x2)=Complex::cpro($re_x ,$im_x ,$re_x,$im_x);
($re_x3,$im_x3)=Complex::cpro($re_x2,$im_x2,$re_x,$im_x);
($re_x4,$im_x4)=Complex::cpro($re_x3,$im_x3,$re_x,$im_x);

($re_dx,$im_dx)=Complex::cdiv($re_x4 - 1, $im_x4,
     4 * $re_x3, 4 * $im_x3);

$re_x=$re_x-$re_dx;
$im_x=$im_x-$im_dx;
    }

    for ($k=0;$k<4;$k++) {
$disol=($re_x-$re_sol[$k])**2+($im_x-$im_sol[$k])**2;
if ($disol < $mindisol) {
    $mindisol=$disol;
    $kk=$k;
}
    }
    $mat[$i][$j]=$kk;

  }
}

open(FILE,"> output.txt");
for ($i=0;$i<$long_sq;$i++) {
  for ($j=0;$j<$long_sq;$j++) {
      print FILE " $mat[$i][$j]";
  }
  print FILE "\n";
}

close(FILE);
```

Package file `Complex.pm`

```
package Complex;

sub cpro {
    my ($re1,$im1,$re2,$im2)=@_;

    return($re1*$re2-$im1*$im2,
           $im1*$re2+$re1*$im2);
}

sub cdiv {
    my ($re1,$im1,$re2,$im2)=@_;
```

```
    if ($re2**2+$im2**2 == 0) { return(0,0); }

    return ( ($re1*$re2+$im1*$im2)/($re2**2+$im2**2),
             ($im1*$re2-$re1*$im2)/($re2**2+$im2**2) );
}
```

- Exercise 11.3.1

  Program file

```
#!/usr/bin/perl -w

use strict;
use Complex;

my ($long_sq);
my ($re_ini_sq,$re_end_sq,$im_ini_sq,$im_end_sq);
my ($re_step,$im_step);
my ($i,$j);
my ($re_x,$im_x);
my ($re_x2,$im_x2);
my ($re_x3,$im_x3);
my ($re_x4,$im_x4);
my ($re_dx,$im_dx);
my ($count);


my (@re_sol,@im_sol);
my ($disol,$mindisol);
my (@mat);
my ($kk,$k);

$long_sq=100;

$re_ini_sq=-1;
$re_end_sq=1;
$im_ini_sq=-1;
$im_end_sq=1;

$re_sol[0]=1;
$im_sol[0]=0;
$re_sol[1]=-1;
$im_sol[1]=0;
$re_sol[2]=0;
$im_sol[2]=1;
$re_sol[3]=0;
$im_sol[3]=-1;


$re_step=($re_end_sq-$re_ini_sq)/$long_sq;
```

```
$im_step=($im_end_sq-$im_ini_sq)/$long_sq;


for ($i=0;$i<$long_sq;$i++) {
  #print "i = $i\n";
  for ($j=0;$j<$long_sq;$j++) {
    $mindisol=7000000;
    $kk=4;
    $re_x=$re_ini_sq+$i*$re_step;
    $im_x=$im_ini_sq+$j*$im_step;

    for ($count=0;$count<5;$count++) {
($re_x2,$im_x2)=cpro($re_x ,$im_x ,$re_x,$im_x);
($re_x3,$im_x3)=cpro($re_x2,$im_x2,$re_x,$im_x);
($re_x4,$im_x4)=cpro($re_x3,$im_x3,$re_x,$im_x);

($re_dx,$im_dx)=cdiv($re_x4 - 1, $im_x4,
     4 * $re_x3, 4 * $im_x3);

$re_x=$re_x-$re_dx;
$im_x=$im_x-$im_dx;
    }

    for ($k=0;$k<4;$k++) {
$disol=($re_x-$re_sol[$k])**2+($im_x-$im_sol[$k])**2;
if ($disol < $mindisol) {
    $mindisol=$disol;
    $kk=$k;
}
    }
    $mat[$i][$j]=$kk;

  }
}

open(FILE,"> output.txt");
for ($i=0;$i<$long_sq;$i++) {
  for ($j=0;$j<$long_sq;$j++) {
      print FILE " $mat[$i][$j]";
  }
  print FILE "\n";
}

close(FILE);
```

Package file `Complex.pm`

```
package Complex;

no strict;
```

```
use Exporter;
@ISA=('Exporter');
@EXPORT=('cpro','cdiv');


sub cpro {
    my ($re1,$im1,$re2,$im2)=@_;

    return($re1*$re2-$im1*$im2,
           $im1*$re2+$re1*$im2);
}

sub cdiv {
    my ($re1,$im1,$re2,$im2)=@_;

    if ($re2**2+$im2**2 == 0) { return(0,0); }

    return ( ($re1*$re2+$im1*$im2)/($re2**2+$im2**2),
             ($im1*$re2-$re1*$im2)/($re2**2+$im2**2) );
}
```

Chapter 12

Object oriented programming

12.1 Introduction

Object oriented programming is a programming technique in which is given much more importance to the data than to the subroutines of the program. It changes the main focus from the program to the data that is manipulated. The advantage of this aproach is that normally, for a given problem, the data is well known and does not change much in time. On the contrary, the program does change frequently with time, sometimes too frequently! The data is normally represented in the program as objects, we can think of them as real objects, for example a vehicle.

One of the goals in object oriented programming is to maintain the data structure inaccesible to the programmer, and the only way it can be accessed is through methods which are very much alike subroutines. In this way, if the data structure changes for some reason, we do not have to change all methods that manipulate those certain objects. This is what is called encapsulating.

Objects of a certain type are grouped in a class, for example "vehicles". A given object of a given class is called an instance of that object, for example "my car". If a class can de subdivided in further classes, these further ones are named subclasses. For example, "cars" is a subclass of "vehicles".

12.2 Perl representation of objects

An object in Perl is normally represented with a hash or a reference to a hash. Let us create an instance of the vehicle class,

```
$r_my_car= {
          "color"        =>   "red",
          "wheel_number" => 4
};
```

Let us make a method or a subroutine that returns the color of the vehicle and another one that changes its color,

```
sub color {
       # The first argument of the subroutine
       # MUST be the object to change
```

```
        my ($r_vehicle)=@_;

        return ${$r_vehicle}{"color"};
}

sub change_color {
        my ($r_vehicle,$new_color)=@_;

        ${$r_vehicle}{"color"}=$new_color;
}
```

Note how the object must be the first argument to the subroutine. Let us integrate all these functions into a program to see how it all works

```
#!/usr/bin/perl -w

# We make an instance of vehicle
$r_my_car= {
    "color"        =>   "red",
    "wheel_number" => 4
    };


print "The color of my car is ",color($r_my_car),"\n";
print "Let us change the color of my car to green\n";
change_color($r_my_car,"green");
print "The color of my car is now ",color($r_my_car),"\n";


sub color {
    # The first argument of the subroutine
    # MUST be the object to change
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"color"};
}

sub change_color {
    my ($r_vehicle,$new_color)=@_;

    ${$r_vehicle}{"color"}=$new_color;
}
```

In object orienred programming it is customary to separate the subroutines for different classes so they can be called in diferent ways. This is done in Perl with packages, so the subroutines should be defined within a package.

It is also normally the case that objects are created not accessing directly the data representation of the object, but by a method called new.

With all these changes, our program now looks like this,

```
#!/usr/bin/perl -w
```

```
# We make an instance of vehicle
# with the new method
$r_my_car=Vehicle::new("red",4);



print "The color of my car is ",
       Vehicle::color($r_my_car),"\n";
print "Let us change the color of my car to green\n";
Vehicle::change_color($r_my_car,"green");
print "The color of my car is now ",
       Vehicle::color($r_my_car),"\n";


package Vehicle;

sub new {
    my ($color,$wheel_number)=@_;

    my $r_vehicle={
"color"        =>   $color,
"wheel_number" =>   $wheel_number
};

    return $r_vehicle;
}


sub color {
    # The first argument of the subroutine
    # MUST be the object to change
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"color"};
}

sub change_color {
    my ($r_vehicle,$new_color)=@_;

    ${$r_vehicle}{"color"}=$new_color;
}
```

12.3 Polymorphism

Up to now there is no difference between object oriented programming and regular subroutines or functions except for the concepts, but polymorphism will change that.

Supose you want to calculate the tax that a vehicle has to pay. This tax is calculated from the number of wheels of the vehicle by multiplying it by 40 euros for a car and 80 euros

for a truck. We can see now that there are two different types of vehicle classes, it is thus convenient to create two new subclasses of the vehicle class: "cars" and "trucks". We can also implement for them a tax method to calculate the taxes each one of them has to pay. Here is how it could be done

```
#!/usr/bin/perl -w

# We make an instance of Car
# with the new method
$r_my_car=Car::new("red",4);
# And a new instance of Truck
$r_my_truck=Truck::new("yellow",6);


print "The color of my car is ",
       Car::color($r_my_car),"\n";
print "Let us change the color of my car to green\n";
Car::change_color($r_my_car,"green");
print "The color of my car is now ",
       Car::color($r_my_car),"\n";

print "My car has to pay   ",
      Car::tax($r_my_car)," euros in taxes\n";
print "My truck has to pay ",
      Truck::tax($r_my_truck)," euros in taxes\n";


# Defining methods for the Car class
package Car;

sub new {
    my ($color,$wheel_number)=@_;

    my $r_vehicle={
"color"        =>   $color,
"wheel_number" =>   $wheel_number
};

    return $r_vehicle;
}

sub color {
    # The first argument of the subroutine
    # MUST be the object to change
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"color"};
}

sub change_color {
```

```
    my ($r_vehicle,$new_color)=@_;

    ${$r_vehicle}{"color"}=$new_color;
}

sub tax {
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"wheel_number"}*40;
}

# Defining methods for the Truck class
package Truck;

sub new {
    my ($color,$wheel_number)=@_;

    my $r_vehicle={
"color"        =>   $color,
"wheel_number" =>   $wheel_number
};

    return $r_vehicle;
}

sub color {
    # The first argument of the subroutine
    # MUST be the object to change
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"color"};
}

sub change_color {
    my ($r_vehicle,$new_color)=@_;

    ${$r_vehicle}{"color"}=$new_color;
}

sub tax {
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"wheel_number"}*80;
}
```

This is getting confusing. There are several identical subroutines for each class and when we call them we always have to pre-pend the class (or package) name. It would be nice to get rid of this last objection. This is done with the `bless` comand, which binds a newly created instance of an object to its class, just the same as a child, in the christian religion, is given a name when it is baptised.

In object oriented programming it is also customary to use another notation to invoke methods on objects. Normally the arrow notation is used instead. With this notation whatever precedes the arrow is the object to which the method is applied and is taken inside the subroutine as the first argument to it. That is the reason why the object had to be the first argument.

With these new changes, our code is beggining to look more like real object oriented programming

```
#!/usr/bin/perl -w

# We make an instance of Car
# with the new method
$r_my_car=Car::new("red",4);
# And a new instance of Truck
$r_my_truck=Truck::new("yellow",6);


print "The color of my car is ",
       $r_my_car->color,"\n";
print "Let us change the color of my car to green\n";
$r_my_car->change_color("green");
print "The color of my car is now ",
       $r_my_car->color,"\n";

print "My car has to pay   ",
       $r_my_car->tax," euros in taxes\n";
print "My truck has to pay ",
       $r_my_truck->tax," euros in taxes\n";


# Defining methods for the Car class
package Car;

sub new {
    my ($color,$wheel_number)=@_;

    my $r_vehicle={
"color"        =>   $color,
"wheel_number" =>   $wheel_number
};
    bless $r_vehicle, 'Car';
    return $r_vehicle;
}

sub color {
    # The first argument of the subroutine
    # MUST be the object to change
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"color"};
```

```
}

sub change_color {
    my ($r_vehicle,$new_color)=@_;

    ${$r_vehicle}{"color"}=$new_color;
}

sub tax {
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"wheel_number"}*40;
}

# Defining methods for the Truck class
package Truck;

sub new {
    my ($color,$wheel_number)=@_;

    my $r_vehicle={
"color"        =>   $color,
"wheel_number" =>   $wheel_number
};
    bless $r_vehicle, 'Truck';
    return $r_vehicle;
}

sub color {
    # The first argument of the subroutine
    # MUST be the object to change
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"color"};
}

sub change_color {
    my ($r_vehicle,$new_color)=@_;

    ${$r_vehicle}{"color"}=$new_color;
}

sub tax {
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"wheel_number"}*80;
}
```

Note how the tax method is invoked without having to state if it is the tax method for a car or for a truck, this is automatically done by Perl and it is what is called polymorphism.

12.4 Inheritance

We have improved in some ways the above program using polymorphism and the arrow notation, but the program still looks messy. Several methods are defined for each one of the classes which are exactly the same subroutine. There is a better way for this, inheritance.

We can, as we theoretically stated at the beginning of this chapter, define a vehicle class and then make the car and truck class subclasses of the former. In this way we can define methods for the more general vehicle class which are common to all subclasses and only make especific methods for each subclass when they are different. In object oriented parlance we say that the car class inherits from the vehicle class. To tell Perl about this we include tow lines like this after the `package` command for the subclass

```
use class_to_inherit_form;
@ISA=('class_to_inherit_from');
```

and to be able to use the `use` command we must place the class in a `.pm` file.

In Perl you can also invoke the `new` method, and any method for that matter, in a different way, we can first state the name of the method `new` and the its class like so

```
$my_car=new Car("red",4);
```

So, our object oriented program is finally written like this

```
#!/usr/bin/perl -w

# We make an instance of Car
# with the new method
$my_car=new Car ("red",4);
# And a new instance of Truck
$my_truck=new Truck("yellow",6);


print "The color of my car is ",
       $my_car->color,"\n";
print "Let us change the color of my car to green\n";
$my_car->change_color("green");
print "The color of my car is now ",
       $my_car->color,"\n";

print "My car has to pay   ",
       $my_car->tax," euros in taxes\n";
print "My truck has to pay ",
       $my_truck->tax," euros in taxes\n";


# Defining methods for the Car class
package Car;
# Inheriting from Vehicle
use Vehicle;
@ISA=('Vehicle');
```

```
sub tax {
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"wheel_number"}*40;
}

# Defining methods for the Truck class
package Truck;
# Inheriting from Vehicle
use Vehicle;
@ISA=('Vehicle');

sub tax {
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"wheel_number"}*80;
}
```

And the `Vehicle.pm` file which defines the base class

```
# Defining methods for the Vehicle class
package Vehicle;

use Exporter;
@ISA=('Exporter');
@EXPORT=('new','color','change_color');


sub new {
    # We include now $type because we changed
    # the way we call the routine and
    # to use it in the bless statement
    my ($type,$color,$wheel_number)=@_;

    my $r_vehicle={
"color"        =>   $color,
"wheel_number" =>   $wheel_number
};
    bless $r_vehicle, $type;
    return $r_vehicle;
}

sub color {
    # The first argument of the subroutine
    # MUST be the object to change
    my ($r_vehicle)=@_;

    return ${$r_vehicle}{"color"};
}
```

```
sub change_color {
    my ($r_vehicle,$new_color)=@_;

    ${$r_vehicle}{"color"}=$new_color;
}
```

12.5 Exercise

Save the previous complex program in a safe place and rewrite using the `Math::Complex` package.

Chapter 13

Windows widgets with Perl: Perl/Tk

Perl/Tk is a tool to make a graphical user interface (GUI) easily. It is very well desingned and very easy to use. Here we will just get a feeling of it.

13.1 Our first Perl/Tk program

All Perl/Tk interfaces have the following structure:

1. Creating the root window. This is the main window on which all other widgets will be places.
2. Creating one or more widgets. A widget is an object which has a reality in the windows environment, it normally reacts to a given action from the user.
3. Placing the widget on the root window or on another widget. This is what is known as geometry management. Widgets are structured in a hierarchical manner. Some widgets can contain others, etc...
4. Start the event loop. Widget programming is an event driven techinque, which means that after we initiate the event loop the program waits until something happens to the widgets, in which case it reacts accordingly.

Let us make now a "Hello world" program.

```
#!/usr/bin/perl -w

use Tk;

# Step 1), we create the main window
$top=MainWindow->new();
# Now we change a property of the main window
$top->title("Hello again");

# Step 2), we create a widget, which
```

```
# will "hang" from the root window
# In this case it is a Label widget
# with several properties
$lab=$top->Label( text   => 'Hello world!',
  relief => 'groove',
  width  =>  10,
  height =>  5,
               );

# Step 3), we place the widget on the
# parent widget, which is the root window
$lab->pack();

# Step 4), we start the main loop
MainLoop();
```

To know more about the label widget type `man Tk::Label`.

13.2 More widgets

We have seen the label widget. There are many more, but here we will just see the button and the text entry widget.

13.2.1 Button widget

As its name indicates, this creates a button which reacts when it is pressed.

Let us modify the above program so that we also have a button.

```
#!/usr/bin/perl -w

use Tk;

# Step 1), we create the main window
$top=MainWindow->new();
# Now we change a property of the main window
$top->title("Hello again");

# Step 2), we create a widget, which
# will "hang" from the root window
$lab=$top->Label(   text     => 'Hello world!',
    relief   => 'groove',
    width    =>  10,
    height   =>  5
);

$but=$top->Button(  text     => 'Press me!',
    width    =>  10,
    height   =>  5,
    # We add a reference
    # to a subroutine to which
```

```
                    # the action responds
    command =>  \&do_task
                );

# Step 3), we place the widget on the
# parent widget, which is the root window
$lab->pack();
$but->pack();

# Step 4), we start the main loop
MainLoop();

sub do_task {
  print "Hello again\n";
  exit;
}
```

In this case we have placed or packed both widgets before calling the event loop, but it is more normal to do so just after creating the widgets. We will see this in the following example.

To know more about the button widget type `man Tk::Button`.

13.2.2 Text entry widget

The text entry widget admits a text as input. This value is automatically stored in a variable. It is used like this

```
use Tk;

# We create the main window
$top=MainWindow->new();
# Now we change a property of the main window
$top->title("Hello again");

# We create widgets, which
# will "hang" from the root window
# and place them on the root window
$top->Label(   text     => 'Hello world!',
    relief   => 'groove',
    width    =>  10,
    height   =>  5
)->pack();

$top->Button(  text     => 'Press me!',
    width    =>  10,
    height   =>  5,
    # We add a reference
    # to a subroutine to which
                    # the action responds
    command =>  \&do_task
```

```
               )->pack();

$top->Entry( width   => 10,
                 textvariable => \$text
)->pack();

# We start the main loop
MainLoop();

sub do_task {
  print "Hello again\n";
  print "Text entry has $text\n";
  exit;
}
```

For more information on the entry widget type `man Tk::Entry`.
To have a general view of Perl/Tk type `widget`.

13.3 Exercise

Write a program which accepts four entries which will be the four corners of a square on the complex plane. Add another entry which will be the length of the side of the square as number of points. Then add a button so that when it is pressed the calculation from the Newton root finding method is done with the above values.

Part II

PDL

Chapter 14

Introduction to PDL

14.1 What is PDL?

PDL is a numerical computation language based on Perl. It can make fast matrix calculations in a simple manner.

14.2 Advantages and disadvantages of PDL

The main advantages of PDL are:

- It is free and it is Open Source software.
- It permits matrix and several dimensional calculations, it does not just stop on the second dimension, but allows more dimensions to be involved in the calculations. In this sense it is very well thought out and scales up very well.
- It is very fast.
- The graphical interface is very good. The two dimensional graphics are based on PGPLOT which is very adecuate both for publication quality figures and for everyday plotting. The three dimensional graphics, based on the MESA library, still does not have a publication quality, but is is very good for plotting three dimensional parameters in a simple way and allows real time rotation of the plotted data.

The main disadvantages of PDL are:

- The available numerical library to make calculations is not very broad. This feature is changing rapidly as more numerical routines like GSL or SLATEC are being inserted in PDL.
- It does not support calculation with complex numbers natively. In version 2.004 of PDL a complex package has been added, but it just enables simple complex number arithmetic.
- Programming with PDL needs a more abstract thinking than other languages to exploit its full capabilities. Also, in some cases, there is no easy way to use commands and a more complicated workaround has to be used.

14.3 Using PDL

There are two ways of using PDL, from a Perl program and interactively. If we want to use it from a program just include the `use PDL;` directive at the beginning of the program. To make use of it interactively type `perldl`.

14.4 Getting help

Help from PDL can be done in several ways.

Interactively we can use the `help` command. To know how to use it type `help help`. To get a manual page from a given command type `help command`. To get help from a given module type `help module_name` and to type a given manual type `help manual_name`. The `apropos` command is also available. If you type `apropos keyword`, this gives you a listing of the commands that has this keyword in its documentation database. To list the available on line manuals type `apropos manual`, to list the available modules type `apropos modules` and lo list all possible commands type `apropos . .`

There are also several HTML pages available where PDL has been installed, normally at `/usr/lib/perl5/PDL/HtmlDocs`

14.5 How to use the rest of this manual

Each of the following chapters is divided in a "Basic usage" and an "Advanced usage" section. On a first read skip the latter one. The "Basic usage" section should be enough to get a general feeling of the language. The "Advanced usage" section is added for completeness. To get more information about each command just type `help command`.

The examples below are written in a tutorial way so that a line with a preceding

```
perldl>
```

are lines that you should type in as an exercise. Lines that do not start with this prompt are the output of the program, DO NOT type this lines into the PDL command line interface. For example in

```
perldl> p 3
3
```

you type the command `p 3` and you get as output `3`.

Chapter 15

Creating PDLs

15.1 Introduction

PDLs or piddles are the basic building blocks of PDL. They are Perl scalars but they do not hold just a numerical value like in regular Perl. They represent a vector, matrix, or any n–dimensional array. In this chapter we will see how to create them in different ways.

15.2 Basic usage

15.2.1 Simple piddles

To create a piddle you can use the `pdl` command. This command accepts a Perl scalar, array reference or an array. For example, to create a vector or one dimensional piddle which has values that go from 7 to 10 type

```
perldl> $a=pdl([7..10]);
```

To print the contents of a PDL use the `print` command

```
perldl> print $a;
[7 8 9 10]
```

To type less, in the command line interface you can omit the trailing semicolon and the `print` command can be substituted by just the letter `p`. Note that this is not so when you type in the commands in a Perl script. So the following is equivalent to the previous command,

```
perldl> p $a
[7 8 9 10]
```

Let us now create a piddle with any values

```
perldl> $b=pdl([4,2,8,1,6])
perldl> p $b
[4 2 8 1 6]
```

Let us make a two dimensional one

```
perldl> $b=pdl([ [3,4,5], [2,8,3] ])
perldl> p $b

[
 [3 4 5]
 [2 8 3]
]
```

15.2.2 More complex piddles

The `sequence` command creates an array with a sequence of values. It accepts as argument an array especifying the length of each of the dimensions the piddle has. For example this line creates a vector with values ranging from 0 to 15.

```
perldl> $d=sequence(15);
perldl> p $d
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14]
```

To create a $3x4$ matrix

```
perldl> $m=sequence(3,4);
perldl> p $m

[
 [ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
]
```

The `zeroes` command makes a piddle filled with ceros with dimensions given by its arguments.

To create a $6x7$ matrix type

```
perldl> $r=zeroes(6,7);
perldl> p $r

[
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
 [0 0 0 0 0 0]
]
```

```
The \verb/ones/ command is very similar
```

```
to the \verb/zeroes/ command and makes a piddle filled with ones
with dimensions given by its arguments.

\begin{verbatim}
perldl> $r=ones(6,7);
perldl> p $r

[
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
 [1 1 1 1 1 1]
]
```

What if we want to have a piddle of any dimensions with ascending values in the x axis? For that it is useful the `xvals` command, which accepts as argument a piddle with identical size to the one we want to get

```
perldl> $x=xvals(3,2);
perldl> p $x
 [
  [0 1 2]
  [0 1 2]
 ]
```

A similar command can be used to span the y axis

```
perldl> $y=yvals(3,2);
perldl> p $y
 [
  [0 0 0]
  [1 1 1]
 ]
```

The above commands span integer values beginnig at cero. But sometimes we want more flexibility and we want to start at a given value x value and end at another one. This can be accomplished with the `xlinvals` command.

```
perldl> $a=zeroes(3,2);
perldl> $x=$a->xlinvals(0.5,1.5);
perldl> p $x
 [
  [0.5   1 1.5]
  [0.5   1 1.5]
 ]
```

Note that this command accepts three arguments, a piddle with dimensions identical to the piddle we want to create, a starting value and a finish value. Note the object oriented

programming notation of this command. Recall that the object or variable that is placed before the -> signs is really the first argument to the function. So an equivalent way to write the above is

```
perldl> $a=zeroes(3,2);
perldl> $x=xlinvals($a,0.5,1.5);
perldl> p $x
 [
  [0.5   1 1.5]
  [0.5   1 1.5]
 ]
```

In the following we will use the notation that seems more intuitive for the command indistinctly.

There is an analogous command for the y axis values

```
perldl> $a=zeroes(2,3)
perldl> $y=$a->ylinvals(0.3,1.3);
perldl> p $y
 [
  [0.3 0.3]
  [0.8 0.8]
  [1.3 1.3]
 ]
```

Also, from the object oriented notation, just as we can call a function with arguments that are the output of another function like this

```
perldl> $y=ylinvals(zeroes(2,3),0.3,1.3);
perldl> p $y
 [
  [0.3 0.3]
  [0.8 0.8]
  [1.3 1.3]
 ]
```

we can concatenate several commands together with arrow signs, -> like this

```
perldl> $y=zeroes(2,3)->ylinvals(0.3,1.3);
perldl> p $y
 [
  [0.3 0.3]
  [0.8 0.8]
  [1.3 1.3]
 ]
```

Again any notation will be used indistinctly.

The `xlinvals` and `ylinvals` commands are useful for creating piddles to be used later either as input for the x and y axis values for plotting or as input for a given function.

The following example calculates the values of a two dimensional gaussian function

```
perldl> $x=zeroes(20,20)->xlinvals(-0.5,0.5);
perldl> $y=zeroes(20,20)->ylinvals(-0.5,0.5);
perldl> $gaus=exp( -($x**2)/0.05 - ($y**2)/0.02 );
```

We can also create random numbers between cero and one in a piddle of a given dimension with the `random` command

```
perldl> $r=random(2,3)
perldl> p $r
[
 [0.84018772 0.39438293]
 [0.78309922 0.79844003]
 [0.91164736 0.19755137]
]
```

15.3 Advanced usage

15.3.1 Data types

PDL can make the piddles with several different data types. For example, to create a piddle filled with `doubles` with the `xvals` we can type

```
perldl> $x=xvals(double,2,3)
```

The available types are `byte`, `short`,`ushort`,`long`, `float`,`double`.

It is also possible to transform one type into another using the above types as functions, like in the following example where we change the type of a piddle to a long integer.

```
perldl> $x=zeroes(3,3)->xlinvals(-1.5,1)
perldl> p $x
[
 [ -1.5 -0.25     1]
 [ -1.5 -0.25     1]
 [ -1.5 -0.25     1]
]

perldl> $i=long($x)
perldl> p $i
[
 [-1  0  1]
 [-1  0  1]
 [-1  0  1]
]
```

To obtain more information about data types use

```
help Datatype_conversions
```

15.3.2 More creators of piddles

For more information about these commands type

```
help command
```

- `zvals`. This command is like the `xvals` and `yvals` command except that it works along the z axis.
- `axisvals`. An analog command to the `xvals`, `yvals` and `zvals` but it can applied to any dimension.
- `rvals`. This command fills a piddle with radial distance values from some centre.
- `diagonal`. Creates a diagonal matrix
- `topdl` This command creates a piddle just like the `pdl` command, except that the if the argument is a piddle the output will be that same piddle. It is used to ansure an argument to some function is really a piddle.

Chapter 16

Arithmetic

16.1 Basic usage

PDL accepts several common known arithmetic operators and functions. The following symbols add, substract, multiply and divide several piddles

```
+ - * /
```

With these operators, just like all the rest included in this chapter, PDL acts on a per element basis. If for example, we multiply two PDLs of the same sizes and dimensions we get an equivalent PDL with each element being the product of each one of the elements of the two.

```
perldl> $a=pdl([1,2])
perldl> $b=pdl([3,4])
perldl> p $a*$b

[3 8]
```

We can for example calculate and plot the points of a parabola as shown in Fig. 16.1

```
perldl> $x=zeroes(30)->xlinvals(-1,1);
perldl> $y=$x**2;
perldl> line($x,$y)
```

The exponetiation operator is the same as in FORTRAN **.
There are also relational and logical operators, very similar to the C ones

```
> < >= <= == != !
```

Note that this operators also act on a per element basis. That is, if a piddle is compared with a number, the results is a piddle of the same size as the original one.

```
perldl> $a=pdl([0,1,2,3,4])
perldl> p $a > 2

[0 0 0 1 1]
```

The mathematical functions also act on a per element basis, some of them are the following

```
sin log abs atan2 sqrt cos exp
```

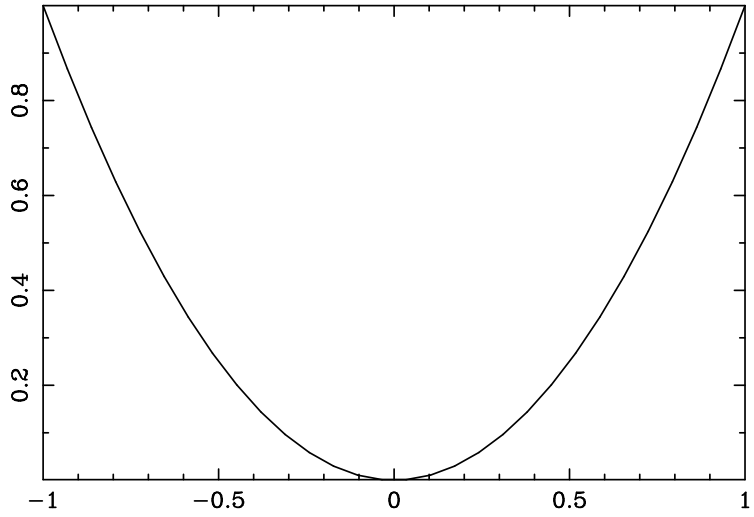

Figure 16.1: A plotted parabola.

16.2 Advanced usage

There are also the bitwise operators

```
>> << & | ^
```

and other specialized ones

```
<=> % ~
```

Chapter 17

Getting properties of piddles

17.1 Basic usage

We shall see now some interesting functions to get properties of piddles

- `nelem`

  This function returns the number of elements in a piddle.

```
perldl> $a=zeroes(10);
perldl> p nelem($a)
10
perldl> $b=ones(10,10);
perldl> p nelem($b)
100
```

- `dims`

  This function returns the dimensions of a PDL as a Perl list or array.

```
perldl> $a=zeroes(10);
perldl> @da=dims($a)
perldl> p @da
10
perldl> $b=ones(10,10);
perldl> @db=dims($b)
perldl> p @db
10 10
```

- `PDL::getndims`

  Returns the number of dimensions of a piddle.

```
perldl> $b=ones(10,10);
perldl> print PDL::getndims($b);
2
```

- `at`

  Returns the value of a piddle for a given index position. For example to get the element with indexes `(1,1)` we can do

  ```
  perldl> $b=pdl [ [1,2,3], [4,5,6] ];
  perldl> print $b->at(1,1);
  5
  ```

  Note that indices on a piddle start at 0, just like the regular Perl array notation. Also note that the regular Perl array notation is NOT valid for piddles. The following is wrong for a `$b` piddle,

  ```
  perldl> print $b[1][1];
  ```

Chapter 18

Plotting

18.1 Basic usage

18.1.1 2D plotting

To plot functions we will use the powerful PGPLOT library, which can be easily called from PDL. To be able to use it we have to load the PGPLOT package,

```
perldl> use PDL::Graphics::PGPLOT;
```

To plot a 2D function we can use the `line` command. The following example will draw a line

```
perldl> $x=pdl([0,1,2])
perldl> $y=pdl([4,5,6])
perldl> line($x,$y)
```

The first time we use a PGPLOT command to make a plot, and if we have not selected a given decive before, a question will appear asking us which device type we want. To see all available device types write `?` and press `<ENTER>`. If we just want to plot on X windows just type `/xserve`.

There are more ways to plot functions, let us use as an example the above cited parabola,

```
perldl> $x=zeroes(30)->xlinvals(-1,1);
perldl> $y=$x**2;
```

Let us now see in how many different ways can we plot this function.

- **line**. As we have seen previously we can draw this function with this command obtaining the plot shown in Fig. 16.1.

  ```
  perldl> line($x,$y)
  ```

  As stated above, if the device has not been selected before we will have to do so now by answering `/xserve` to the device question.

- **points**. We can also plot the points separately obtaining the plot shown in Fig. **??**.

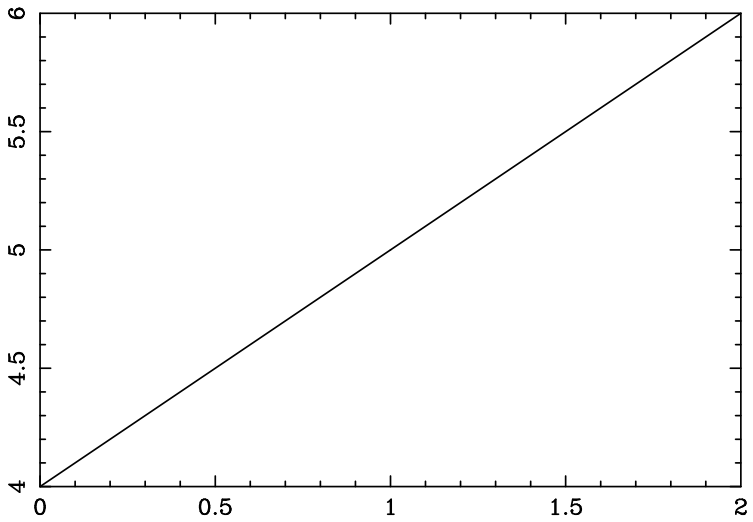

Figure 18.1: A plotted line.

```
perldl> points($x,$y)
```

- **errb**. We can also plot the points with error bars. Let us assume that the error in x is constant and $\epsilon(x) = 0.1$, then the error in y can be expressed as $\epsilon(y) = 2x\epsilon(x)$. To plot the points with error bars in the y axis we can do,

  ```
  perldl> errb($x,$y,2*$x*0.1)
  ```

  This result is shown in Fig. **??**

- **bin**. To plot the parabola with a staircase look

  ```
  perldl> bin($x,$y)
  ```

  this is shown in Fig **??**.

- **poly**. We can easily draw a polygon with this command. In this example a triangle will be drawn

  ```
  perldl> $xx=pdl([1,2,3]);
  perldl> $yy=pdl([0,1,0]);
  perldl> poly($xx,$yy);
  ```

  like is shown in Fig. **??**.

18.1.2 3D plotting

There are two different ways to make 3D plots. One of them is to use the same PGPLOT package as we did with the 2D plots, the other one is to use the TriD package.

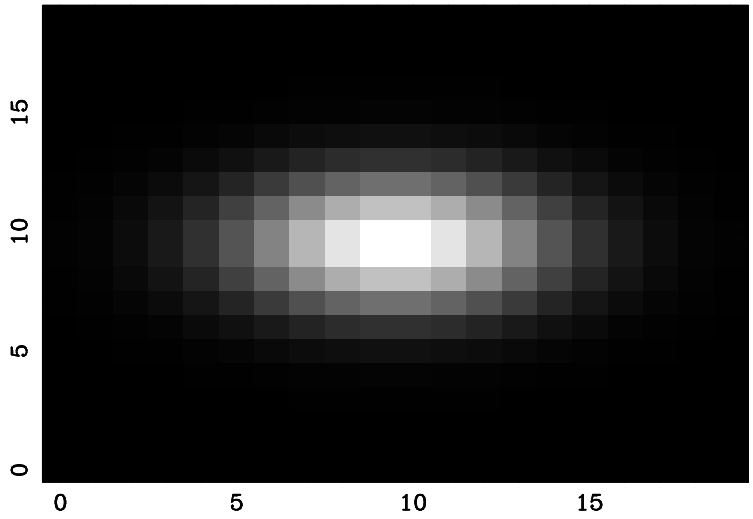

Figure 18.2: Gaussian function plotted on a 2D plane.

18.1.2.1 3D plotting with PGPLOT

We will plot a two dimensional gaussian function in several ways using the PGPLOT package

```
perldl> use PDL::Graphics::PGPLOT
perldl> $x=zeroes(20,20)->xlinvals(-0.5,0.5);
perldl> $y=zeroes(20,20)->ylinvals(-0.5,0.5);
perldl> $gaus=exp( -($x**2)/0.05 - ($y**2)/0.02 );
```

- **imag**. To make a 2D plot of an image with grayscale we can use the `imag` command. We can plot the above gaussian with

  ```
  perldl> imag($gaus)
  ```

  which should show a graph like Fig. 18.4.

- **cont**. A contour map can also be obtained like this

  ```
  perldl> cont($gaus)
  ```

  which is shown in Fig. **??**

- **hi2d**. To plot a mesh with PGPLOT we can use this command. It is also worth noting that if you have TriD working you will probably be better off using **mesh3d** or a similar command. The result shown in Fig. **??** can be achieved like this

  ```
  perldl> hi2d($gaus)
  ```

- **vect**. To plot a vector field with arrows we can use this command. If we assume that the x components of the vectors are in the above formed `$gaus` matrix and the y components are given by `-$gaus` we can obtain the vector field as shown in Fig. **??** with

  ```
  vect($gaus,-$gaus)
  ```

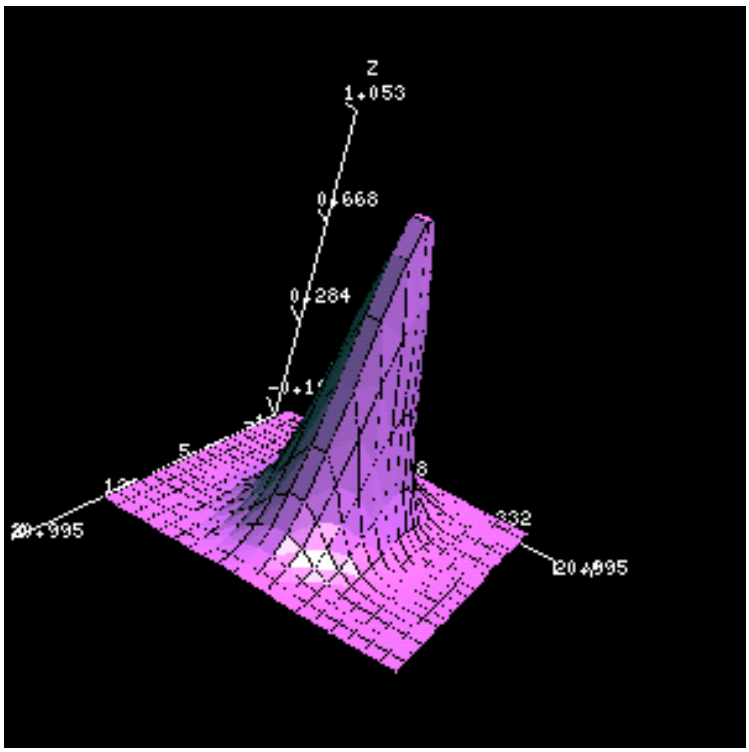

Figure 18.3: 3D gaussian function.

18.1.2.2 3D plotting with TriD

The three dimensional plotting is not loaded into PDL at startup so we must first load it as a module

```
perldl> use PDL::Graphics::TriD;
```

To plot 3D funtcions we will use the `imag3d` command. We can for example plot the above gaussian function like this (remember to write the above `use` statement before)

```
perldl> $x=zeroes(20,20)->xlinvals(-0.5,0.5);
perldl> $y=zeroes(20,20)->ylinvals(-0.5,0.5);
perldl> $gaus=exp( -($x**2)/0.05 - ($y**2)/0.02 );
perldl> imag3d([$gaus])
```

Something like Fig. 18.3 should appear.

Note that the prompt does not return back. This is because we can modify the plotted function in place. If we put the mouse pointer on the graph, we can rotate it by pressing the left mouse button and moving the mouse. We can also zoom in and out by pressing the right mouse button and moving the mouse. To exit this state type `q` on the graph. The prompt will return. After that do not kill the graphics window if we want to keep the PDL command line interface.

The above example of the gaussian function can be plotted in several other ways which are shown below.

- **imag3d**. We have already seen this command. It will show what is seen in Fig. 18.3, that is, a solid surface of the function.

```
perldl> imag3d([$gaus])
```

- **mesh3d**. This function will show the same gaussian function as a mesh as is shown in Fig. **??**.

```
perldl> mesh3d([$gaus])
```

- **points3d**. The same surface can also be plotted as separate points in space as shown in Fig. **??**. This command accepts three arguments, the x coordinates of the points, the y coordinates and the z coordinates.

```
perldl> points3d([$x,$y,$gaus])
```

- **line3d**. Lines can also be plotted in the 3D space. We will use the example that comes in the demo of PDL obtaining a nice spiral as shown in Fig. **??**.

```
# Number of subdivisions for lines / surfaces.
$size = 25

$cz = (xvals zeroes $size+1) / $size;  # interval 0..1
$cx = sin($cz*12.6);    # Corkscrew
$cy = cos($cz*12.6);
line3d [$cx,$cy,$cz];   # Draw a line
```

18.2 Advanced usage

18.2.1 2D plotting

Making 2D plots with PGPLOT is very powerful and almost any figure can be made with it. To obtain more information on the different type plots that can be achieved with it try having a look at the very informative PGPLOT demo by typing

```
perldl> demo pgplot
```

or see the help pages of the commands **line**, **points**, **errb**, **bin** and **poly** by typing for example,

```
help points
```

Now several commands that control the way figures are plotted will be shown.

- **dev**. With this command we can send the output to a file instead of the monitor. For example, to write the figure on a nice PostScript file named `file.ps` we can use this command before any other plotting one

```
dev('file.ps/ps');
```

To see the list of the available devices in which we can plot draw any graph with PGPLOT without especifying a determined device and we will be asked which device we want. Answering with a `?` will give us a list of all available devices.

- **hold**. This command puts the graph on hold. This means that any other plotting PGPLOT command will be drawn on top of the graph we have without erasing anything. This option is good for including several different things in a graph.
- **release**. This command frees the previous hold on the graph. The next figure will be drawn after erasing all previous graphs.
- **env**. This command defines the region in physical space which will be plotted and puts the graph on hold. In this way if we want to plot a region between 1 and 5 in the x axis and 1 and 25 in the y axis we can use

  ```
  env(1,5,1,25)
  ```

  for a practical use of this command see the next item, "Directly using PGPLOT commands".
- **Directly using PGPLOT commands**. If we want to make sofisticated plots it is very probable that we will end up using direct PGPLOT commands. The direct PGPLOT calls can be used from Perl (not PDL) directly. To use them we must load the PGPLOT module first with

  ```
  perldl> use PGPLOT
  ```

  A listing of all the PGPLOT commands can be obtained in HTML form from the PGPLOT installation. In Linux systems this is located at `/usr/share/doc/pgplot5` To call them we must substitute single FORTRAN numbers by Perl scalars and FORTRAN matrices with matrix Perl references.

  For example to plot five points with a direct call to PGPLOT we can use the `PGPT` PGPLOT subroutine. The help page from PGPLOT states that this FORTRAN subroutine should be called like this

  ```
      SUBROUTINE PGPT (N, XPTS, YPTS, SYMBOL)
      INTEGER N
      REAL XPTS(*), YPTS(*)
      INTEGER SYMBOL

  Arguments:
   N      (input)  : number of points to mark.
   XPTS   (input)  : world x-coordinates of the points.
   YPTS   (input)  : world y-coordinates of the points.
   SYMBOL (input)  : code number of the symbol to be drawn at each
                     point:
  ```

  The integer numbers can be substituted by Perl scalars and the FORTRAN vectors by Perl array references. Note that we must also supply the number of points as a scalar. The direct call to obtain the plot in Fig. **??** would be done like this

```
perldl> use PGPLOT
perldl> dev('/xserve')
perldl> @x=(1,2,3,4,5)
perldl> @y=(1,4,9,16,25)
perldl> env(1,5,1,25)
perldl> pgpt(5,\@x,\@y,0)
```

18.2.2 3D plotting

18.2.2.1 3D plotting with PGPLOT

- **ctab**. The colors corresponding to each level in the above plotted gaussian image can be changed with this command. Let us assume that we have the following normalized levels whose representation we want to change,

  ```
  perldl> $x=zeroes(20,20)->xlinvals(-0.5,0.5);
  perldl> $y=zeroes(20,20)->ylinvals(-0.5,0.5);
  perldl> $gaus=exp( -($x**2)/0.05 - ($y**2)/0.02 );
  perldl> imag($gaus)
  perldl> $levels=zeroes(11)->xlinvals(0,1)
  perldl> p $levels
  [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1]
  ```

  We can now assign the highest red color and the highest green color to the first value and the inversley for the blue color,

  ```
  perldl> $red=zeroes(11)->xlinvals(1,0)
  perldl> $green=zeroes(11)->xlinvals(1,0)
  perldl> $blue=zeroes(11)->xlinvals(0,1)
  ```

  Now we change the color table

  ```
  perldl> ctab($levels,$red,$green,$blue)
  ```

  and redisplay the image

  ```
  perldl> imag($gaus)
  ```

  obtaining something like is shown in Fig. **??**.

imagrgb.
hold3d
release3d
keeptwiddling3d
nokeeptwiddling3d
imagrgb3d
grabpic3d

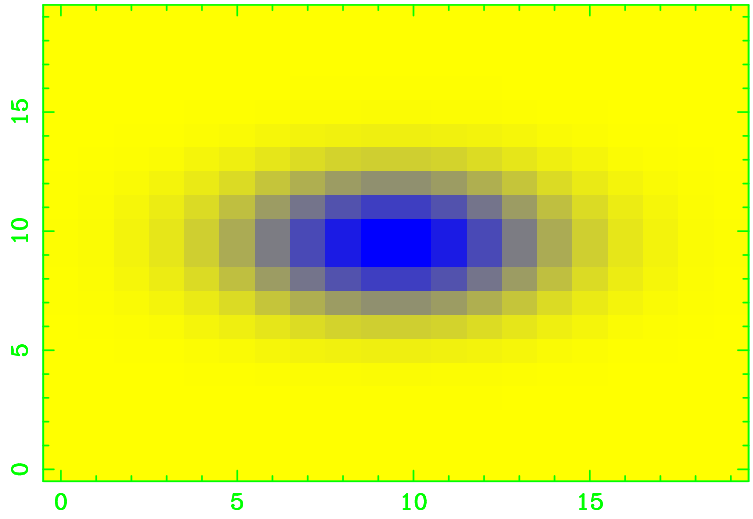

Figure 18.4: Gaussian function plotted on a 2D plane with the colour table changed.

Chapter 19

Modifying piddles

19.1 Basic usage

19.1.1 set

Changes a single value inside a piddle

```
$a=zeroes(10,10);
# Changing the (1,3) component
# to 5
$a->set(1,3,5);
```

19.1.2 slice

xchg

19.1.3 list

Convert the PDL into a Perl array or list

```
$a=zeroes(10,10);
@a_list=list($a);
```

19.1.4 listindices

Converts the indices of a one dimensional piddle into a Perl array or list

```
$a=zeroes(5);
# This returns
# (0,1,2,3,4)
@a=listindices($a);
```

19.2 clip

This function clips a piddles with a certain upper a lower bound

```
$b=pdl [ [1000, 2, 3], [ 1, -1000, 4] ];
$c=$b->clip(0,10);
# $c now contains
# [
#  [10  2  3]
#  [ 1  0  4]
# ]
```

19.2.1 badmask

Signature: (a(); b(); [o]c())

Clears all infs and nans in a to the corresponding value in b

19.3 Advanced usage

19.3.1 dummy

19.3.2 hclip

19.3.3 lclip

19.3.4 one2nd

19.3.5 mslice

19.3.6 reshape

19.3.7 convert

Chapter 20

Combining several PDLs

20.1 Basic usage

20.1.1 append

Append two piddles by concatenating them over their first dimension.

```
$a=zeroes(3);
$b=ones(2);
# $c now contains [0 0 0 1 1]
$c=append($a,$b);
```

20.1.2 cat

Concatenates N piddles of the same sizes and returns a piddle with one more dimension of length N+1.

```
$a=zeroes(3);
$b=ones(3);
# This way we get
# [
#  [0 0 0]
#  [1 1 1]
# ]
$c=cat($a,$b);
```

20.1.3 dog

Takes a single N dimensional piddles and splits it into a N-1 dimensional piddles.

```
$a=zeroes(3);
$b=ones(3);
$c=cat($a,$b);
# Now $a == $d
# and $b == $e
($d,$e)=dog($c);
```

Chapter 21

Matrix operations

21.1 Basic usage

21.1.1 Matrix multiplication

```
$c=$a x $b;
```

21.1.2 matinv

```
use PDL::Slatec;
$inv = matinv($mat)
```

21.1.3 eigsys

```
use PDL::Slatec;
($eigvals,$eigvecs) = eigsys($mat)
```

`eigens` Eigenvalues and -vectors of a matrix in lower triangular form
`simq` Solution of simultaneous linear equations.
`squaretotri` Convert a symmetric square matrix to triangular vector storage.

21.2 Advanced usage

21.2.1 inner

21.2.2 outer

21.2.3 innerwt

21.2.4 inner2

21.2.5 inner2d

21.2.6 inner2t

Chapter 22

Descriptive statistics and internal piddle operations

22.1 Basic usage

sumover
- intover
- cumsumover
- prodover
- cumprodover
- sum
- min
- max
- median
- oddmedian
- minmax
- qsort
- qsorti
- minimum
- minimum_ind
- maximum
- maximum_ind
- minmaximum
- wtstat
- hist
- histogram2d
- stats

22.2 Advanced usage

minimum_n_ind
- maximaum_n_ind

Chapter 23

Piddle selection

which
    where
    index

Chapter 24

Interpolation

24.1 Basic usage

interpol

24.2 Advanced usage

vsearch

Chapter 25

Input output functions

rfits
- wfits
- rcols
- wcols
- rgrep
- rdsa
- rasc
- isbigendian
- rcube
- bswap2
- bswap4
- bswap8
- readfraw
- writefraw
- mapfraw
- maptextfraw
- readflex
- writeflex
- rpiccan
- wpiccan
- rpic
- wpic
- wmpeg
- rpnm
- wpnm
- rndf
- wndf
- propndfx

Chapter 26

Math functions

The following math functions are available within PDL

- `acos` Standard arc cosine trigonometric function.
- `asin` Standard arc sine trigonometric function.
- `atan` Standard arc tangent trigonometric function.
- `cosh` Standard hyperbolic cosine function.
- `sinh` Standard hyperbolic sine function.
- `tan` Standard tangent trigonometric functions.
- `tanh` Standard hyperbolic tangent functions.
- `ceil` Returns the smallest integral value not less than the argument, that is, it rounds the argument upward to the nearest integer.
- `floor` Returns the largest integral value not greater than the argument, that is, it rounds the argument downwards to the nearest integer.
- `rint` Rounds the argument to the closest integer.
- `pow` Power funtcion, raises a number to the specified power. It is a synonym for `**`.
- `acosh` Standard hyperbolic cosine function.
- `asinh` Standard hyperbolic sine function.
- `atanh` Standard hyperbolic tangent function.
- `erf` The error function.
- `erfc` The complement of the error function.
- `bessj0` Standard Bessel J_0 function.
- `bessj1` Standard Bessel J_1 function.
- `bessy0` Standard Bessel Y_0 function.

- `bessy1` Standard Bessel Y_1 function.
- `bessjn` Standard Bessel J_n function.
- `bessyn` Standard Bessel Y_n function.
- `lgamma` The log gamma function. This returns 2 piddles – the first set gives the log(gamma) values, while the second set, of integer values, gives the sign of the gamma function. This is useful for determining factorials, amongst other things.
- `erfi` The inverse of the error function.

Chapter 27

Image manipulation

conv2d
- med2d
- patch2d
- max2d_ind
- centroid2d
- cc8compt
- convolve
- ninterpol
- rebin
- cquant
- interlrgb
- bytescl

Chapter 28

Fourier analysis

fft
- ifft
- realfft
- fftnd
- ifftnd
- fftconvolve
- convmath
- cmul
- cdiv