Journal of Statistical Software

May 2004, Volume 11, Issue 1. http://www.jstatsoft.org/

Using Perl for Statistics: Data Processing and Statistical Computing

Giovanni Baiocchi
University of Durham

Abstract

In this paper we show how Perl, an expressive and extensible high-level programming language, with network and object-oriented programming support, can be used in processing data for statistics and statistical computing. The paper is organized in two parts. In Part I, we introduce the Perl programming language, with particular emphasis on the features that distinguish it from conventional languages. Then, using practical examples, we demonstrate how Perl's distinguishing features make it particularly well suited to perform labor intensive and sophisticated tasks ranging from the preparation of data to the writing of statistical reports. In Part II we show how Perl can be extended to perform statistical computations using modules and by "embedding" specialized statistical applications. We provide example on how Perl can be used to do simple statistical analyses, perform complex statistical computations involving matrix algebra and numerical optimization, and make statistical computations more easily reproducible. We also investigate the numerical and statistical reliability of various Perl statistical modules. Important computing issues such as ease of use, speed of calculation, and efficient memory usage, are also considered.

Keywords: Perl, data processing, statistical computing, reproducibility.

1. Introduction

Statistics is often defined as the collection, presentation, characterization, analysis, and interpretation of data. The large availability of powerful computers and sophisticated software has meant that many of these activities are routinely solved by statisticians using powerful statistical systems.

Statisticians use computers, not only for computations on data, but also for simple, mechanical operations such as searching for information, collecting and storing data, changing the format of data, validating data, post-processing output from statistical applications, writing reports, collaborating with other researchers, and in disseminating the final results through

the Internet. These operations typically require using a variety of data processing, communication, and other software tools.

In this paper, we demonstrate how Perl, a high-level programming language, provides a powerful and flexible computing environment in which the statistician can integrate many of the above mentioned activities. We will also show how Perl can facilitate the solution of practical problems faced by the statistician, that are typically not addressed using computer programs, and that often require considerable labor intensive and error prone manual intervention.

Perl, an acronym for *Practical Extraction and Report Language*,[1] is a cross-platform, high-level, open source programming language, originally designed and implemented by Larry Wall.

Perl can be downloaded from the WWW's Comprehensive Perl Archive Network (CPAN) located at http://www.cpan.org/, where the source package, modules, documentation, and links to binaries distributions of Perl are available for various platforms, including Win32, Mac, and Unix/Linux.[2] Perl is free, and is released under either the GNU GPL license or the less restrictive Artistic License.[3]

The Perl language originally incorporated and extended some of the best features of the C programming language, and from software tools such as sed, awk, grep, and the Unix shell. Perl has considerably evolved since its beginning and now features a full range of network and object-oriented capabilities.

The paper is organized as follows. In Section 2 we present the typographical and style conventions used in this paper. The rest of the paper is organized into two parts. In Part I we introduce Perl as a high-level language that can be used to process data for statistics and can complement existing tools by functioning as a "glue" between different applications. This part is organized as follows. Section 3 introduces the distinction between conventional programming languages and scripting languages. An overview of the available documentation and other useful resources on Perl is provided in Section 4. Section 5 introduces Perl's basic programming features with particular emphasis on the features that distinguish Perl from conventional programming languages. Sections 7 demonstrates, with a practical example, how Perl, viewed as a scripting language, can be used for data processing in statistics. Section 8 introduces Perl's object-oriented language features and demonstrates how Perl's data processing capabilities can be augmented through extension modules. Section 9 illustrates how Perl modules can provide an intuitive interface to the WWW, making this expanding environment readily accessible within statistical applications thus creating opportunities for innovations in the way statistics is practiced and taught.

In Part II we illustrate how Perl can be extended to perform simple statistical analyses, numerical linear algebraic computations, and efficient statistical computing. Part II is organized as follows. In Section 11 we illustrate how Perl scripts can be used to collect commands that can reproduce the computational results of a statistical project in its entirety. The next two sections illustrate the methodology used to empirically assess the modules surveyed. Specifically, Section 12 reviews the methodology used to assess the numerical reliability of modules

[1] Though many Perl programmer like to think it stands for "Pathologically Eclectic Rubbish Lister."

[2] We used ActivePerl release 5.8.0.806, which is based on Perl version 5.8, the standard Win32 release available at the time of writing the present paper. The code has also being tested on platforms running the Solaris and the Linux operating system. The hardware was a 600 MHz dual Pentium III with 256 MB of RAM running on Windows NT.

[3] For details on the Perl license consult the GNU Project's home page at http://www.gnu.org/.

and Section 13 provides a quick overview on how speed benchmarking can be performed in Perl. Several modules implementing basic statistical analyses, such as univariate descriptive statistics, t tests, one-way analysis of variance, simple linear regression, and random number generation are reviewed in Section 14. Modules useful for numerical linear algebra computations are introduced in Section 15. Section 16 provides an overview of PDL, a module for efficient numerical computing in Perl. In this Section we illustrate how modules for linear algebraic computations that can be used for more sophisticated statistical computations. Section 17 illustrates how other statistical and computing applications can be embedded in Perl. Section 18 concludes.

2. Typographical and style conventions used

For the body of text the following typographical convention is used throughout this paper:

- an *italic* face is used for URLs, Newsgroups, journal titles, filenames, functions, for emphasis, and to identify the first instance of a concept requiring definition,
- UPPER CASE letters are used to denote *filehandles* and delimiters in *here documents* constructs,
- a `fixed width` font face is used for Perl syntax, code examples, and user input in interactive examples,
- a *slanted* font face is used to typeset output from interactive examples and the listings of files produced by executing Perl programs.

Perl is a *free-form* language, as opposed to older languages like FORTRAN or COBOL, that are described as *columnar*. In a free-form language all white space, which includes spaces, tabs, commented out text, and newlines, is insignificant except insofar as it separates words and inside quoted strings, as in the string `"median =  23 "`. A judicious use of white space can considerably improve code readability. Code is formatted according to the Perl style guide, available from Perldoc.com Web site located at `http://www.perldoc.com/`. Also, besides font shapes and capitalization, color is used to typeset the code examples in this paper. Color has long been used in statistics to visualize data more effectively. Color can also play an important role in the display of textual information, particularly as color printing devices are becoming the norm and journals are increasingly available in electronic form. A uniformly colored body of text is often hard to read and difficult to debug. Color can make language constructs stand out, thus facilitating their recognition.[4] The following color scheme was used to typeset the code examples

- `dark blue` for language reserved words such as `do` and `while`,
- `dark green` for comments such as `# prints variable name`,

[4] Galton (1880) was the first to observe that a few gifted individuals are able to experience colors for letters and digits. This perceptual phenomenon is known as colour-graphemic synaesthesia. The physicist Richard Feynman was quoted saying: "*When I see equations, I see the letters in colors – I don't know why. As I'm talking, I see vague pictures of Bessel functions from Jahnke and Emde's book, with light-tan j's, slightly violet-bluish n's, and dark brown x's flying around. And I wonder what the hell it must look like to the students.*"

- `dark cyan` for numeric literals such as `3000` and `-1.2e-6`,
- `purple` for string literals such as `"The sample t-statistic is"`, and
- `normal black` text for other special language symbols and user-defined identifiers.

Consider the following Perl script that: 1) downloads the Boston data from STATLIB, 2) extracts and prepares the data, and finally 3) uses R to estimate the coefficients of a regression model and to save the estimates in a file. The right panel presents the same code of the left panel only colored applying the above-mentioned scheme. Note that, for example, the colored right panel clearly shows that a string literal, the purple-colored multi-line string at the bottom of the script, is used to store the R program.

```
# downloads Boston dataset from STATLIB
use LWP::Simple;
getstore(
    "http://lib.stat.cmu.edu/datasets/boston",
    "boston.raw" );

# corrects for record spanning two lines
open( IN,  "boston.raw" );
open( OUT, ">boston.asc" );
do { $line = <IN> } until $. == 22
  or eof;    # Skips the first 22 lines of header
while ( $line = <IN> ) {
    chomp $line;
    $line .= <IN>;    # joins two lines
    print OUT $line;
}
close(OUT);

# sends data to R for regression analysis
open( RPROG,
"| c:/rw1081/bin/Rterm.exe --no-restore --no-save"
);
select(RPROG);
print <<'CODE';
bost<-read.table("boston.asc",header=F)
names(bost)<- c( "CRIM",  "ZN",    "INDUS",
                 "CHAS",  "NOX",   "RM",
                 "AGE",   "DIS",   "RAD",
                 "TAX",   "PTRAT", "B",
                  "LSTAT", "MEDV")
attach(bost)
boston.fit <- lm(log(MEDV) ~ CRIM + ZN + INDUS +
   CHAS + I(NOX^2) + I(RM^2) + AGE + log(DIS) +
   log(RAD) + TAX + PTRAT + B + log(LSTAT))
sum <- summary(boston.fit)$coe[,1:2]
write.table(sum,"boston.out",quote = FALSE)
q()
CODE
close(RPROG);
```

```
# downloads Boston dataset from STATLIB
use LWP::Simple;
getstore(
    "http://lib.stat.cmu.edu/datasets/boston",
    "boston.raw" );

# corrects for record spanning two lines
open( IN,  "boston.raw" );
open( OUT, ">boston.asc" );
do { $line = <IN> } until $. == 22
  or eof;    # Skips the first 22 lines of header
while ( $line = <IN> ) {
    chomp $line;
    $line .= <IN>;    # joins two lines
    print OUT $line;
}
close(OUT);

# sends data to R for regression analysis
open( RPROG,
"| c:/rw1081/bin/Rterm.exe --no-restore --no-save"
);
select(RPROG);
print <<'CODE';
bost<-read.table("boston.asc",header=F)
names(bost)<- c( "CRIM",  "ZN",    "INDUS",
                 "CHAS",  "NOX",   "RM",
                 "AGE",   "DIS",   "RAD",
                 "TAX",   "PTRAT", "B",
                 "LSTAT", "MEDV")
attach(bost)
boston.fit <- lm(log(MEDV) ~ CRIM + ZN + INDUS +
   CHAS + I(NOX^2) + I(RM^2) + AGE + log(DIS) +
   log(RAD) + TAX + PTRAT + B + log(LSTAT))
sum <- summary(boston.fit)$coe[,1:2]
write.table(sum,"boston.out",quote = FALSE)
q()
CODE
close(RPROG);
```

Part I
Data Processing

3. Perl as a scripting language

High-level programming languages can be classified into conventional programming languages and scripting languages. This distinction is not always clear and is not universally accepted, but it is a useful one (see, e.g., Barron 2000).

Conventional (also *system* or *application*) programming languages are typically strongly typed, i.e., have many data types and each variable can be of one type only, make provision for complex data structures, and their programs need to be compiled before they can be executed. By contrast, scripting languages are weakly typed or untyped, make little or no provision for complex data structures, and programs, referred to as scripts, are interpreted, i.e., executed virtually instantaneously. Conventional programming languages are usually employed to develop large applications from scratch meant to work independently from other applications, whereas scripting languages are mostly useful for small, "throw-away" programs, and in connecting and managing other applications.

Scripting languages tend to specialize in performing tasks, such as text manipulation, the creation of GUIs and interactive web pages. They are easier to use but tend to be less efficient, both in terms of memory requirements and speed of execution.

In a study conducted by Prechelt (2000), seven languages, four scripting languages (Perl, Python, Rexx, and Tcl), and three conventional system programming languages (C, C++, Java), were compared with each other in terms of program length, programming effort, runtime efficiency, memory consumption, and reliability. It was found that programs written in conventional languages were typically two to three times longer than scripts, and that productivity, in terms of line of code per hour, was half that of scripting languages. Programs written in a conventional language (excluding Java) consumed about half the memory of a typical script program, and run twice as fast. In terms of efficiency, Java was consistently outperformed by scripting languages. Scripting languages were, on the whole, more reliable than conventional languages. In interpreting these results it is important to bear in mind that they depend critically on the experimental design. Another factor to consider is that computer languages are not immutable entities, but are the subject of continuous improvements, fixes, etc., and, now and then, undergo more drastic revisions.

4. Perl documentation and other resources

A plethora of information on Perl is available from CPAN and from the official Perl's home page at `http://www.perl.com/`, where documentation, tutorials, book reviews, and FAQ can be consulted.

The Perl documentation included with the distribution can be accessed invoking the *perldoc* command, which retrieves the documentation from the Perl POD (for *plain old documentation*) format that comes with the distribution. Table I lists the arguments on which the *perldoc* command can be called and a short description of the documentation output. Documentation

Table I: Arguments and short description of the output of the *perldoc* command

Argument	**Short description of output**
perl:	Perl overview
perltoc:	Perl documentation table of contents (this section)
perldata:	Perl data structures
perlsyn:	Perl syntax
perlop:	Perl operators and precedence
perlre:	Perl regular expressions
perlrun:	Perl execution and options
perlfunc:	Perl built-in functions
perlvar:	Perl predefined variables
perlsub:	Perl subroutines
perlmod:	Perl modules
perlref:	Perl references
perldsc:	Perl data structures intro
perllol:	Perl data structures: lists of lists
perlobj:	Perl objects
perltie:	Perl objects hidden behind simple variables
perlbot:	Perl OO tricks and examples
perldebug:	Perl debugging
perldiag:	Perl diagnostic messages
perlform:	Perl formats
perlipc:	Perl inter-process communication
perlsec:	Perl security
perltrap:	Perl traps for the unwary
perlstyle:	Perl style guide
perlxs:	Perl XS application programming interface
perlxstut:	Perl XS tutorial
perlguts:	Perl internal functions for those doing extensions
perlcall:	Perl calling conventions from C
perlembed:	Perl how to embed perl in your C or C++ app
perlpod:	Perl plain old documentation

on installed extension modules and Perl functions is also available through *perldoc*. Note that to obtain help on a Perl function, the `-f` flag is required. For instance, to get help on the string manipulation function *join*, we can execute

```
%perldoc -f join
    join EXPR,LIST
            Joins the separate strings of LIST into a single string with
            fields separated by the value of EXPR, and returns that new
            string. Example:

                $rec = join(':', $login,$passwd,$uid,$gid,$gcos,$home,$shell);

            Beware that unlike "split", "join" doesn't take a pattern as its
            first argument. Compare the split entry elsewhere in this
            document.
```

There are several active Internet Newsgroups, such as *comp.lang.perl*,[5] and an electronic magazine, *The Perl Journal*, where information and support can be obtained. A complete and updated repository of online Perl and CPAN module documentation is available at the Perldoc.com Web site located at http://www.perldoc.com/. There are also dozens of Perl books on the market. For general purposes, the most complete reference on Perl is the book by Wall, Christiansen, and Orwant (2000). A simpler introduction to Perl is provided by Schwartz and Phoenix (2001) or Sebesta (1999). A very useful reference guide is Christiansen and Torkington (2003), which is a collection of Perl codes useful in solving a wide range of problems. Basic algorithms implemented in Perl useful for statistics are discussed in Orwant, Hietaniemi, and Macdonald (1999). Of particular interest to statisticians are the chapters on matrices, probability, statistics, and numerical analysis.[6] More specific references is provided in subsequent sections.

5. Introduction to the Perl language

This section provides a succinct introduction to Perl's basic language features. For space reasons, several important topics are either left out completely or just briefly mentioned in applied examples. Since, many of Perl's language constructs such as logical operators, control structures, and built-in functions, work exactly as their C and C++ homonymous constructs, more emphasis is given to language features that distinguish Perl from more conventional programming languages used by statisticians. For a more detailed introduction to the Perl language, the sources suggested in the previous section should be consulted.

5.1. Basic data types

Perl has three built-in basic data types: *scalars*, *arrays* of scalars, and *associative arrays* of scalars, known as *hashes*.

Scalars In a Perl program, numeric and textual information is stored in scalars. In Perl, scalars can be numbers, strings, or references (references will be discussed in Section 5.6).

[5] For a complete list of Newsgroups and Mailing Lists consult http://www.cetus-links.org/oo_perl.html.

[6] The Perl code of the examples in the book can be downloaded from the Web page http://examples.oreilly.com/maperl/.

Scalar literals are values determined "literally" by their denotation. A *scalar variable* is denoted by a name and is associated with an address of a memory location where its value is stored. The value of a scalar variable can be used by prefixing its name by the symbol `$`.

Example of acceptable Perl numeric literals include, `-345`, `.0543`, `10e-9` (scientific notation), and `0x24E7C` (hexadecimal). Standard computer arithmetic operators and mathematical functions are predefined for numeric scalars. As an example consider the following numeric scalar assignment operations:[7]

```
$beta_hat = 1.829151e+003;          # scientific notation
$var      = 273884.7556;
$std_err  = sqrt($var);             # computes standard error
$t_stat   = $beta_hat / $std_err;   # assigns 3.49514942484809 to $t_stat
```

Example of *string literals* include single quoted strings, such as, `'ROM'`, and double quoted strings, such as, `"slope coefficient"`. Double quoted strings accept *escape sequences* of special characters (such as tab, `\t`, or newline, `\n`), and *interpolate* "embedded" variable strings, i.e., replace them with their stored value. The difference is illustrated in the following string scalar assignment operations:

```
$str1 = 'Beta is:\t $beta_hat';    # assigns "Beta is:\t $beta_hat" to $str1
$str2 = "Beta is:\t $beta_hat";    # assigns "Beta is:   1829.151" to $str2
```

Several useful string operators, such as `.` (concatenation) and `x` (replication), in addition to string functions, such as *length*, are predefined for string scalars.

Arrays In Perl, *arrays* are ordered lists of scalars indexed by an integer number, starting with zero. A negative index specifies a position starting from the end of the array. Arrays in Perl can have only one index. This means that to represent, say, a matrix, a more complex data structure has to be used (see the discussion in the example on page 22, after the introduction of references). Arrays (and "slices" of arrays) are prefixed by the symbol `@`. For example, `@oecd`, is an example of array. An array can be initialized to the values of a *list*, a set of comma separated values delimited by a pair of matching parentheses. As an example consider the creation of an array of OECD member countries,

```
@oecd = (
    "Australia",      "Austria",     "Belgium",     "Canada",
    "Czech Republic", "Denmark",     "Finland",     "France",
    "Germany",        "Greece",      "Hungary",     "Iceland",
    "Ireland",        "Italy",       "Japan",       "South Korea",
    "Luxembourg",     "Mexico",      "Netherlands", "New Zealand",
    "Norway",         "Poland",      "Portugal",    "Slovakia",
    "Spain",          "Sweden",      "Switzerland", "Turkey",
    "United Kingdom", "United States"
);
```

Individual elements of an array can be accessed using the name of the array prefixed by a `$` symbol and followed by an integer, representing the index, enclosed in a pair of matching square brackets. So, for example,

[7] Note that in Perl a statement must terminate with a semicolon and that lines can be commented out using the symbol `#`.

```
$fourth = $oecd[3];
```

assigns `Canada`, the fourth element of the array `@oecd`, to the scalar `$fourth` and

```
$last   = $oecd[-1];
```

assigns `United States`, the last element of the array, to the `$last` scalar. Note that scalar values are always prefixed by the `$` symbol, even when referring to a scalar that is part of an array or a hash (see the next data type).

A list can also be used to withdraw elements from an array. For example, the statement

```
( $first, $second, $third, $fourth ) = @oecd;
```

assigns `Australia` to `$first`, `Austria` to `$second`, and so forth.

Special function such as *pop*, *push*, *shift*, and *unshift*, can be used to manipulate boundary elements of an array. For example, the *shift* function removes the first element of an array, as illustrated in the following example.

```
$first  = shift @oecd;    # assigns Australia to $first
$second = shift @oecd;    # assigns Austria to $second
```

An example of using the *push* function to build an array of data, will be given in Section 7.1 on page 25.

Hashes An *associative array* can be thought as an array that instead of using a sequence of integers as indices to its values, uses string values ordinarily referred to as *keys*. In Perl associative arrays are known as *hashes*, as its elements are stored and accessed by means of so called *hash functions*. A hash variable is prefixed by the `%` symbol and can be initialized, similarly to arrays, with a list. The initializing list consists of a sequence of alternating values of keys and their associated values. For example, we can create a dictionary that converts country character codes into digit codes with the statement,[8]

```
%convert = ( "AFG", "004", "ALB", "008", "DZA", "012" );
```

or, with the semantically equivalent, but syntactically clearer statement,

```
%convert = ( "AFG" => "004", "ALB" => "008", "DZA" => "012" );
```

As an example, evaluating the expression `$convert{AFG}`, returns the three-digit code 004.[9] Note that, as in the case of arrays, to access an individual elements, the `%` symbol is changed to `$`. Like ordinary arrays, associative arrays do not need to be declared and are automatically initialized at their first usage. Moreover, they do not have any prespecified order to their elements but is it possible to access all the elements in turn using the *keys* function and the *values* function. For instance, the statement,

[8] International Standard ISO 3166-1 codes for the representation of names of countries and their subdivisions, available United Nations Statistics Division Web site.

[9] Note that in Perl a number with a leading 0 digit is a literal for a number expressed by octal notation. Attempting to initialize the hash with the following statement,

```
%convert = ( "AFG" => 004, "ALB" => 008, "DZA" => 012 );
```

results in an error as $(008)_8$ is not a legal number in base 8.

```
@c_codes = keys %convert;
```

stores the three-letter country codes contained in the hash in the array `@c_codes`.

5.2. Basic input/output

In this section we review some basic ways in which Perl can communicate with the "outside world," through peripherals and files.

Interactive i/o The *print* function prints a string or a comma-separated list of strings to the screen (by default). Text enclosed in double quotation marks following the *print* operator is printed literally with the exception of variable names which are replaced with the value of the corresponding variables and standard "backslash escape code" sequences will be interpreted in much the same way as in C or C++. As an example, consider the following lines of Perl code,

```
$t_stat = 13.7;
print "The sample t-statistic is:\t$t_stat\n";
```

where, `\n`, represents the newline character and, `\t`, a tab. Executing the above statements produces the following output:

```
The sample t-statistic is:      13.7
```

To print special characters like a literal backslash, underscore, or a double quote, these characters need to be *escaped*, i.e., prefixed by the symbol \. If a filehandle (see next paragraph) is specified after the *print* operator, the output is printed to the associated file (or process). To get user input inside a Perl program, we can use a statement of the type `$user_input = <STDIN>`, which assigns to the variable `$user_input`, the text typed by the user (terminating with a "return") on the keyboard.

File i/o Files can be accessed within a Perl program through *filehandles*. A filehandle is a name given to a file for use inside a program, through an *open* statement. Typically, filehandle names are written in uppercase letters, to differentiate them from other language elements. For example, the statement

```
open( FILE, "filename" );
```

ties the file *filename* to the filehandle FILE. To read a file the so called *line input* or *angle* operator, `<>`, can be used. Evaluating a filehandle in angle brackets yields the "next line," including the terminating newline character, from the associated file. If the input line inside the conditional expression of a *while* loop is not assigned to a scalar, as with the statement `$line = <FILE>`, then it is automatically assigned to the special variable denoted `$_`. As long as the loop does not reach the end-of-file (EOF) marker, the condition returns always `true`. Often, the trailing newline character of the string is not needed an can be removed with the string function *chomp*. See Section 7.1 on page 24, for an example. Using `<>` without any argument, is equivalent to using `<STDIN>`, a predefined Perl filehandle that returns the lines from a file specified on the command line, `perl namefile`.

Perl's process manipulation capabilities allows to easily interact with other programs. The *open* function can be used to create filehandles not only to access files for reading, but also for various other purposes, such as creating new files, writing to files, and piping. *Pipes* allow to get input from, and send output to files, the operating system, and other applications. Because of this built-in functionality, Perl is also known as a "glue language." The following template statements, can be used to perform various input-output operations

```
open( FILE, ">filename" );     # writes to filename
open( FILE, ">>filename" );    # appends to existing file
open( FILE, "| command" );     # sets up an output filter
open( FILE, "command |" );     # sets up an input filter
```

Once created, the filehandle FILE can be used to access the associated file or process until it is explicitly closed with a *close* statement.

Here document In order to conveniently input multi-line commands into another application, the so-called *here document* notation for a multi-line string can be used. The notation consists of the redirection symbol << and a string, typically in upper case letters, that serves to delimit the lines of input.

The following example illustrates how the language features described in this section can be used to interact with a statistical software application.

Interacting with R example Consider the problem of converting a file, in this example the *anscombe* dataset (see Anscombe 1973), *P025b.dta*,[10] from the stata format to the CSV format (see Section 7.1 on page 24), using R's *foreign* library,

```
open( RPROG, "| c:/rw1081/bin/Rterm.exe --no-restore --no-save" )
  ;    # creates the filehandle RPROG
$R_program = <<'R_CODE';
library(foreign)                  # begining of R program
ans <- read.dta("c:/P025b.dta")
write.table(ans,"anscombe.csv",quote = FALSE,row.names=FALSE, sep = ",")
q()                               # end of R program
R_CODE
print RPROG $R_program;
close(RPROG);
```

Note that the delimiter indicating the beginning of the multi-line string is followed by a semicolon and that no space precedes the delimiter indicating the end. The first five lines of the *anscombe.csv* file contains:

```
y1,x1,y2,x2,y3,x3,y4,x4
8.03999996185303,10,9.14000034332275,10,7.46000003814697,10,6.57999992370605,8
6.94999980926514,8,8.14000034332275,8,6.76999998092651,8,5.76000022888184,8
7.57999992370605,13,8.73999977111816,13,12.7399997711182,13,7.71000003814697,8
8.8100004196167,9,8.77000045776367,9,7.1100001335144,9,8.84000015258789,8
```

[10] Available from the URL: http://www.ats.ucla.edu/stat/stata/examples/chp/chpstata_dl.htm.

Note that in the conversion the non-integer values are slightly changed and appear with extra spurious nonsignificant digits. A default filehandle can be selected with the select function.

The *print* function prints using a default format. To control the output format the *printf* function can be used. The *printf* function follows the rules of the homonymous function available in C or C++. See the next Section for an application of the *printf* function.

5.3. Program control statements

Perl provides a wide collection of program execution control statements. Most will work as the homonymous statements available in C or C++. In this section, we will introduce conditional expressions in Perl, and a few conditional and loop statements, that are specific to Perl.

Control expressions Most control statements have at their heart a control expression whose evaluation will determine the flow of the program. A *control expression* must evaluate to either `true` or `false`. In Perl, the number zero, as well as the empty string `""` and anything that evaluates to a string containing a single zero string, `"0"`, is `false`. The value `undef`, the special value given to a scalar by default if no other value is assigned to it, is also `false`. Every other value is `true`. Typically control expression are constructed utilizing relational operators such as `==` (equal), `!=` (not equal), `>` (greater than), and `>=` (greater than or equal), and Boolean operators like, `&&` (AND) and `||` (OR). For example, using relational and Boolean operators we can construct the compound expression

```
( $rgdpch < 0 || $rgdpch > 100000 )
```

which is `true`, if the value stored in the variable `$rgdpch` is strictly less then zero or strictly greater than 100,000.

Conditional statements Conditional execution uses either the `if` statement or the `unless` statement. An `else` statement following `if`, is optional. The `if` and `else` statements are used as conditionals in C, C++, or Java. The `unless` statement is specific to Perl. It allows the execution of a block of code only if the conditional expression is false. As an example, consider the two-tail test decision implemented in Perl code:

```
$t_stat         = 1.7;
$critical_value = 2.228;
unless ( $t_stat > abs($critical_value) ) {
    print "The null can not be rejected\n";
}
```

The `elsif` is also specific to Perl. `elsif` simply combines an `else` and an `if` statements together in one. As an example consider the following block of code.

```
$p_value = .5;
print "Strength of evidence coming from the data against the null:\n";
if ( $p_value < .02 ) {
    print "The data provides strong evidence against the null\n";
}
elsif ( $p_value < .07 ) {
```

```
    print "The data provides some evidence against the null\n";
}
elsif ( $p_value < .20 ) {
    print "The data provides little evidence against the null\n";
}
else {
    print "The data provides no evidence against the null\n";
}
```

Note that, as with most programming languages, each condition is tested in the order in which it appears in the script. If several conditions in the statement are simultaneously "true", Perl will execute the action associated with the first true condition encountered and ignore the rest.

Perl does not have a built-in C-type `switch` statement. A switch-type multiple selection construct can be simulated using the `last` operator (see Sebesta 1999, page 74). Another option is to use the `Switch` module available from CPAN (see Section 8 on extending Perl with modules).

Loops Perl provides the standard `while` and `for` loops. The `while` loop is typically used to process all lines in a file. The `for` loop makes use of a *loop variable*, to allow the code inside the loop block to be iterated, as the loop variable assumes the values in a sequence of integers. These loop constructs work as their C/C++ equivalents. Perl provides a powerful extension of the `for` loop, that is designed for iteration over lists and arrays: the `foreach` loop control statement. `foreach` allows the extraction of elements out of an array or a list, one at a time, without employing a loop variable.

Array printing example Often, we need to print the content of an array. The statement

```
print @oecd;
```

produces[11]

```
AustraliaAustriaBelgiumCanadaCzech RepublicDenmarkFinlandFranceGermanyGreece...
```

which is, obviously, difficult to interpret and use. The `foreach` loop can be used to extract and print all the elements of the `@oecd` array. For example, executing the code

```
foreach $member_country (@oecd) {
    print "$member_country \n";
}
```

prints

```
Australia
Austria
Belgium
Canada
Czech Republic
```

[11] The output is truncated for space reasons.

```
Denmark
Finland
France
Germany
Greece
...
```

The following example combines input-output language elements and iteration to format a table of values.

Table formatting example Often regression output from a statistical program needs to be processed in order to be included inside a document. Processing output from a statistical package can become a repetitive, tedious, and error prone task. Consider the case of creating a LaTeX table from the following typical regression output in human readable format, that has been saved in the file *longley.reg*[12]

```
X1       -52.99357014      129.54487        -.409    .6911
X2        .7107319907E-01  .30166400E-01    2.356    .0402
X3       -.4234658557      .41773654       -1.014    .3346
X4       -.5725686684      .27899087       -2.052    .0673
X5       -.4142035888      .32128496       -1.289    .2263
X6        48.41786562      17.689487        2.737    .0209
```

where the columns contain the variable names, the estimated coefficients, the standard errors, the t-statistics, and the p-values respectively. In order to include these results in a LaTeX document as a table, we need to ensure that

- within the body of the table, the ampersand character, `&`, is used to separate columns of text within each row, and the double backslash, `\\`, is used to separate the rows of the table;
- figures are rounded to the desired number of decimal places, typically lower than the number of digits provided by statistical and computing software to avoid "specious accuracy."

One way to print a number with the desired format is to pass it through Perl's *printf* function.[13] The arguments of the *printf* function are a format string and a list of values. The format string consists of a string incorporating "place-holders," beginning with a percentage sign. The formatting convention used is inherited from the C function *printf*. The following script, reads the *longley.reg* and outputs a LaTeX-formatted table

```
open( TABLE, "longley.reg" );
$prec  = 3;    # sets number of decimals
$width = 8;    # sets the width of the field
while (<TABLE>) {
    chomp;
    @line = split;
```

[12] See Section 16.8 on page 62 in Part II.

[13] A more flexible approach is to use the `Convert::SciEng` module available from CPAN. Modules will be introduced in Section 8.

```
        printf "%2s", $line[0];    # prints variable name
        for ( $i = 1 ; $i <= $#line ; $i++ ) {
            printf "& %${width}.${prec}f", $line[$i];  # prints all other fields
        }
        print "\\\\ \n";    # prints latex end-of-line carachter
    }
    close(TABLE);
```

The output of the script is:

```
X1&  -52.994&  129.545&   -0.409&    0.691\\
X2&    0.071&    0.030&    2.356&    0.040\\
X3&   -0.423&    0.418&   -1.014&    0.335\\
X4&   -0.573&    0.279&   -2.052&    0.067\\
X5&   -0.414&    0.321&   -1.289&    0.226\\
X6&   48.418&   17.689&    2.737&    0.021\\
```

5.4. User defined functions

Subroutines can be defined, anywhere in a Perl script, by typing the keyword *sub*, followed by the subroutine name and a block with the body of code enclosed in matching curly brackets. For instance, to define a function that calculates the sample mean, we can use the following code,

```
sub mean {
    @data = @_;
    $sum  = 0;
    foreach $elem (@data) { $sum += $elem; }
    return $sum / @data;
}
```

where the special array `@_` makes the vector argument passed to the function available within the function's block of code. The argument of the function *return* specifies the value returned by the subroutine. The *mean* function can then be invoked by its name and with an array as argument,

```
@vect = ( 2.5, 3, 4.3, 5, 6.3 );    # defines a numerical array
print mean(@vect);                  # prints: 4.22
```

We will see in Section 5.6, on page 21, how routines can be made more efficient by allowing arrays to be passed *by reference*.

5.5. Regular expressions

One of Perl's most distinguishing language feature is its extensive built-in support for string-pattern matching. A string that uses a special notation to describe a pattern to be matched against another string is known as a *regular expression*. A large number of software tools incorporate regular expressions as a key part of their functionality. Unix-oriented command line tools like grep, sed, and awk are specialized in regular expression processing. Unix-oriented text editors, such as vi and emacs, allow search and/or replacement based on regular expressions. Even operating systems like Windows support a restricted form of regular expressions

as "wild-card characters," `*`, and, `?`, useful for selecting sets of file names. Besides Perl, other scripting languages such as Tcl and Python, have extended built-in support for regular expressions. Among the special-purpose languages used by statisticians, S-PLUS and R, support standard (POSIX) regular expressions in a few of their functions. R supports Perl-compatible regular expressions as well. Some applications of regular expressions include checking the validity of input data, automatic editing of source code, and extracting data from program output. In Perl there are two important string-pattern matching operators that make use of regular expressions: the *matching* operator, `m//`, and the *substitution* operator, `s///`. Usually a string matches itself so, for example, consider a string representing the result of tossing fifty times a coin,

```
$seq = "HHTTHTHHTTTHHTTTTHHHHHHHHHTTHHTTHHTHTHTHHHHTHHTTTHT";
```

By default, Perl matches the leftmost possible pattern. The following statement

```
if ( $seq =~ m/HTTH/ ) { print "Pattern HTTH found!" }
```

matches the first occurrence of the pattern HTTH at trial number 2, and returns

```
Pattern HTTH found!
```

Note that,

- `=~`, is a *string binding* operator, which indicates that the pattern on the right-hand side, will be matched against the string stored in the variable `$seq` on the left-hand side, and
- the expression `$seq =~ m/HTTH/` evaluates to true if a match occurs.

If no string is specified via the `=~` operator, the special, `$_`, string is searched.

Beside characters that match themselves, there are also special characters known as *metacharacters*, that match other characters. For instance, `.` (matches any character), `?` (matches zero or one of the previous character), `*` (matches zero or more of the previous character), `+` (matches one or more of the previous character). These metacharacters are collectively known as *quantifiers*. Other quantifiers are, for instance, `{n}` (match exactly n times), `{n,m}` (match a minimum of n times and a maximum of m times), and `{n,}` (match a minimum of n times or more). For instance the following expression

```
$seq =~ m/H{3}/;
```

matches the occurrence of the first H triplet. Perl also provides the positions of what was last matched through the special `@-` and `@+` arrays. In particular, the first element of `@-`, `$-[0]`, stores the position of the start of the entire match and the first element of `@+`, `$+[0]`, stores the position of the end. For instance,

```
printf "First head run occurs at trial: %d\n", $-[0] + 1;
```

returns

```
First head run occurs at trial: 18
```

One or more single letter can be placed after the final delimiter to change the behaviour of the *matching* or *substitution* operator. For instance, `/i`, makes the match insensitive to case, `/g`, finds all possible matches within a string, etc. Continuing our sequence example, using the the `/g` matching modifier, it is possible to match all the head runs of length three in the sequence.[14]

```
$runs = 0;    # initializes runs count variable
print "Runs occur at trials number: ";
while ( $seq =~ m/H{3}/g ) {
    $runs++;    # increment runs count
    $trial = $-[0] + 1;    # stores start of run
    print "$trial, ";
}
printf "\nNumber of runs: %d\n", $runs;
```

returns

```
Runs occur at trials number: 18, 21, 39,
Number of runs: 3
```

The results show that the sequence contains three runs of length three, occurring at trials 18, 21, and 39. The metacharacter `|` allows to choose among alternatives. For instance,

```
$seq =~ m/(HTTTH|THHHT)/;
```

matches the first occurrence of either `HTTTH` or `THHHT`. Parentheses, used to group parts of a pattern, capture what they group to a special variable. Each set of parentheses has a number associated with it. The first pair of parentheses assigns its substring to the special variable `$1`, the second to `$2`, and so on. Therefore, the code

```
print "Sequence matched: $1\n";
printf "Starting at trial: %d\n", $-[0] + 1;
```

prints

```
Sequence matched: HTTTH
Starting at trial: 8
```

A string of characters enclosed in matching square brackets is called a *character class* and matches any one of the characters in the string. For instance, the regular expression `[123456789]` matches any non-zero digit. The notation can be shortened using a hyphen, `-`, to indicate a range, as in `[1-9]`. If the first character in the list is the caret, `^`, then the regular expression matches any character not in the list. Many character classes have a special metasymbol to denote them. For instance, *metasymbols* such as `\s` (whitespace),[15] `\S` (non-whitespace), `\d` (digit), `\D` (any non digit), `\w` (word character), `\W` (non-word character), etc. *Anchors* are special metasymbols that match positions rather then characters. For instance, `^` (matches at the beginning of a string), For instance, `$` (matches at the end of a string), `\b` (matches at any position that is a word boundary), `\B` (matches at any position that is not a word boundary).

[14] The counting of consecutive head runs is restarted when the desired value 3 is reached according to the definition given by Feller (1968, page 305).

[15] Defined as the character class `[ \t\n\r\f]`.

Benford's law example Let us verify if the GDP data, that we will introduce in Section 7.1 on page 24 together with how to read data in Perl, follows Benford's first digit law (see, e.g, Hill 1996).

```
open( IN, "f:/web/statmeth/pwt6_year.csv" );
do { <IN> } until $. == 4 or eof; # Skips the top 4 lines header of the file
while (<IN>) {
    $rgdpch = ( split(/,/) )[12];    #  extracts real GDP from 13th column
    $nonzero_digit = '[1-9]';        #  defines a "nonzero digit" pattern
    if ( $rgdpch =~ /($nonzero_digit)/ ) {    #  matches and captures digit
        $count[$1]++;    #  counts and stores the occurrences of each digit
        $total++;        #  counts and stores the number of observations
    }
}
```

To print the results with the expected frequency, given by logarithmic distribution

$$f(x) = \log_{10}\left(1 + \frac{1}{x}\right),$$

we can use the following code:[16]

```
print "Digit Count Obs. freq.  Exp. freq.\n";
foreach $digit ( 1 .. 9 ) {
    printf(
    "%5d  %5d  %10.4f  %10.4f\n",
    $digit, $count[$digit],
    $count[$digit] / $total,           #  computes observed frequencies
    log( 1 + 1 / $digit ) / log(10)    #  computes the expected frequencies
    );
}
```

When executed the code produces the following output.

```
Digit Count Observed Expected
1     1841    0.3348   0.3010
2      956    0.1738   0.1761
3      591    0.1075   0.1249
4      435    0.0791   0.0969
5      399    0.0726   0.0792
6      346    0.0629   0.0669
7      312    0.0567   0.0580
8      314    0.0571   0.0512
9      305    0.0555   0.0458
```

Typesetting example A typical TEX typesetting problem affecting documents that make use of many acronyms (like this paper) is to put the correct amount of space after a sentence-ending period (see Knuth 1984, page 74). Usually TEX puts the correct amount of space, larger than intra-sentence spaces, after a period. There are a few special cases in which TEX

[16] Note that since Perl's *log* function returns the natural logarithm of its input, in this example, to convert to logarithms with base 10, we used the base conversion formula, $\log_{10}(x) = \frac{\log_e(x)}{\log_e(10)}$.

makes an incorrect decision. For example, TEX assumes that a period following an upper case letter represents someone's initial and does not recognize it as a period ending a sentence. This assumption can lead to typesetting errors. Consider the case where a sentence ends with an acronym. In such cases, TEX will put an insufficient amount of space after the period. One way to correct this problem is to type `\@.` after an acronym that ends a sentence. The following script looks for at least two consecutive uppercase letters followed by a period and by one or more spaces. The regular expression `([A-Z]{2,}\.\s+)` assigns the desired match to the variable `$1`.

```
while ( $line = <> ) {
    if ( $line =~ /([A-Z]{2,})\.\s+/ ) {
        print "$1 - at line number: $.\n";
    }
}
```

The special variable `$.` contains the line number where the match occurred. Another use of this variable will be shown in Section 7.1 on page 25. When applied to the TEX file of this paper (before adding this section!), it generated the following output.

```
WWW - at line number: 283
FILE - at line number: 1041
CPAN - at line number: 1215
CPAN - at line number: 1450
II - at line number: 2014
ID - at line number: 2157
ID - at line number: 2791
HTTP - at line number: 3461
LWP - at line number: 3545
LRE - at line number: 4536
LRE - at line number: 4587
RNG - at line number: 5226
RANDLIB - at line number: 5485
IDL - at line number: 5776
PGPLOT - at line number: 5827
WARRANTY - at line number: 5868
NIST - at line number: 6269
PDL - at line number: 6297
```

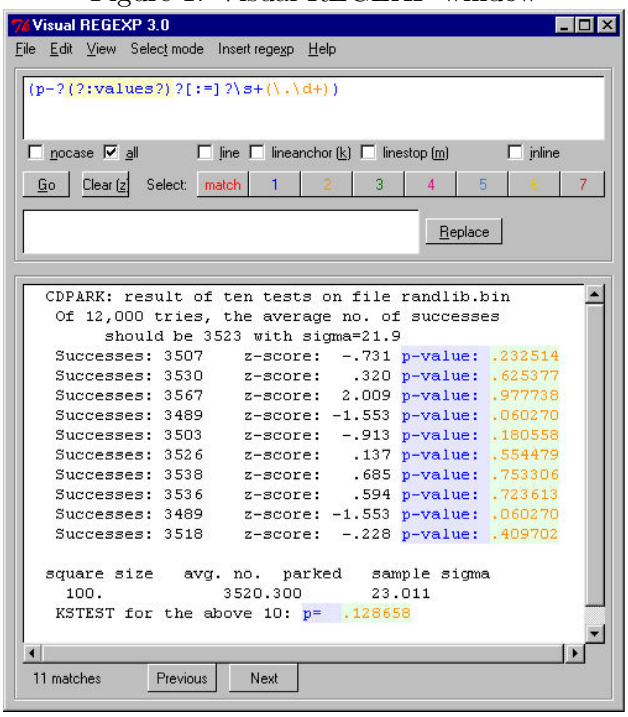
Split* and *Grep Regular expressions are also used by Perl's built-in operators *split* and *grep*. The *split* operator accepts a pattern and a string as arguments and returns list of substrings, by finding delimiters that match the pattern. The *split* function will be discussed in more details in Section 7.1. The *grep* operator which accepts a pattern and an array as arguments and returns a list that contains all the elements of the argument array that match (or do not match, if the negation operator, `!`, is used). Consider printing the mean of the array `@lgdp`,[17] using the previously defined function *mean*, excluding missing values, represented by the `NA` string,

```
print mean( grep !/NA/, @lgdp );
```

[17] This array will be defined in Section 7.2 on page 28.

For more information on regular expressions consult the excellent `perlre` manpage, Wall *et al.* (2000) and Friedl (2002). Very useful tools to rapidly gain confidence and insight in how to use effectively regular expression are tools that visualize regular expression matches. One of such tools is Visual REGEXP.[18] To visualize a regular expression, we can type it in the expression text area at the top of the window. After pressing the 'Go' button, the matches in the sample text at the bottom of the window, are highlighted. Figure 1 illustrates the result of such an action.

Figure 1: Visual REGEXP window



Different colors are used to show different matching groups. For more detail consult the accompanying documentation.

5.6. References

A reference is a scalar whose value is the memory location of another variable, which can be of any of the basic types. References are essential in making efficient use of computer resources. For instance, passing an array to a subroutine will cause a copy of the array to be made, which can result in excessive pressure on memory and processing units resources. By passing a reference to an array to a subroutine, no new copy of the array will be made. References are also needed to create complex data structures from simpler ones. For example, as we mentioned earlier, arrays can have only one index, and are therefore not suitable, as such,

[18] The version used here is the stand-alone Win32 Visual REGEXP 3.0, obtained from `http://laurent.riesterer.free.fr/regexp/`. For other platforms a script requiring Tcl/Tk 8.3.0 (or later) is available.

to represent matrices. We will see how references allow arrays to contain matrices. In Perl there are two fundamental reference operations, assignment and dereferencing. The *reference assignment* operation, sets a reference variable to a useful memory address. References are created by prefixing a variable with the symbol `\`. For example, given a scalar, say `$sca=47`, `\$sca` is a reference to `$sca` which can be assigned to a scalar variable with the following statement

```
$ref_to_sca = \$sca;
```

now `$ref_to_sca` is a reference to the scalar 47. Note that, in this example,

```
print $ref_to_sca;
```

prints the hexadecimal representation of the array's memory address, `SCALAR(0x1bd28a4)`.

The *dereferencing* operation allows access to the value of the variable referred to. Dereferencing is done by prefixing the reference with the data type symbol associated with the value referenced. For example, since `$ref_to_sca` is a reference to a scalar variable, to access its value, we need to prefix the reference by the symbol `$`. The statement

```
print $$ref_to_sca;    # prints: 47
```

References to arrays and hashes can also be created. Consider initializing an array and assigning a reference to it to a scalar,

```
@data       = ( 5, 10, 15, 20 );
$ref_to_dat = \@data;
```

The arrow operator `->`, when used between a reference to an array and an array index, provides access to array elements. In our example, `print $ref_to_dat->[2]` prints the third element of the array. To retrieve the array value `$ref_to_dat` refers to we can use the expression: `@$ref_to_dat`.

Passing by reference example Let's reconsider the *mean* subroutine defined in Section 5.4 on page 15. The function, needs just a few small modifications to handle references to arrays instead of arrays.

```
sub mean {
    ($dataref) = @_;
    $sum = 0;
    foreach $elem (@$dataref) { $sum += $elem; }
    return $sum / @$dataref;
}
```

Now it can be called passing a reference to an array as argument as in the statement

```
mean($ref_to_dat)
```

An example of how references can be used to pass functions to subroutines will be given in section 16.9 in part II.

Matrix data structure example Consider defining a data structure for matrices. An $n \times m$ matrix can be implemented as an anonymous list of n, length m, anonymous lists. For example, a 2×3 matrix can be defined with

```
$mat = [ [ 1, 2, 3 ], [ 3, 4, 5 ] ];
```

The statement `$mat->[1][0]` accesses the matrix element at the intersection of the second row and first column.

A way to conveniently print complex data structures will be discussed in Section 8 on page 31.

6. Running Perl programs

There are several ways in which Perl programs may be run. Depending on the operating system, Perl scripts can be run by clicking a file icon, calling an executable file, or by invoking the `perl` interpreter at the operating system command prompt. The latter mode is more likely to work on most systems. For instance, to run a list of valid Perl statements included in a script named *pwt.pl*, we can type the command

```
> perl pwt.pl
```

It is possible to invoke the `perl` interpreter with a one-line program using the flag `-e`. As an example, consider a simple calculation

```
> perl -e 'print -52.99/129.54'
-0.409062837733519
```

Note that some environments require double, instead of single quotes, to delimit the Perl statement after the flag. To process more than one line, the `-n` flag may be used. Flags can be combined as the following example illustrates. Consider the following regression output from a statistical package, saved in a file named *longley.out*

```
+----------------------------------------------------------------------+
| Ordinary    least squares regression    Weighting variable = none     |
| Dep. var. = TOTEMP   Mean=   65317.00000    , S.D.=   3511.968356     |
| Model size: Observations =      16, Parameters =   6, Deg.Fr.=     10 |
| Residuals:  Sum of squares= 2257822.600    , Std.Dev.=      475.16551 |
| Fit:        R-squared=  .987796, Adjusted R-squared =          .98169 |
| Model test: F[  5,     10] =  161.88,    Prob value =          .00000 |
| Diagnostic: Log-L =   -117.5616, Restricted(b=0) Log-L =    -152.8096 |
|             LogAmemiyaPrCrt.=   12.646, Akaike Info. Crt.=     15.445 |
| Autocorrel: Durbin-Watson Statistic =   1.27749,   Rho =       .36125 |
+----------------------------------------------------------------------+
+---------+--------------+----------------+--------+---------+----------+
|Variable | Coefficient  | Standard Error |t-ratio |P[|T|>t] | Mean of X|
+---------+--------------+----------------+--------+---------+----------+
 X1       -52.99357014      129.54487        -.409   .6911  101.68125
 X2        .7107319907E-01  .30166400E-01    2.356   .0402  387698.44
 X3       -.4234658557      .41773654       -1.014   .3346  3193.3125
 X4       -.5725686684      .27899087       -2.052   .0673  2606.6875
 X5       -.4142035888      .32128496       -1.289   .2263  117424.00
 X6        48.41786562      17.689487        2.737   .0209  1954.5000
```

To extract only those records that have 6 fields, we can type

```
> perl -ne 'print if split==6' longley.out
 X1        -52.99357014       129.54487         -.409    .6911   101.68125
 X2         .7107319907E-01   .30166400E-01     2.356    .0402   387698.44
 X3        -.4234658557       .41773654        -1.014    .3346   3193.3125
 X4        -.5725686684       .27899087        -2.052    .0673   2606.6875
 X5        -.4142035888       .32128496        -1.289    .2263   117424.00
 X6         48.41786562       17.689487         2.737    .0209   1954.5000
```

To print only the results for which the t-statistic, the fourth field in the record, is greater than two in absolute value[19]

```
> perl -ne 'print if abs((split)[3])>2' longley.out
 X2         .7107319907E-01   .30166400E-01     2.356    .0402   387698.44
 X4        -.5725686684       .27899087        -2.052    .0673   2606.6875
 X6         48.41786562       17.689487         2.737    .0209   1954.5000
```

Though a simple text editor that can create ASCII text files, and a command-line to execute Perl programs are enough to write, run, and maintain small Perl programs, several tools that can facilitate programming are available. More complex programs are typically developed using an Integrated Development Environment (IDE). An IDE provides a GUI-oriented interface for managing the steps required to craft and run a computer program (edit, color syntax, execution, profiling, debugging, etc.).

The Emacs multi-platform programmable editor,[20] achieves the functionality of an IDE by providing a Perl *programming mode* that determines how Perl code is displayed, checks the syntax, automates indentation, defines shorthand keystrokes, menus, etc. This mode is "entered" by default when a file terminating with a recognized Perl extension (such as *.pm* or *.pl*) is accessed or when a file with that extension is saved. There is an advanced Perl mode, the *cperl* mode,[21] that can be invoked with the keystroke sequence `ALT x cperl-mode`.[22] A Perl program can be executed by selecting, using the mouse, the `Tools` title from the menu bar, and then the `Compile` item from the `Tools`'s drop-down menu.

Another, easier to use, tool that provides IDE functionality is **SciTE**.[23] **SciTE** is free and can be used with dozens of useful computer languages, including LaTeX, C/C++, FORTRAN, and MATLAB. Within the IDE provided by **SciTE**, a Perl program can be executed by selecting, using the mouse, the `Tools` title from the menu bar, and then `Go` from the `Tools`'s drop-down menu. Other Perl debugging tools can be analogously accessed. Figure 2 displays an example of **SciTE** window.

[19] Obviously, in general, this statement may print other lines in which there happens to be a number that satisfies the condition.

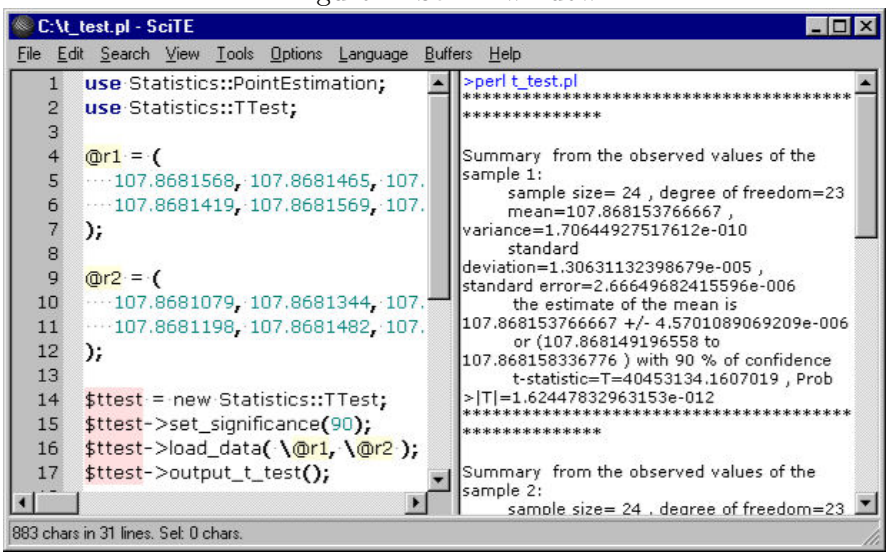
[20] Emacs can be obtained through http://www.gnu.org/software/emacs/emacs.html.

[21] If not already included in the Emacs distribution, the Lisp file that implements the mode can be downloaded from http://www.cpan.org/modules/by-module/Softref/ILYAZ/cperl-mode/.

[22] It is more convenient to change the default Perl mode in Emacs by adding a line such as `(add-to-list 'auto-mode-alist '("\\.\\([pP][Llm]\\|al\\)\\'" . cperl-mode))` to the *.emacs* configuration file.

[23] The version used here is the stand-alone Win32 Release version 1.58, available from http://www.scintilla.org. **SciTE** is also available for Linux and Unix systems.

Figure 2: **SciTE** window



7. Data processing with Perl

Preparing a dataset for analysis by a statistical program typically involves several steps. Some of these steps might be considerably labor intensive and might involve direct "manual" intervention. In this section we will illustrate how Perl can be used to considerably simplify and automate many data processing tasks. By minimizing the amount on "manual" editing of large data files, using Perl can considerably reduce the chance of errors such as transcription and transposition errors.

In the next sections we will illustrate how a longitudinal dataset can be processed using Perl's capability of manipulating text by pattern matching through hashing and regular expressions.

7.1. Reading data

Even though Perl can easily deal with data in binary format, all the subsequent examples presuppose the use of textual (ASCII) data. Textual data files can be readily transferred from one system to another,[24] can be read by any program, and are mostly self-documenting. By contrast, binary files, though more compact in size, need specialized (and often proprietary) software to be processed, cannot be easily ported between platforms, are not human readable, and require considerable accompanying documentation (often not available).

Suppose we want to read and process the recently released version 6.1 of the well known Penn World Table[25] (PWT), which provides national income accounts converted to international

[24] There is a source of constant irritation when transferring text files between MSDOS/Win32 and Unix platforms. MSDOS/Win32 use a carriage return (`CR`, ASCII 13) followed by a line feed (`LF`, ASCII 10) to end a line, whereas Unix flavors use only an `LF` as line terminator (Macintosh uses a `CR`). The result of this different standards is that MSDOS and Win32 regard a file created on a Unix system as being one long line, often too long to be read by regular word processors, whereas an MSDOS/Win32 text file, when viewed on a Unix system, will display the symbol `^M` at the end of each line. That is also why text files have a on MSDOS/Win32 and Unix platforms (1 byte per line, except for the last, shorter in Unix).

[25] Alan Heston, Robert Summers and Bettina Aten, Penn World Table Version 6.1, Center for International Comparisons at the University of Pennsylvania (CICUP), October 2002. The dataset can be found at the Web

prices for 168 countries for the period 1950–2000. The file is available in the CSV (for *comma-separated variable*) format, an extremely popular computer file format, that may come from spreadsheets, statistical software, databases, and various other sources. Here we show the first 6 (out of 10,209) records of the file named *pwt61.csv* (note that the first record has been split into two lines for space reasons):[26]

```
isocode,XRAT,POP,cc,cg,ci,kc,kg,ki,openc,openk,csave,
   rgdpch,rgdpl,cgdp,rgdptt,y,p,pc,pg,pi,yr,cgnp,rgdpeqa,rgdpwok
ABW,,,,,,,,,,,,,,,,,,,,,1960,,,
ABW,,,,,,,,,,,,,,,,,,,,,1961,,,
ABW,,,,,,,,,,,,,,,,,,,,,1962,,,
ABW,,,,,,,,,,,,,,,,,,,,,1963,,,
ABW,,,,,,,,,,,,,,,,,,,,,1964,,,
```

The first line contain the file header. The actual data is contained in the lines that follow the header. The simplest way to process a comma-separated line is by using the Perl *split* operator, which takes the separator, like a comma `/,/` or a space, `/s/`, and the string to process (the special variable `$_` if this is omitted), as arguments. A more robust method to process CSV files will be discussed in Section 8 on page 31. Say we are also interested in extracting three variables (to be used in later examples), the country code (`isocode`), the year (`yr`), and the real GDP *per capita* (`rgdpch`), from the PWT data file. The variables can be read, stored into arrays using the *push* operator, and then printed, using the following script:

```
open( IN, "pwt61.csv" );
do { $line = <IN> } until $. == 1 or eof;  # Skips the first line
while ( $line = <IN> ) {
    chomp $line;
    (
        $isocode, $XRAT,   $POP,     $cc,    $cg,    $ci,     $kc,
        $kg,      $ki,     $openc,   $openk, $csave, $rgdpch, $rgdpl,
        $cgdp,    $rgdptt, $y,       $p,     $pc,    $pg,     $pi,
        $yr,      $cgnp,   $rgdpeqa, $rgdpwok
      )
      = split ( /,/, $line );
    push ( @code,   $isocode );
    push ( @year,   $yr );
    push ( @rgdpch, $rgdpch );
}
for ( $rec = 0 ; $rec < 10208 ; $rec++ ) {
    print "$code[$rec], $year[$rec], $rgdpch[$rec]\n";
}
close(IN);
```

Say we named the script *readPWT.pl*, to run it and save (redirect) the output in file named *output.file*, one use the command `perl readPWT.pl > output.file`, at the operating system command prompt. The first ten lines of *output.file* contain:

---

address: http://pwt.econ.upenn.edu/.

[26] Perl can easily process more complex data formats, with records that span over many lines.

```
ABW, 1960,
ABW, 1961,
ABW, 1962,
ABW, 1963,
ABW, 1964,
ABW, 1965,
ABW, 1966,
ABW, 1967,
ABW, 1968,
ABW, 1969,
```

7.2. Examples of recoding, validating, and transforming variables

Preparing a dataset most often consists of merging, i.e., combining, several data files from different sources into one, and of putting the data into the correct format required by a specific statistical software for analysis. The preparation of datasets might require several, often labor intensive, steps. Perl can be used to considerably simplify and automate these processes. Also, by minimizing the amount of "manual" editing of large data files, using Perl can considerably reduce the chance of errors.

Consider the longitudinal dataset example of Section 7.1. Several issues require particular care in practice. For instance, often packages require missing values to be coded in specific ways before they can be correctly processed. Also, to analyze longitudinal data, programs often require two identifying variables, usually a time series index and a unit index, like a household ID or a country name, of a specific type and with a particular range. Most statistical packages require the unit index to be of numeric type, with an arbitrary range and order; some packages require the time index to be a sequence starting with 1.

The following sections provide several examples on how features like pattern matching and hashes can be used to address the above mentioned problems, and other problems that could arise in practice, like the creation of dummy variables, and the validation and transformation of data. Several important built-in Perl functions and operator will also be introduced in the examples.

Example of recoding missing values The CSV file format is particularly suited to represent missing values, as they appear as empty (or null) strings delimited by commas. Most statistical packages will not accept a null string (henceforth represented by `""`) to denote a missing value. Different packages use different symbols to denote missing values: some use punctuation marks, like a period, some numbers, like -999, and others strings, like `NA`. Perl can easily recode missing values. Consider, for example, replacing a null string `""` with a `NA` string. In Perl this task can be accomplished by including the following statement after the *split* function:

```
if ( $rgdpch == "" ) { $rgdpch = "NA"; }
```

Example of recoding numerical variables Some statistical packages require the time index variable to be a sequence of integers starting with 1. We can recode the year variable in Perl, for example, by subtracting from it the starting year value minus one. This can be

accomplished simply with the following expression: `$year -= 1949`. The `-=` operator, analogously to C and C++, is simply a shorthand for the *assignment* operator used in conjunction with the *minus* operator to avoid typing the variable name twice. We can also "multiply" with the *repetition* operator, `x`, a sequence going from 1 to 49 obtained using the range operator, `..`, with the expression[27]

```
@trend = (1..49) x 168;
```

Note that a vector of ones can be obtained in a similar way simply with the statement,

```
@ones = (1) x 8232;
```

Example of recoding categorical variables Using regular expressions and hashes we can show how a laborious and error prone task, like converting a categorical variable into a numerical variable, can be automated with Perl. Consider converting the three-letters country code of our example into a three-digits one by using a "dictionary" created from a suitable file. Consider the file containing a list of countries and their associated three-digit numerical codes used for statistical processing purposes by the Statistics Division of the United Nations Secretariat, and three-letter alphabetical codes assigned by the International Organization for Standardization, available from the United Nations Statistics Division Web site.[28] Here follow the first ten lines

```
Numerical
code    Country or area name ISO ALPHA-3 code
004 Afghanistan AFG
008 Albania ALB
012 Algeria DZA
016 American Samoa ASM
020 Andorra AND
024 Angola AGO
660 Anguilla AIA
028 Antigua and Barbuda ATG
```

We can immediately observe that the *split* operator with separator `/\s/`, would not be useful here as it would break up names like "American Samoa." A regular expression can be used to extract the information required from the file. We can use the expression `(\d{3})` to define a pattern that matches three consecutive digits, and assigns the match to the variable `$1`. The expression `([A-Z]{3})` defines a pattern that can be used to capture three adjacent capital letters into the `$3` variable. The country names, not needed in our example, can be captured into the variable `$2` with the pattern `(\D+)`. Combining, these pattern into one expression we can build a dictionary that will do the desired conversion, in this case, from a letter code to a digit code, suitable for merging files and for longitudinal data analysis,

[27] It is considered poor programming practice to have what programmers call "magic numbers" interspersed in the code. In statistical programs this numbers are typically the number of observations, years, and individuals. To improve code readability and to facilitate code reusability, these should be assigned to names of their own like `$nobs`, `$ncontries`, and `$nyears`, respectively. In Perl the length of an array can be obtained evaluating the array in a "scalar context", for example, the length of the GDP array can be obtained using the expression `$nobs = @rgdpch`.

[28] The Web address of the site is: http://unstats.un.org/unsd/methods/m49/m49alpha.htm.

```
open( IN, "iso.out" );
while ( $line = <IN> ) {
    chomp($line);
    $line =~ /(\d{3})\s(\D+)\s([A-Z]{3})/;
    $convert{$3} = $1;
}
close(IN);
```

the dictionary can be used, after reading the longitudinal data file, with the expression

```
$NEWCODE = $convert{$CODE};
```

Another example, where recoding a categorical variable into a numeric variable might be needed, is in the creation of dummy variables. Consider creating a dummy variable that is equal to 1 if a country is an OECD member, and 0 otherwise. The following Perl code defines a function called *isOECD* that can accomplish the task.

```
sub isOECD {
    @oecd = ( aut, bel, cze, dnk, fin, fra, deu, grc, hun, isl,
        irl, ita, lux, nld, nor, pol, prt, esp, svk, swe,
        swz, tur, gbr, aus, jpn, nzl, can, mex, usa, prk
    );
    my ($country) = shift;
    %OECD = map { $_ => 1 } @oecd;
    if ( exists $OECD{ lc($country) } ) {
        return $OECD{ lc($country) };
    }
    else {
        return 0;
    }
}
```

For example, evaluating the statement `isOECD(ITA)` returns 1, whereas evaluating `isOECD(ALG)` returns 0. To create an array containing the dummy variable, we can use,

```
@OECD = map { isOECD($_) } @code;
```

We can also substitute the variable `isocode` "in place," i.e., without making a new copy of the data for output, with the expression,

```
$line =~ s/([A-Z]{3})/isOECD($1)/e;
```

where the `/e` modifier causes the replacement string to be evaluated as Perl code.

Example of transforming variables To transform a variable one can use the *map* operator which applies a specified function to each element of a list. Say we want to apply the usual *log* transformation to a variable. Once missing values have been purged, this can be done in with the following Perl statement:

```
@lgdp = map { log($_) }  @rgdpch;
```

To apply the *log* transformation only to values in its domain of definition, i.e. when the argument is greater then zero, we can use the *conditional* or *ternary operator* (because it takes three arguments), combined with the *map* operator. The ternary operator first evaluates its first argument, a conditional expression. If the argument evaluates to a true value, then the operator returns the second argument. Otherwise it evaluates and returns the value of the third argument. We want to take the *log* of the value only when positive otherwise we want to assign a missing value symbol, say "NA", to the variable, i.e., `$rgdpch>0 ? log($rgdpch) : "NA"`. Combined with *map* the code becomes:

```
@lgdp = map { $_ > 0 ? log($_) : "NA" } @rgdpch;
```

Example of validating data Data can easily be validated using logical operators. These follow the same rules as C, C++, etc.

```
if ( $rgdpch < 0 || $rgdpch > 100000 ) { print "GDP out of range" }
```

7.3. Sorting observations

When processing data, sorting is an important prerequisite for merging data files. Also, it might be required in order to prepare a data file to be fed into a statistical package. For instance, for longitudinal data analysis, all the years should be in chronological order.

In Perl the *sort* operator returns a copy of the array argument, sorted in ascending alphabetic order. Here are some common ways to sort in Perl:

```
(sort {$a cmp $b} @array)  # sort alphabetically, with uppercase first
(sort {$a <=> $b} @array)  # sort numerically
(sort {$b cmp $a} @array)  # sort reverse alphabetically
```

The *sort* operator above passes a "comparison" anonymous function `{...}` to the *sort* operator, where the special variables `$a` and `$b` are the two elements to compare, `cmp` is the built-in *string comparisons* operator, and, `<=>`, is the built-in *numerical comparison* ("spaceship") operator.

For example, if we want to sort our observations using the year as the first sorting key and the country code as the second, we can combine the numerical and alphabetical comparisons and then print the sorted observations with the following lines of code

```
  @index =
  sort { $year[$a] <=> $year[$b] or $code[$a] cmp $code[$b] } 0 .. $#year;
foreach $ind (@index) { print "$code[$ind], $year[$ind], $rgdpch[$ind]\n"; }
```

The first ten lines of output contain:

```
AGO, 1950,
ALB, 1950,
ARG, 1950, 6430.0134743
ARM, 1950,
ATG, 1950,
AUS, 1950, 9173.8190347
```

```
AUT, 1950, 4213.7210261
AZE, 1950,
BDI, 1950,
BEL, 1950, 6099.8246926
```

8. Enhancing Perl with modules

A Perl module is a plain text file, ending with the *.pm* suffix, that contains reusable Perl code. The `use` statement makes the code in the module file available to the rest of the Perl script. Modules greatly extend Perl's functionality in several application areas. Many modules are already included in standard Perl distributions. Installing modules is very simple as most Perl distribution provide *module managers* that directly access the CPAN module repository,[29] and automatically install the required modules.

Interface The way modules can be used depends upon what is known as their *interface*. There are two important interfaces: functional and object-oriented.

Modules with *functional interface* export functions they provide making them accessible just as any other Perl or user defined function. Consider the module `Locale::Country` for ISO codes for country identification. The module provides, amongst other, a function, *country2code*, that converts a country name into its 3 characters or its digits ISO code, depending on the arguments,

```
use Locale::Country;
$codealph = country2code( 'United states of america', LOCALE_CODE_ALPHA_3 );
$codenum  = country2code( 'United states',            LOCALE_CODE_NUMERIC );
```

`$codealph` is now "usa" and `$codenum`, 840. Note that `Locale::Country` supports alternative names for countries.

Modules, with interfaces known as *object-oriented*, provide *methods*. Methods, as functions, have a name and take a number of argument. From a user's point of view, methods differ from functions in the way they are invoked. Implementation and interface are kept well distinct in object oriented programming. Users of modules do not need to know how they are implemented. Statisticians familiar with the S language (and its implementations) should not be new to the concept of object oriented programming. Extensive documentation on how to use hundreds of modules is available at the *Perldoc.com* Web site.

Consider the module `Number::Format` which allows the formatting of numbers in a more flexible fashion than can be done with standard Perl functions. The module can be loaded with

```
use Number::Format;
```

The method *new* in the expression

[29] With ActivePerl, the Perl Package Manager (PPM) is loaded by executing the *ppm* command at the operating system prompt. The PPM is a tool that manages the installation install, removal, and the upgrade, of common Perl CPAN modules. For example, to install the `Locale::Country` module, one can just execute the command *install Locale::Country*.

```
$gdp = Number::Format->new(%options);
```

creates an *object* belonging the *class* `Number::Format`. Key/value pair arguments (`%options`) may be provided to set up the desired number format. The following script

```
$number = 23435464.5433577;
use Number::Format;
$numeraire = Number::Format->new(
    -thousands_sep   => ',',
    -decimal_point   => '.',
    -int_curr_symbol => 'USD'
);
$formatted = $numeraire->format_price($number);
print $formatted;
```

prints `USD 23,435,464.5434`. A method expects an object reference as its first argument. Note that both *new* and *format_price* are methods of the "class" `Number::Format`, only that *new*, the so-called *constructor* method, has the class name as the extra first parameter, whereas the *format_price* method takes an object belonging to that class, i.e., `$numeraire`, as its extra argument. In both methods this extra parameter is passed using a special *arrow notation*. Another useful module is the `Text::CSV_XS` module, which enhances Perl's capabilities in handling CSV files. CSV files might be difficult to process if they include non-separating commas as part of their value, as in *"USSR, Russian Federation"*,

```
$str = 'Russia,"USSR, Russian Federation",922,365,RUS,RUS,143,USSR';
use Text::CSV_XS;
my $csv = Text::CSV_XS->new;
$csv->parse($str);
my @fields = $csv->fields;
(
    $Name,   $Name2,   $IMFcode, $ICPSRcode,
    $WBcode, $McCcode, $PWTcode, $CapCode
  )
  = @fields;
print $Name2;    # prints: USSR, Russian Federation
```

Often, different interfaces are available to the user. Consider the matrix data structure example in Section 5.6 implemented as an anonymous array of anonymous arrays

```
$mat = [ [ 1, 2, 3 ], [ 3, 4, 5 ] ];
```

To print the matrix in its entirety, we can use the `Data::Dumper` module. We can use the functional interface, by calling the *Dumper* function

```
use Data::Dumper;
print Dumper($mat);
```

```
use Data::Dumper;
print Dumper($mat);
```

```
$VAR1 = [
        [
          1,
          2,
          3
        ],
        [
          3,
          4,
          5
        ]
      ];
```

The same can be done using the object oriented interface

```
$d = Data::Dumper->new( [$mat] );
print $d->Dump;
```

9. WWW interfacing with Perl

The Internet has become an essential tool to assist in the preparation and completion of a statistical project. Students and practicing statisticians regularly access the Internet to locate data, working papers, and statistical software. The Internet has also facilitated various forms of collaborative work among statisticians and is widely used to disseminate results.

Perl is widely used to write server-side scripts such as interactive Web pages via the CGI (Common Gateway Interface) protocol. Server-side applications can exploit the Internet's wide reach and popularity as a medium for surveys, experiments, and learning. An example of such applications is Berkeley's Internet Virtual Laboratory[30] (IVLab), where the possibilities of employing the Web as a source of survey data is investigated. Another example Rweb,[31] a Web based interface to R (the GNU implementation of the S programming language) were R code can be modified or typed into a Web form and then submitted for on-the-fly execution. CGI.pm is the standard Perl module for creating and parsing CGI forms. For more information on using Perl for server processes consult Guelich, Gundavaram, and Birznieks (2000).

Perl can be used to write client-side processes as well. These processes can, for example, mimic a "browser" as it collects information from a remote server using HTTP. It is possible to check for software and data updates, list recent publications in a specific topic, etc. LWP (short for "Library for World Wide Web in Perl") is a collection of Perl modules for searching and collecting data from the Web and for extracting information from HTML pages. For example, the module LWP::Simple provides several functions for fetching Web pages. The statements

```
use LWP::Simple;
$data = get('http://lib.stat.cmu.edu/datasets/boston');
```

assign the Boston house-price data from the StatLib dataset archive to the variable `$data`. The variable can then be processed further within Perl.

[30] Located at http://emlab.berkeley.edu/ivlab/welcome.html.
[31] Located at http://calculators.stat.ucla.edu/Rweb/Rweb.JavaScript.html.

As another more sophisticated example of data collection consider getting the latest Euro rate against the US dollar from Yahoo!Finance. Here we use the `Finance::YahooQuote` module,

```
use Finance::YahooQuote;
print @{ getonequote("^XEU") }[2];
```

which printed, at the time of writing the review, 1.1516.

Another important use of the Web is to track down relevant information through *search engines*. Using the `SOAP::Lite` module,[32] which has an object-oriented interface, a search string can be easily posted on a Web search engine and the results of the search can be directly processed in Perl. Consider assessing the popularity of several programs used by statisticians by the number of pages found by the Google search engine,

```
use SOAP::Lite;
$key = $ENV{GOOGLEKEY};    # needed to access SOAP API at Google
$google = SOAP::Lite->service('file:./GoogleSearch.wsdl');
foreach $query (qw/matlab minitab perl sas s-plus spss stata/) {
    $result = $google->doGoogleSearch(
        $key,    $query, 0,        10, 'false', '',
        'false', '',     'latin1', 'latin1'
    )->{'estimatedTotalResultsCount'};    # see Google docs
    printf( "%8s: %9d\n", $query, $result );
}
```

which produces the following output

```
  matlab:   1160000
 minitab:    123000
    perl:  12600000
     sas:   4040000
  s-plus:    133000
    spss:    726000
   stata:   3540000
```

Note that this example requires the *GoogleSearch.wsdl* file in the current directory (a different path could be specified), as well as a Google SOAP API access key (here retrieved from the environment variable `GOOGLEKEY`), both available free of charge from http://www.google.com/apis/. Note that the `LWP` module can also be used to retrieve the data of the last two examples.

Robots, to search through the Web and collect data can also be easily implemented using LWP. Robot applications cover such activities as searching, and surveying. Robots can be programmed to collect statistics, or surf the Web and summarize their findings for a search engine. These capabilities are provided by the `LWP::RobotUA` module. For more information on the LWP modules see Burke (2002). Other modules provide simple interfaces to e-mail, telnet, FTP, and other network services that can facilitate collaboration between researchers, through synchronous and asynchronous forms of communication, and spread computations and storage across different physical locations on the Web. See Stein (2001) for such examples of network applications.

[32] The use of this module was suggested by an anonymous referee.

Part II
Statistical Computing

10. Introduction

Part I provided the basic introduction to the Perl programming language, and showed how it can be used to process data for statistical purposes. In particular, we showed how Perl's process, file, and text manipulation facilities, can be used for a wide range of data processing tasks that Statisticians face regularly, such as data transformation, validation and recoding. We also showed how modules, which enhance Perl's in many areas of application, can be used in the collection, preparation, and formatting of data for statistical analysis. In Part I, Perl was used mostly as a scripting language stressing its nature of "glue language" and its built-in specialization in string manipulation through regular expression and hashes. Here in Part II we show how Perl's can be enhanced, through extension modules, to perform simple statistical analysis and more powerful statistical computations.

There are many Perl modules that can be used for statistical computations. A search on the CPAN web site returned, amongst others, the following modules potentially of interest to Statisticians with different level of sophistication:

```
Statistics::Contingency, Statistics::OLS, Statistics::Descriptive, Statistics::TTest,
Statistics::Descriptive::Discrete, Statistics::GammaDistribution, Statistics::PointEstimation,
Statistics::Distributions, Statistics::ChiSquare, Statistics::DependantTTest,
Statistics::Frequency, Statistics::Table::F, Math::CDF, Math::Matrix, Math::MatrixReal,
PDL::R::math, and PDL::Matrix.
```

Even from a quick inspection of the list, it is apparent that different modules can provide overlapping functionality. Indeed, statistical modules, such as `Statistics::Distributions` and `Math::CDF`, all provide tools for computing with statistical distributions. From an implementation point of view, modules providing similar functionality might differ in their interface, the algorithms implemented, design, and so on. From a user point of view, the choice of which module to employ will depend upon several factors, such as the field of application, ease of use, efficiency, and sophistication of the user.

Given the open source nature of Perl, considerable information about these issues can be gathered form inspecting the Perl code directly. Even so, testing can still provide critical information for several reasons. For example, visual code inspection is not always practical as it might be too time consuming and require a considerable knowledge of Perl and its different programming styles. Also, even though a particular algorithm is theoretically sound, it is important to assess whether it has been implemented correctly and efficiently in the Perl module under scrutiny. Another potential benefit of thoroughly testing the modules using a standard battery of tests is that it allows to make comparisons with other software packages useful for statistics that have already been tested for reviews in specialized journals.

Given the plethora of modules currently available, it is difficult, if not impossible, to review all modules potentially useful for doing statistics. Besides, any such information would rapidly become outdated as new modules and module updates become available. We decided to survey what seem to be the most "obvious" statistical modules, either because they appear

more readily in simple searches on CPAN, for example, or because mentioned more often in the Perl literature.

11. Reproducible statistical computations using Perl

Claerbout (see, e.g., Buckheit and Donoho 1995), has recently championed the issue of reproducibility in the computational sciences. Reproducing statistical computation results from published work often proofs to be a difficult and daunting task. Reproducibility relies on a plethora of implementation details that are difficult to communicate through conventional printed publications.

Statistics is in essence computation on data. For a statistical project to be independently reproducible, a prerequisite is that adequate documentation on the dataset used, its sources, the processing it has been subjected to, as well as documentation on software and programming code used for its analysis, should be made available. The Internet has become an important medium for sharing data, software, and to communicate scientific research, making it an ideal environment for providing such information.

Perl's extensive network programming support, allows to collect in a short script, not only details about data processing and the code needed for the analysis, but also, for instance, the commands that retrieve the data itself and the software used for its analysis. A complete statistical project with all the data and commands needed for its reproduction can be compactly preserved and effectively communicated through a Perl script. The following example illustrates this point. The example requires only R and LaTeX to be reproduced.

Consider estimating the demand for clean air model using the well-known Boston housing data (see Belsey, Kuh, and Welsch 1980, for example). Note that in the dataset available at STATLIB each record spans over two contiguous lines.

```
 0.00632  18.00   2.310  0  0.5380  6.5750  65.20  4.0900   1  296.0  15.30
  396.90   4.98  24.00
 0.02731   0.00   7.070  0  0.4690  6.4210  78.90  4.9671   2  242.0  17.80
  396.90   9.14  21.60
...
```

The script downloads the Boston data from STATLIB, cleans and prepares the dataset, uses R to estimate the coefficients of the model, saves the results in a file, and finally prints the results in LaTeX format.

```
# downloads Boston dataset from STATLIB
use LWP::Simple;
getstore(
    "http://lib.stat.cmu.edu/datasets/boston",
    "boston.raw" );

# corrects for record spanning two lines
open( IN,  "boston.raw" );
open( OUT, ">boston.asc" );
do { $line = <IN> } until $. == 22
or eof;    # Skips the first 22 lines of header
while ( $line = <IN> ) {
    chomp $line;
    $line .= <IN>;    # joins two lines
    print OUT $line;
}
close(OUT);

# sends data to R for regression analysis
open( RPROG,
"| c:/rw1081/bin/Rterm.exe --no-restore --no-save"
);
select(RPROG);
print <<'CODE';
bost<-read.table("boston.asc",header=F)
names(bost)<- c( "CRIM",  "ZN",    "INDUS",
                 "CHAS",  "NOX",   "RM",
                 "AGE",   "DIS",   "RAD",
                 "TAX",   "PTRAT", "B",
                 "LSTAT", "MEDV")
attach(bost)
boston.fit <- lm(log(MEDV) ~ CRIM + ZN + INDUS +
   CHAS + I(NOX^2) + I(RM^2) + AGE + log(DIS) +
   log(RAD) + TAX + PTRAT + B + log(LSTAT))
sum <- summary(boston.fit)$coe[,1:2]
write.table(sum,"boston.out",quote = FALSE)
q()
CODE
close(RPROG);

# creates LaTeX table with regression results
open( TABLE, "boston.out" );
open( TEX,   ">table.tex" );
$prec  = 3;    # sets number of decimals
$width = 9;    # sets the width of the field
do { <TABLE> } until $. == 1 or eof;    # Skips the first  line of header
while (<TABLE>) {
    chomp;
    @line = split;
    printf TEX "%11s & %${width}.${prec}g & %${width}.${prec}g\\\\ \n", $line[0],
      $line[1], $line[2];
}
close(TABLE);
```

Table II: OLS estimates of the demand for clean air model

Variable	Coefficient estimate	Standard error
(Intercept)	4·56	0·154
CRIM	−0·0119	0·00124
ZN	$8{\cdot}02e-005$	0·000506
INDUS	0·00024	0·00236
CHAS	0·0914	0·0332
I(NOX^2)	−0·638	0·113
I(RM^2)	0·00633	0·00131
AGE	$9{\cdot}07e-005$	0·000526
log(DIS)	−0·191	0·0334
log(RAD)	0·0957	0·0191
TAX	−0·00042	0·000123
PTRATIO	−0·0311	0·00501
B	0·000364	0·000103
log(LSTAT)	−0·371	0·025

```
close(TEX);
```

The TEX file can be included in a document using the command `\input{table}`. The result, after compilation in LATEX is the typeset Table II.

These results are easily reproducible, as they can be generated in a matter of minutes and do not require proprietary data or licensed software.

Another illustration of reproducible statistical computations using Perl, is provided in Appendix D. The script downloads and installs the **DIEHARD** test programs, produces a file containing 3 millions random numbers in hex format, transforms the file into a binary format using an auxiliary program, sends the binary file to the main test program, extracts from the output the computed p-values,[33] uses the `Statistics::Descriptive` module to obtain a frequency table, prints the results with tabular format in a TEX file. Explanation of most of the code is provided in this paper.

12. Assessing the numerical reliability of modules

To assess the numerical and probabilistic reliability of the Perl modules we follow McCullough (1998) proposed methodology. This testing methodology focuses on three features of statistical software:

(i) estimation, using the Statistical Reference Datasets[34] (StRD) (Rogers, Filliben, Gill, Guthrie, Lagergren, and Vangel 1998) from the National Institute of Standards and

[33] The standard practice of doubling the one-sided p-value to obtain a two-sided p-value is not appropriate for an asymmetric distribution. The CDF scale can be conveniently used to assess the results of the tests. A large CDF-value corresponds to a small p-value and indicates a significant result. A small CDF-value indicates a close match between observed and expected frequencies. In theory, if the CDF-value is less that 0.025 or greater than 0.975 we reject the null at the 5 per cent significance level, however, in practice, because of the large number of tests, unless stated differently, a test is considered failed if all CDF-values are either zero or one.

[34] Available at the web address http://www.itl.nist.gov/div898/strd/.

Technology (NIST) to evaluate the accuracy of univariate summary statistics and linear regression;

(ii) statistical distributions, using the exact values, computed with ELV (Knüsel 1989) to verify the accuracy of statistical distributions computations; and

(iii) random number generation, using the **DIEHARD** (Marsaglia 1996) Battery of Randomness tests to determine whether random numbers are seem to behave as independent samples uniformly distributed over (0,1).

Accuracy of the estimates will be measured by the base-10 logarithm of the relative error (LRE) given by the formula

$$lre(q, c) = -\log_{10}\left(\frac{|q - c|}{|c|}\right), \tag{1}$$

where q represents the estimated value and c the correct value. When the two values are sufficiently close, the LRE is a measure of the number of correct significant digits. The implementation in Perl used for this paper that allows for cases where *lre* function is undefined and checks for closeness of estimated and correct values, is provided in Appendix A.

13. Speed of execution benchmarking in Perl

As often there are several Perl modules providing similar functionality. Also there are many ways in which the same task can be accomplished. It is therefore, essential to be able to assess the relative efficiency in terms of speed of execution, of various modules and approaches. To compare the execution speed of different implementations we can use the `Benchmark` module. The module is part of the standard Perl distribution. The `Benchmark` module provides a number of routines to time the execution of Perl code. For example, *timethis*, times the time it takes to run a block of code for a specified number of times, *timethese*, does the same for several blocks of code, and *cmpthese*, conveniently prints the results of *timethese* as a comparison chart. Consider comparing two ways taking the natural log of an array, one using the *map* operator and the other using the for `loop`. This can be achieved with the following block of code.

```
use Benchmark qw( timethese cmpthese );
@vect = ( 2.5, 3, 4.3, 5, 6.3, 34, 2323232, 0.000000032 );
$results = timethese(
    1e6,
    {
        MAP => 'map { log($_) } @vect;',
        FOR => 'for ($i = 0; $i <= $#vect; $i++)  { log($vect[$i]) }'
    }
);
cmpthese($results);
```

The above script returns the output

```
Benchmark: timing 1000000 iterations of FOR, MAP...
FOR: 15 wallclock secs (15.83 usr +  0.00 sys = 15.83 CPU) @ 63179.18/s
```

```
   (n=1000000)
MAP:  7 wallclock secs ( 7.94 usr +  0.00 sys =  7.94 CPU) @ 125992.19/s
   (n=1000000)
       Rate  FOR  MAP
FOR  63179/s   -- -50%
MAP 125992/s  99%   --
```

From these results we can conclude that the transformation using the map operator is 50 per cent faster than the for loop.

14. A Survey of modules for statistical analysis

14.1. Descriptive statistics

The `Statistics::Descriptive` module provides basic methods useful for descriptive statistics. It has an object oriented design. The following example uses the Michelso NIST benchmark dataset for univariate summary statistics. The certified values to 15 figures for the sample mean and sample standard deviation were also obtained from the same source.

```
@Michelso = qw(
    299.85 299.74 299.90 300.07 299.93 299.85 299.95 299.98 299.98 299.88 300.00 299.98
    299.93 299.65 299.76 299.81 300.00 300.00 299.96 299.96 299.96 299.94 299.96 299.94
    299.88 299.80 299.85 299.88 299.90 299.84 299.83 299.79 299.81 299.88 299.88 299.83
    299.80 299.79 299.76 299.80 299.88 299.88 299.88 299.86 299.72 299.72 299.62 299.86
    299.97 299.95 299.88 299.91 299.85 299.87 299.84 299.84 299.85 299.84 299.84 299.84
    299.89 299.81 299.81 299.82 299.80 299.77 299.76 299.74 299.75 299.76 299.91 299.92
    299.89 299.86 299.88 299.72 299.84 299.85 299.85 299.78 299.89 299.84 299.78 299.81
    299.76 299.81 299.79 299.81 299.82 299.85 299.87 299.87 299.81 299.74 299.81 299.94
    299.95 299.80 299.81 299.87
);
```

To calculate the sample mean and sample standard deviation we need the following code.

```
use Statistics::Descriptive;
$stat = Statistics::Descriptive::Full->new();
$stat->add_data(@Michelso);
$mean = $stat->mean();
$stat->standard_deviation();
$Statistics::Descriptive::Tolerance = 1e-10;
```

The *mean()* method is used to obtain the mean. To obtain the standard deviation, the *standard_deviation()* method is used. The results of the calculation, along with the results from other NIST test data of higher difficulty, are presented in Table III. The results from the module are quite accurate and almost identical to the performance of statistical packages like S-PLUS and SPSS (see, McCullough 1999). The sample mean is always accurately computed. The sample standard deviation, as the complexity of the test data increases, is calculated with decreasing accuracy. A quick look at the code reveals that an accurate one-pass algorithm is implemented. This implementation retains the speed of a standard one-pass algorithm and still seeks the accuracy of the two-pass algorithm.

Table III: Numerical accuracy of the `Statistics::Descriptive` module

StRD (difficulty)	Sample statistic	Module estimate	Certified value	LRE
Michelso	mean	299·8524	299·852400000000	15
(low)	s.d.	0·0790105478190846	0·0790105478190518	12·38227
NumAcc3	mean	1000000·2	1000000·2	15
(medium)	s.d.	0·100000000035104	0·1	9·45464
NumAcc4	mean	10000000·2	10000000·2	15
(high)	s.d.	0·100000000558905	0·1	8·25266

Frequency distribution example As we will see in Section 14.6, the output of the **DIEHARD** tests is a file containing several p-values interspersed in the other test output. Under the respective null hypotheses, all of the p-values are independent and uniformly distributed over the interval $(0, 1)$. Consider testing informally this hypothesis by construction a frequency distribution.

As the battery of tests can return more than 200 p-values, if done for several generators, the process of manually collecting, summarizing, and elaborating the results can become a quite tedious and error prone task. Here we show how Perl can be used to "capture" the p-values, store them into an array, and then, using the `Statistics::Descriptive` module, construct a frequency distribution table to assess whether the observed p-values follow a uniform distribution. In the output file, the p-values are reported using different printing formats. Here is a representative list

```
p-value  .098288
p-value: .781201
p-value= .73935
p-values:  .424192
pvalue= .48598
p=  .33395
```

Let us assume for simplicity that this list exhausts all possible cases. A basic Perl regular expression to subsume the commonality in all of the above expressions can be defined as follows:

```
m/
    p           # mandatory character 'p'
    -?          # optional '-'
    (?:         # starts optional non-remembered group
       values? # text string with optional 's'
    )?          # ends optional non-remembered group
    [:=]?       # optional ':' or '='
    \s*         # optional white space
    (           # capture in $1
       \.       # mandatory decimal point
       \d+      # mandatory decimal number
    )
/gx
```

Note that the `/x` pattern modifier allows white space can be embedded in the regular expression without altering its meaning. Since comments are treated as white space, the regular

expression can be extensively commented. The `/g` pattern modifier finds all occurrences. Consider processing the *TT800.out* file (see Section 14.6.3 on page 51).

```
open( IN, "TT800.out" );
$count = 0;
while ( $line = <IN> ) {
    if ( $line =~ m/p-?(?:values?)?[:=]?\s*(\.\d+)/ ) {
        $count++;    # counts matches
        push ( @p_vals, $1 );    # stores p-values
    }
}
```

These patterns could be refined, expanded, and layered further to match exactly all occurrences of the p-values. Now, the function *frequency_distribution()* can be used to

```
use Statistics::Descriptive;
$stat = Statistics::Descriptive::Full->new();
$stat->add_data(@p_vals);
%f = $stat->frequency_distribution(10);
```

To print the frequency distribution in order we can use the following block of code

```
print "Total matches: $count\n";
for ( sort { $a <=> $b } keys %f ) {
    $co = 100 * $f{$_} / $count;
    printf "key = %2.1f, count = %2.0f\n", $_, $co;
}
```

which produces the output

```
Total matches: 81
key = 0.1, count = 12
key = 0.2, count = 15
key = 0.3, count =  9
key = 0.4, count =  6
key = 0.5, count = 17
key = 0.6, count =  7
key = 0.7, count =  7
key = 0.8, count =  9
key = 0.9, count =  7
key = 1.0, count = 10
```

With discrete or discretized data, a much more efficient option for descriptive statistics, both in terms of memory use and speed, is provided by the `Statistics::Descriptive::Discrete` module.

14.2. Two-sample t test

The `Statistics::TTest` module can be used to compare two independent samples. It takes two array of point measures and requires the `Statistics::PointEstimation` module to compute confidence intervals, the t-statistic, and p-value, needed to test the null hypothesis of no difference in means. There are no NIST dataset for a simple t test. However, since t tests

are just a special case of ANOVA when there are only two levels, we can use the AtmWtAg ANOVA balanced dataset from NIST, which consists of 48 observations on one factor with 2 levels. This test dataset has a difficulty rating of "average." The square of the computed sample t-statistic can be compared with the the certified F-statistic. The following script can be used to perform a two-sample t test analysis.

```
use Statistics::PointEstimation;
use Statistics::TTest;

@r1=(107.8681568, 107.8681465, 107.8681572, 107.8681785, 107.8681446, 107.8681903,
    107.8681526, 107.8681494, 107.8681616, 107.8681587, 107.8681519, 107.8681486,
    107.8681419, 107.8681569, 107.8681508, 107.8681672, 107.8681385, 107.8681518,
    107.8681662, 107.8681424, 107.8681360, 107.8681333, 107.8681610, 107.8681477);

@r2=(107.8681079, 107.8681344, 107.8681513, 107.8681197, 107.8681604, 107.8681385,
    107.8681642, 107.8681365, 107.8681151, 107.8681082, 107.8681517, 107.8681448,
    107.8681198, 107.8681482, 107.8681334, 107.8681609, 107.8681101, 107.8681512,
    107.8681469, 107.8681360, 107.8681254, 107.8681261, 107.8681450, 107.8681368,);

$ttest = new Statistics::TTest;
$ttest->set_significance(90);
$ttest->load_data( \@r1, \@r2 );
$ttest->output_t_test();
$ttest->set_significance(99);
$ttest->print_t_test();    # list out t-test related data
```

The output includes the p-value of the statistic, the confidence interval, and a test for the equality of the variances. The summary output of the test (shortened for space reasons) is shown below.

```
*************************************************

Summary  from the observed values of the sample 1:
sample size= 24 , degree of freedom=23
mean=107.868153766667 , variance=1.70644927517612e-010
standard deviation=1.30631132398679e-005 , standard error=2.66649682415596e-006
the estimate of the mean is 107.868153766667 +/- 4.5701089069209e-006
or (107.868149196558 to 107.868158336776 ) with 90 % of confidence
t-statistic=T=40453134.1607019 , Prob >|T|=1.62447832963153e-012
*************************************************

Summary  from the observed values of the sample 2:
sample size= 24 , degree of freedom=23
mean=107.868136354167 , variance=2.85666938357049e-010
standard deviation=1.69016844828274e-005 , standard error=3.45004189803694e-006
the estimate of the mean is 107.868136354167 +/- 5.91302680904552e-006
or (107.86813044114 to 107.868142267193 ) with 90 % of confidence
t-statistic=T=31265746.7770299 , Prob >|T|=1.62436730732907e-012
*************************************************

Comparison of these 2 independent samples.
F-statistic=1.67404295288863 , cutoff F-statistic=2.0144 with alpha level=0.1 and  df =(23,23)
equal variance assumption is accepted(not rejected) since F-statistic < cutoff F-statistic
degree of freedom=46 , t-statistic=T=3.99333614216051 Prob >|T|=0.000236340000000057
the null hypothesis (the 2 samples have the same mean) is rejected since the alpha level is 0.1
difference of the mean=1.74124999858805e-005, standard error=4.36038924999182e-006
the estimate of the difference of the mean is 1.74124999858805e-005 +/- 7.31978543396126e-006
or (1.00927145519192e-005 to 2.47322854198417e-005 ) with 90 % of confidence
```

Table IV: Independent samples t test using the *AtmWtAg* dataset

Module	Module estimate	Certified F-statistic	LRE
`Statistics::TTest`	15·9467335442853	15·9467335677930	8·83146
`Statistics::Table::F`	14	15·9467335677930	0·91337

Table V: Upper-tail probability of standard normal distribution

x	Module estimate	Exact ELV value	LRE
3·0	1·34990e−003	1·34990e−003	6·0000
4·0	3·16710e−005	3·16712e−005	5·1996
5·0	2·86650e−007	2·86652e−007	5·1563
8·0	6·22100e−016	6·22096e−016	5·1918
9·0	1·12860e−019	1·12859e−019	5·0525
9·7	1·50750e−022	1·50749e−022	5·1783
9·8	5·62930e−023	5·62928e−023	5·4494
9·9	2·08140e−023	2·08138e−023	5·0173
10·0	7·61990e−024	7·61985e−024	5·1830
11·0	1·91070e−028	1·91066e−028	4·6791

Table IV shows the certified F-statistic calculated by NIST and the results of the t test using the `Statistics::TTest` and the `Statistics::Table::F` modules. The latter module performs a one-way analysis of variance (see Section 14.4). The computed LRE show that the t test module is quite accurate. Note that the ANOVA module fails to pass the medium difficulty test.

14.3. Statistical distributions

For simple statistical analysis, in order to test hypothesis, statistical distribution functions that return critical values and calculate p-values are required. Perl provides several packages that can perform statistical computations. The most obvious choice, from searching CPAN, seems to be the `Statistics::Distributions` module. The `Statistics::Distributions` module provides functions to compute the upper-tail quantiles and upper-tail probabilities for the Normal, Student's t, chi-square, and F distributions. The module documentation claims that results are computed with 5 significant figures. It is not clear how many are correct and for what range of values the results are acceptable. A few examples of testing distributions will be given.

Standard normal distribution The function *Statistics::Distributions::uprob ($x)* computes the probability that random variable distributed standard normal is greater than $$x$$. Table V reports the estimated values, the exact values, computed by ELV (Knüsel 1989), and the LRE.

Student's t distribution The function *Statistics::Distributions::tprob ($df,$x)* computes the probability that an t-distributed random variable with $$df$$ degrees of freedom is greater

Table VI: Upper-tail probability of Student's t distribution

x	d.f.	Module estimates	Exact ELV value	LRE
3·0	100	$1{\cdot}70400e-03$	$1{\cdot}70396e-03$	4·6294
4·0	100	$6{\cdot}07620e-05$	$6{\cdot}07618e-05$	5·4826
8·0	100	$1{\cdot}13640e-12$	$1{\cdot}13643e-12$	4·5784
9·0	100	$7{\cdot}60500e-15$	$7{\cdot}68039e-15$	2·0081
9·7	100	$2{\cdot}22040e-16$	$2{\cdot}25155e-16$	1·8590
9·8	100	$1{\cdot}66530e-16$	$1{\cdot}35896e-16$	0·6470
9·9	100	$5{\cdot}55110e-17$	$8{\cdot}20226e-17$	0·4905
10·0	100	0	$4{\cdot}95084e-17$	0
10·1	100	0	$2{\cdot}98859e-17$	0

Table VII: F distribution

x	n_1	n_2	$P(X>x)$ fprob	$P(X>x)$ exact
1	1000	1000	1/2	0.5000
1	2000	2000	1/2	0.5000
1	2100	2100	1/2	0.0000
1	3000	3000	1/2	0.0000

than $\$x$. The following code prints a table of x values and the corresponding upper-tail probabilities.

```
use Statistics::Distributions;
@x = ( 3, 4, 8, 9, 9.7, 9.8, 9.9, 10, 10.1 );
foreach $x (@x) {
    $tprob = Statistics::Distributions::tprob( 100, $x );
    print "$x $tprob \n";
}
```

Table VI shows the results of the computations, the ELV exact values, and the LRE. The estimated tail probabilities are rather inaccurate in the extreme tails. Figure 3 provides more comprehensive view of the function's numerical accuracy.[35] For comparison, we note that R matches all the ELV exact values in the above tests.

General Comment The accuracy seems adequate for most practical situations. For more sophisticated uses it might be advisable to use another module. There is a problem with the order of the arguments of the functions. The variable comes after the parameters of the distributions. This is both non conventional and mathematically unacceptable. User of other statistical programs could find this confusing. An optional noncentrality parameter is missing, although the noncentral Student's t distribution plays an important role in statistical testing theory. A better option is to use the `Math::CDF` module which provides a Perl interface to the DCDFLIB library of C routines for cumulative distribution functions, quantile functions, etc.

[35] The plot was done using the **lattice** R library.

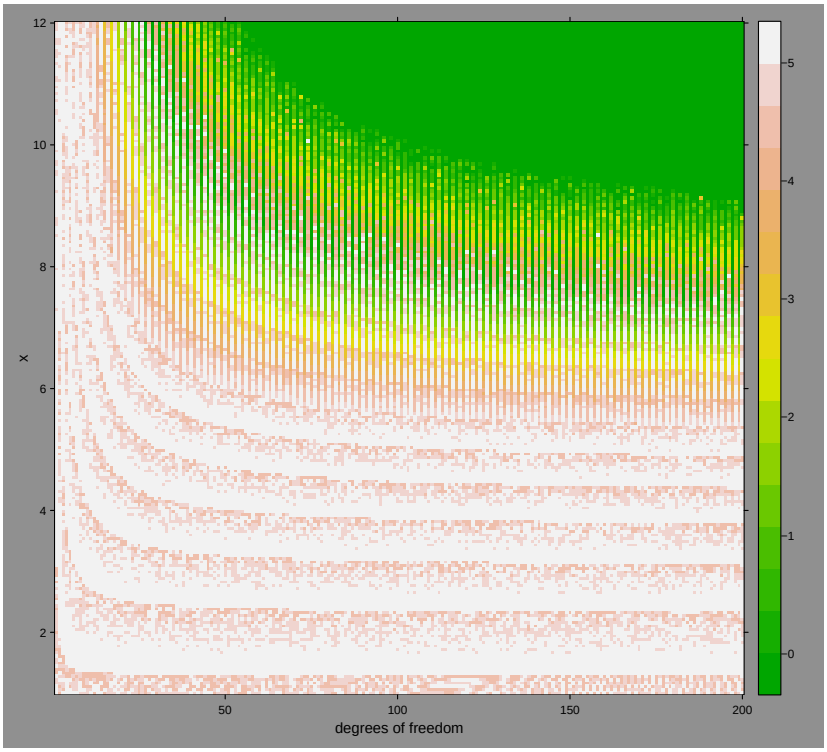


Figure 3: LRE of Student's t distribution for a range of x's and degrees of freedoms

This library is also used, as an alternative to ELV, in assessing the reliability of distribution computations. More information on the library can be obtained at `http://www.netlib.org/`.

Another approach to compute with distributions, is to use special functions such as the logarithm of the gamma function and the incomplete beta function (see, Abramowitz and Stegun 1965). The logarithm of the gamma function can be used for calculating the pdf of Student's t distribution, the F distribution, the chi-square distribution and others. The CDF for Student's t distribution, the F distribution, the binomial distribution and the negative binomial distribution is evaluated using the incomplete beta function. The CDF for the chi-square distribution can be evaluated using the incomplete gamma function (see, Press, Teukolsky,

Vetterling, and Flannery 1995). The module `Statistics::ROC` provides, among others, the logarithm of the gamma function, the incomplete beta function, and their inverses. Consider computing the upper-tail probability for the binomial distribution. The following script uses the incomplete beta function to compute the upper-tail binomial cumulative distribution function with $n = 19$, $x = 15$, and $p = 0.5$.

```
use Statistics::ROC;
$n = 19;
$x = 15;
$p = 0.5;
$y = Betain( $p, $x, $n - $x + 1 );
```

The statement `print $y` returns 0.00960540771503556 which, according to ELV, is correct to at least 6 significant digits.

14.4. ANOVA and Kruskal-Wallis tests

The module `Statistics::Table::F` provides the function *anova* which returns the one-way analysis of variance F-statistic. The name of the function is misleading as no estimated variance components are returned. The following examples obtain the variance components needed to construct ANOVA tables, by slightly modifying the *anova* function (see Orwant *et al.* 1999, pp. 617–620). Consider the SiRstv ANOVA balanced dataset from NIST, which consists of 25 observations on one factor with 5 levels. This test dataset has a difficulty rating of "low." The following script can be used to obtain the F-statistic of the one-way ANOVA analysis.

```
use Statistics::Table::F;
$SiRstv = [
    [ 196.3052, 196.1240, 196.1890, 196.2569, 196.3403 ],
    [ 196.3042, 196.3825, 196.1669, 196.3257, 196.0422 ],
    [ 196.1303, 196.2005, 196.2889, 196.0343, 196.1811 ],
    [ 196.2795, 196.1748, 196.1494, 196.1485, 195.9885 ],
    [ 196.2119, 196.1051, 196.1850, 196.0052, 196.2090 ]
];
print anova($SiRstv);
```

The `print` statement returns 1.18046238617325. Table VIII shows the ANOVA tables obtained by applying the modified *anova* function on two NIST dataset. The Table reports also the certified values calculated by NIST, and the LREs of the F-statistics. The results show that the module does not pass the medium difficulty problem. It should be kept in mind, that other well known statistical packages perform similarly on these tests (see, McCullough 1999).

The `Statistics::KruskalWallis` module can be used to perform the Kruskal-Wallis rank sum test of the null that the location parameters are the same for three or more groups of unequal sizes. Consider using the SiRstv dataset. The following script

```
use Statistics::KruskalWallis;
@group_1 = ( 196.3052, 196.124,  196.189,  196.2569, 196.3403 );
@group_2 = ( 196.3042, 196.3825, 196.1669, 196.3257, 196.0422 );
@group_3 = ( 196.1303, 196.2005, 196.2889, 196.0343, 196.1811 );
```

Table VIII: Numerical accuracy of the `Statistics::Table::F` module

StRD (diffic.)	Source of variation	d.f.	Sums of squares	Mean square	*F*-statistic	LRE
			Certified values			
SiRstv (low)	**between**	4	0·0511462616000000	0·0127865654000000	1.18046237440255	8.00125
	within	20	0·216636560000000	0·0108318280000000		
			`Statistics::Table::F` module			
	between	4	0·0511462620925158	0·012786565523129	1.18046238617325	
	within	20	0·216636559925973	0·0108318279962987		
			Certified values			
SmLs04 (medium)	**between**	8	1·68000000000000	0·210000000000000	21.0000000000000	0
	within	180	1·80000000000000	0·0100000000000000		
			`Statistics::Table::F` module			
	between	8	1·40625	0·17578125	17.7631578947368	
	within	180	1·78125	0·00989583333333333		

```
@group_4 = ( 196.2795, 196.1748, 196.1494, 196.1485, 195.9885 );
$kw = new Statistics::KruskalWallis;
$kw->load_data( 'group 1', @group_1 );
$kw->load_data( 'group 2', @group_2 );
$kw->load_data( 'group 3', @group_3 );
$kw->load_data( 'group 4', @group_4 );
( $H, $p_value ) = $kw->perform_kruskal_wallis_test;
print "Kruskal Wallis statistic is $H\n";
print "p value for test is $p_value\n";
```

returns the output

```
Kruskal Wallis statistic is 3.43428571428571
p value for test is 0.32939
```

The Newman-Keuls test to test differences between pairs of groups is also implemented in the module. The following block of code can be used to test differences between groups 1 and 2, 2 and 3, and 3 and 4. This is done here just for illustrative purposes; the results from the ANOVA and the Kruskal-Wallis test do not warrant such an investigation.

```
( $q, $p ) = $kw->post_hoc( 'Newman-Keuls', 'group 1', 'group 2' );
print "Newman-Keuls statistic for groups 1, 2 is $q, p value $p\n";
( $q, $p ) = $kw->post_hoc( 'Newman-Keuls', 'group 2', 'group 3' );
print "Newman-Keuls statistic for groups 2, 3 is $q, p value $p\n";
( $q, $p ) = $kw->post_hoc( 'Newman-Keuls', 'group 3', 'group 4' );
print "Newman-Keuls statistic for groups 3, 4 is $q, p value $p\n";
```

The output is

```
Newman-Keuls statistic for groups 1, 2 is -0.0534522483824847, p value <0.1
Newman-Keuls statistic for groups 2, 3 is 1.12249721603218, p value <0.1
Newman-Keuls statistic for groups 3, 4 is 0.374165738677394, p value <0.1
```

Table IX: Numerial accuracy of the `Statistics::OLS` module

Variable	Module estimate	Certified value	LRE
constant	−0.26232307377389	−0.262323073774029	12.2758219338587
slope	1.00211681802045	1.00211681802045	15

14.5. Simple linear regression

Currently Perl modules do not provide multiple regression estimation methods. The module `Statistics::OLS` performs a simple linear regression estimation. The module has an object-oriented interface and computes slope and intercept of the regression line. Other regression statistics such as t-statistics, R^2, standard error of regression, Durbin-Watson, predicted values, and residuals, are also provided. The following example uses the Norris benchmark NIST dataset, the only test data suitable for simple linear regression.

```
@xdata =
  qw( 0.2 337.4 118.2 884.6 10.1 226.5 666.3 996.3 448.6 777.0 558.2 0.4 0.6
  775.5 666.9 338.0 447.5 11.6 556.0 228.1 995.8 887.6 120.2 0.3 0.3 556.8 339.1 887.2
  999.0 779.0 11.1 118.3 229.2 669.1 448.9 0.5);
@ydata =
  qw( 0.1 338.8 118.1 888.0 9.2 228.1 668.5 998.5 449.1 778.9 559.2 0.3 0.1
  778.1 668.8 339.3 448.9 10.8 557.7 228.3 998.0 888.8 119.6 0.3 0.6 557.6 339.3 888.0
  998.5 778.9 10.2 117.6 228.9 668.4 449.2 0.2);

use Statistics::OLS;
$ls = Statistics::OLS->new();
$ls->setData( \@xdata, \@ydata );
$ls->regress();
( $intercept, $slope ) = $ls->coefficients();
$R_squared = $ls->rsq();
( $tstat_intercept, $tstat_slope ) = $ls->tstats();
$sigma         = $ls->sigma();
$durbin_watson = $ls->dw();
@predictedYs   = $ls->predicted();
@residuals     = $ls->residuals();
print "intercept: $intercept, slope:  $slope\n";
```

Table IX shows the results of the test. Even though the results are quite accurate, this module has limited practical use for Statisticians. More useful regression computations using matrix operations will be presented in Section 16.8 starting on page 62.

14.6. Random number generation

Introduction

An important computational tool in statistics is the Monte Carlo method. The Monte Carlo method is a controlled statistical experiment executed on a computer using algorithms that produce deterministic, repeating, sequences of computer numbers, referred to as *pseudorandom* numbers, that "appear" as random samples drawn from a known distribution, typically,

samples of independent and identically distributed $U(0,1)$ random variables. An algorithm that generates such sequences of pseudorandom numbers is commonly known as a *random number generator* (RNG).

In Perl, several modules that export RNG are available. Some modules are not readily useful to the statistician, as they provide random numbers for which the underlying distribution is not known. For, example, because of Perl's strong emphasis on network programming, there are modules for the generation of so called "truly" random numbers, which are employed chiefly in cryptographic applications.

We will survey the available Perl RNG modules that produce random sequences that appear to be independent and uniformly distributed over (0,1). Several techniques are available to generate variates from other distribution, using uniform random numbers as building blocks (see, e.g., Devroye 1986). We will summarize their main deterministic properties of the RNGs supplied by the modules, show how they can be used to generate random numbers within Perl scripts, test their probabilistic qualities by subjecting their output to a battery of test, and assess the efficiency through speed benchmarking.

There are several collection of tests for RNG in wide use today. A popular one is the **DIEHARD** battery of randomness tests Marsaglia (1996). This battery of tests provides a wide range of statistical tests for evaluating the stream of output of RNGs. The **DIEHARD** program, provided as an MSDOS executable or as C source code, is freely available from `http://stat.fsu.edu/~geo`.

Speed comparison will performed using the `Benchmark` module, as outlined in Section 13. This benchmarking is particularly critical for RNGs as they are designed to take full advantage of the computer architecture and the available fast basic operations. If not coded properly their execution might be considerably slowed down.

Survey of Perl RNG modules

Perl's built-in *rand()* function *Rand* uses the *rand()*[36] RNG from the standard C library which is a linear congruential RNG with period length 2^{31}.

```
print rand();
```

By default, when the seeds are set to values based on the system clock. The function to set the seeds, is *srand()*. For instance,

```
$seed = 4343434;
srand($seed);
print rand();    # prints 0.85723876953125
```

The `Math::Random` Module Math::Random is a Perl implementation of **RANDLIB**, the library of FORTRAN routines for random number generation, by Brown, Lovato, Russell, and Venier (1997). The uniform RNG implemented is derived from the package of generators described in L'Ecuyer and Côté (1991), based on the same combined linear congruential

[36] This is not always the case as it depends on the platform and compiler used. In more recent Perl versions, depending on the availability, one of *drand48()*, *random()*, and *rand()* (with decreasing order of priority) could be implemented.

generator, with a period length approximately equal to 2^{61}. A set of seeds, spaced 2^{50} values apart, give rise to 32 (virtual) generators. Each generator is further split into $2^{20} = 1,048,576$ blocks of numbers each of length $2^{30} = 1,073,741,824$. In the original implementation, each generator can be made to jump ahead, with a function call, to the start of the next block, or of its current block, or to the start of the first block. These feature can be used to code more efficient computer simulations. In the Perl module, the advanced facilities to split the generator into many virtual generators and to skip ahead in the sequence are not provided.

The simplest way to use the **RANDLIB** generator in Perl is by executing the following script

```
use Math::Random;
print random_uniform();    # prints U(0,1) number
```

The generator requires a two-dimensional seed. The first element of the seed vector should be an unsigned integer in the (1, 2147483562) range, the second in (1, 2147483398). By default, when the Perl processor loads package `Math::Random`, the seed is set to values based on the system clock. The function to set the seed is *random_set_seed()*. For instance,

```
use Math::Random;
$seed_1 = 1234567890;
$seed_2 = 1234567890;
@seed   = ( $seed_1, $seed_2 );
random_set_seed(@seed);
print random_uniform();
```

prints 0.222457440993228. The function *random_get_seed()* can be used to retrieve the current seeds. This function can also return an array of uniform random numbers when given an integer argument. For example,

```
@rnd_vec = random_uniform(100);
```

creates an array of uniform random numbers of length 100.

The `Math::Random::TT800` Module This Perl extension module implements a particular parametrization of Matsumoto's twisted generalized feedback shift register generator (TGFSR), called **TT800**, described in Matsumoto and Kurita (1992, 1994), with a period length of $2^{800} - 1 \approx 6.7 \cdot 10^{240}$. This generator is derived from the generalized feedback shift-register generator suggested by Lewis and Payne (1973). The module has an object-oriented interface. Once the generator has been created with a `new` statement, the method *next()* returns a double-precision floating point number between 0 and 1. The following script prints one uniform random number

```
use Math::Random::TT800;
$rand = new Math::Random::TT800;
print $rand->next();
```

The function *next_int()* returns a integer value filled with 32 random bits.

The **TT800** generator takes 25, not all zero, 32-bit integers as seed. If less than 25 integers are supplied, the rest are taken from the default seed. To set the seed we can use the fllowing script

```
use Math::Random::TT800;
@seed = (
    913860295,  1086204226, 1322218135, 2107674887, 1421458520, 2141267188,
    575078061,  1796978786, 476959775,  129791043,  2047223682, 1572134678,
    571817061,  629482166,  694818294,  1914617414, 2018740633, 234577687,
    1795180530, 1071903645, 821640312,  456869517,  829823942,  1469601523,
    2145855977
);
$rand = new Math::Random::TT800 @seed;
print $rand->next();    #  prints 0.616834293961719
```

The `Math::Random::MT` Module This module implements the *Mersenne Twister* RNG developed by Matsumoto and Nishimura (1998), a modification Matsumoto's TGFSR generator of Matsumoto and Kurita (1992, 1994). This generator has a huge Mersenne prime period length of $2^{19937} - 1 \approx 10^{6000}$ and its output is "twisted" to free it of long-term correlations when considered from a viewpoint of 623 dimensions. The Perl implementation passed all the **DIEHARD** statistical tests and, at the same time, proved to be faster than the RANLIB RNG.

To generate a number between 0 and 1 using a default seed of 0, we can use the following script

```
use Math::Random::MT;
$gen = Math::Random::MT->new();
print $gen->rand();     # prints 0.548813502304256
```

To set the seed and the range of the RNG, we can use the script

```
$seed = 42;                              # sets the seed
$gen  = Math::Random::MT->new($seed);    # creates generator
print $gen->rand( 2**32 );               # 1608637542
```

Note that in this case the generator returns a positive integer in the specified range.

Other RNG Modules There are other Perl modules that provide RNGs that could be of interest to Statisticians. For instance, the `Math::RandomOrg` module provides functions for retrieving thoroughly tested random numbers from the random.org server. The `Truly::Random` module provides an ability to generate truly random numbers from within Perl programs. The source of the randomness is from interrupt timing discrepancies. The `Math::Rand48` module provides an interface to the 48-bit family of random number functions, commonly provided on UNIX systems. Another potentially useful module is `PDL::RandVar::Sobol` which is a source of the increasingly popular *quasi* random numbers.

Statistical tests

The **DIEHARD** battery of tests[37] requires as input a specially formatted binary file of 10 to 11 megabytes size. The RNG should produce 32-bit integers that should be saved in a text file in hexadecimal form, 8 hex 'digits' per integer, 10 integers per line, and with no intervening spaces.

[37] The version available at the moment of writing was: DOS, Jan 7, 1997.

Consider continuing the the **TT800** example above, we can create a file satisfying the stated conditions with the following statements,

```
#  Matsumoto's TT800 pseudorandom number generator
use Math::Random::TT800;
open( OUT, ">TT800.hex" );
select(OUT);    # selects OUT as default output filehandle
for ( $i = 0 ; $i < 3e6 ; $i++ ) {
    printf "%08x", $rand->next_int();    # prints random 32-bit integer
    if ( $i % 10 == 9 ) { printf "\n" }
    ;                                    # starts a new line every 10 no.
}
```

which saves the output in a file named *TT800.hex*. The MSDOS auxiliary program, *asc2bin.exe*, provided in the test suit, can then be used to convert the hexadecimal file into a binary file, the can be directly fed to the main **DIEHARD** routines. The code used to prepare the hex input files for all generators is provided in Appendix B.

The following scripts runs the auxiliary `asc2bin` utility and the main `diehard` program providing the necessary input arguments. The output of the script is the file *rand.out* which contains the results of the 15 tests.

```
open( INPUT, "| asc2bin" );
select(INPUT);
print "\n";
print "\n";
print "\n";
print "TT800.hex\n";                    # supplies hex input file name
print "TT800.bin\n";                    # supplies binary output file name
close(INPUT);

open( INPUT, "| diehard" );
select(INPUT);
print "TT800.bin\n";                    # supplies binary input file name
print "TT800.out\n";                    # supplies test output file name
print "111111111111111\n";              # asks for all 15 tests
close(INPUT);
```

Table X presents the results of the 15 **DIEHARD** tests. Unless stated differently, a test is considered failed if all p-values are either zero or one. As expected Perl's internal RNG failed most of the tests. Considering also its relatively short period, we conclude that this generator is not suitable for serious or intensive use. The `Math::Random` module failed two tests, whereas the two long period TGFSR modules passed them all. These test results should be interpreted with extreme caution. The **DIEHARD** tests were designed with generators with a 2^{32} period in mind. The observed inverse relationship between period length and number of failed tests, suggests that further testing is required, before a more definite conclusion on the safety of a generator can be reached.

Table X: Marsaglia's **DIEHARD** tests for Perl modules

Test	**Perl's rand**	**RANDLIB**	**MT**	**TT800**
Birthday Spacings	fail	pass	pass	pass
Overlapping 5-Permutation	pass	fail[a]	pass	pass
Binary Rank (31×31)	fail	pass	pass	pass
Binary Rank (32×32)	fail	pass	pass	pass
Binary Rank (6×8)	fail[b]	pass	pass	pass
Bitstream (p-values)	fail	pass	pass	pass
OPSO	fail[b]	pass	pass	pass
OQSO	fail[b]	pass	pass	pass
DNA	fail[b]	pass	pass	pass
Count the 1's (stream of bytes)	fail	fail	pass	pass
Count the 1's (specific byte)	fail[b]	pass	pass	pass
Parking Lot	pass	pass	pass	pass
Minimum Distance	pass	pass	pass	pass
3-D Spheres	pass	pass	pass	pass
Squeeze	fail	pass	pass	pass
Overlapping Sums	pass	pass	pass	pass
Runs	pass	pass	pass	pass
Craps	pass	pass	pass	pass

[a]One of the two p-values of the test was zero.
[b]Most p-values are either one or zero.

RNG timings

We can compare the speed of the different RNG module implementations using the `Benchmark` module. There are two stages in the random generation process that can affect speed of execution: the setting up stage and the random number generation stage. We will time only the latter stage. The random number generation stage will be the more critical particularly when large sequences or random numbers are required. The following script has been used to time the generation of 3 million random numbers.

```
use Benchmark qw( timethese cmpthese);
use Math::Random;                          # begin set-up stage
use Math::Random::MT;
use Math::Random::TT800;
$RNG_MT    = Math::Random::MT->new();
$RNG_TT800 = new Math::Random::TT800;      # end set-up stage
$results   = timethese(
    3e6,
    {
        rand    => 'rand()',
        TT800   => '$RNG_TT800->next()',
        RANDLIB => 'random_uniform()',
        MT      => '$RNG_MT->rand()'
    }
);
cmpthese($results);
```

The timing results are presented in Table XI. The *cmpthese()* function produces a comparison

Table XI: CPU time to generate $3 \cdot 10^6$ pseudo-random numbers

CPU time	**Perl's rand**	**RANDLIB**	**TT800**	**MT**
seconds	0.67	10.80	6.17	19.92

chart. Note that this function will be made available to the script only imported explicitly. The results of the comparison comparison are

```
              Rate       MT RANDLIB   TT800    rand
MT        150587/s       --    -46%    -69%    -97%
RANDLIB   277829/s      84%      --    -43%    -94%
TT800     486066/s     223%     75%      --    -89%
rand     4464286/s    2865%   1507%    818%      --
```

The internal Perl *rand()* function turned out to be extremely fast, more than 8 times faster than the second fastest, **TT800**. MT turned out to be the slowest implementation, 3 times slower than **TT800**, two times slower than **RANDLIB**.

15. An overview of modules for numerical linear algebra

The ease with which Perl can be used for statistics depends critically on how closely the available data types match the Statistician's problem space. For example, matrices are commonly used in statistics to store the original and transformed dataset, to perform matrix computations, and to store the final results. Perl's object-oriented nature allows to create data structure suitable to store and manipulate matrices. Several of these types are made available through linear algebraic modules.

There are several modules that provide linear algebraic computational tools. For example, one of the most complete is the `Math::MatrixReal` module. This module implements the data type "real matrix," and a set of operators and methods to conveniently manipulate them. Besides basic matrix operations, the module provides methods to compute norms, inverses, determinants, condition numbers, vector lengths, various matrix decomposition methods, etc. The module provides also convenient input-output methods for matrices (including a method that writes matrices in LaTeX typesetting code).

This module has an object-oriented interface. An object-oriented language feature useful in using modules is the possibility to overload basic arithmetic operators. Perl allows to *overload* most of the existing operators with additional meanings depending on the context in which they are used. This feature is particularly convenient when using functions and methods to define and manipulate elementary mathematical structures such as matrices, results in an excessively burdensome and unintelligible syntax.

Consider using `Math::MatrixReal` for a simple matrix computation, such as $\boldsymbol{A} \cdot \boldsymbol{B}^T + \boldsymbol{C}$, after defining the three matrices involved using the following code.

```
use Math::MatrixReal;
$A =
  Math::MatrixReal->new_from_rows( [ [ 4, 2, 2 ], [ 3, 6, 1 ] ] )
  ;    # creates a 2x3 matrix
$B =
  Math::MatrixReal->new_from_rows( [ [ 1, 2, 4 ], [ 1, 0, 2 ] ] )
  ;    # creates a 2x3 matrix
$C =
  Math::MatrixReal->new_from_rows( [ [ 4, 2 ], [ 3, 7 ] ] )
  ;    # creates a 2x2 matrix
```

we can then use methods provided by the module to perform the computation. The following operations are required:

```
$BT  = new Math::MatrixReal( 3, 2 ); # creates matrix to store the transpose
$TOT = new Math::MatrixReal( 2, 2 ); # creates matrix to store final result
$BT->transpose($B);                  # transposition of matrix
$TOT->add( $A->multiply($BT), $C );  # multiplication & addition of matrices
```

It is clear that using the method-based interface is problematic, even in this simple calculation. Using the overloaded operators, the code simplifies to

```
$S = $A * ~$B + $C;
```

Note that by using the overloaded operators the creation of new matrices containing the results of the matrix operations is arranged automatically. The statement

```
print $S->as_latex( ( format => "%d", align => "l", name => "S" ) );
```

prints

```
$S = $ $
\left( \begin{array}{ll}
20&10 \\
22&12
\end{array} \right)
$
```

which, processed with LaTeX produces

$$S = \begin{pmatrix} 20 & 10 \\ 22 & 12 \end{pmatrix}$$

More limited linear algebraic tools are provided also by the `Math::Matrix` module and by the `Math::Cephes::Matrix` module, a Perl interface to the Cephes matrix routines. The `Math::MatrixSparse` module provides basic matrix operations for sparse matrices. Methods for solving linear systems iteratively, including Jacobi, Gauss-Seidel, and Symmetric Over-Relaxation are made available.

In the next Section we look at the PDL module, for efficient numerical computing. The problem with the above mentioned modules is their inefficiency, both in terms of speed and memory usage. As an illustration, consider comparing the `Math::MatrixReal` and `PDL` using the `Benchmark` module.

```
use Benchmark qw( timethese cmpthese );
use PDL;
use Math::MatrixReal;
$A = pdl [ [ 4, 2, 2 ], [ 3, 6, 1 ], [ 3, 6, 1 ] ];    # 3x3 matrix
$B =
  Math::MatrixReal->new_from_rows(
    [ [ 4, 2, 2 ], [ 3, 6, 1 ], [ 3, 6, 1 ] ] );    # creates a 3x3 matrix
$results = timethese(
    100000,
    {
        PDL              => '$A x $A',
        Math::MatrixReal => '$B->multiply($B)'
    }
);
cmpthese($results);
```

```
Benchmark: timing 100000 iterations of Math::MatrixReal, PDL...
Math::MatrixReal: 20 wallclock secs (19.20 usr +  0.00 sys = 19.20 CPU) @ 5207.25/s
  (n=100000)
PDL: 16 wallclock secs (16.48 usr +  0.00 sys = 16.48 CPU) @ 6066.49/s
  (n=100000)

           Rate Math::MatrixReal              PDL
Math::MatrixReal 5207/s               --             -14%
PDL              6066/s              17%               --
```

From these results we can conclude that the PDL matrix multiplication operation is 17 per cent faster than the Math::MatrixReal *multiply()* function.

16. The Perl Data Language extension module

The Perl Data Language[38] (PDL) module endows Perl with capabilities analogous to those of interactive systems for array manipulation such as MATLAB and IDL. PDL was founded by the astronomer Karl Glazebrook, and is an ongoing project that involves many Perl programmers.

The PDL module enables efficient numerical computing in Perl. PDL introduces a compactly stored multidimensional array data type that can be manipulated with fast low-level languages like C, FORTRAN, or Perl itself.

The PDL class provides the fundamental operations of numerical linear algebra. Various constructors can create multidimensional arrays from lists of numbers and other arrays. Several functions can be used to access elements and slices of arrays. User-friendly, overloaded, basic array operator, including array addition and multiplication, and element-by-element array operations, are made available to manipulate arrays, in a fashion analogous to the matrix algebra as introduced in textbooks. A print method for conveniently printing arrays, is also provided.

This module provides an interface to part of the SLATEC library[39] of routines to manipulate matrices, calculate FFT's, fit data using polynomials, and integrate using piecewise cubic Hermite interpolation.

PDL provides also a number of interfaces to powerful two and three dimensional graphics libraries such as OpenGL and PGPLOT. We will not make use of graphics in our examples as this would require the separate installation of graphics libraries. Moreover, these libraries are have some limitations and are more difficult to install in non-Unix environments.

16.1. PDL documentation and other resources

Information on the PDL module is available, as most Perl modules, at the Perldoc.com Web site located at http://www.perldoc.com/. PDL has also a dedicated official site located at http://pdl.perl.org/, where documentation, demos, frequently asked questions, mailing lists, and other useful information can be consulted. A brief introduction of PDL can be found in Chapter 7 of Orwant *et al.* (1999). As some PDL routines are not well documented, often the original FORTRAN or C code source documentation needs to be consulted. This is the case of the implemented SLATEC functions (see Footnote 39).

[38] At the moment of writing this paper, a Windows binary version of PDL-2.2.1 was available for use with ActivePerl 5.6. With ActivePerl, PDL can be installed using the Perl Package Manager (PPM). After downloading the *zip* file containing the binary distribution of PDL for Win32, and unzipping the PDL binary distribution for Win32, the file needs to be unzipped using folder names. From within the folder containing the PDD file *PDL.ppd*, PDL can be installed by typing `ppm install PDL`. *PPD* files are made available for ActivePerl users. These files are produced compiling open-source code available through CPAN using Microsoft Visual C++.

[39] The SLATEC Common Mathematical Library is a comprehensive software library written in FORTRAN 77, containing over 1400 general purpose mathematical and statistical routines (For more information, see http://www.netlib.org/slatec/).

16.2. PDL shell

PDL can be used as any other Perl module from a script by including the statement

```
use PDL;
```

at the top of the Perl script. PDL has also a shell interface for interactive data analysis. The PDL shell can be invoked from the operating system command line by typing `perldl`, as the following screen output illustrates.

```
C:\>perldl
perlDL shell v1.30
 PDL comes with ABSOLUTELY NO WARRANTY. For details, see the file
 'COPYING' in the PDL distribution. This is free software and you
 are welcome to redistribute it under certain conditions, see
 the same file for details.
ReadLines enabled
Reading PDL/default.pdl...
Found docs database I:/Perl/site/lib/PDL/pdldoc.db
Type 'help' for online help
Type 'demo' for online demos
Loaded PDL v2.2.1cvs (supports bad values)
perldl>
```

Here follow a few commands that demonstrate some of PDL's capabilities, in its interactive mode. Note that, in this mode, a statement need not and with a semicolon. Also, the `print` command can be shortened to `p`.

```
perldl> $y = grandom(100); $x = grandom(100);  # normally distributed random numbers
perldl> p histogram2d($y,$x,1,-4,8,1,-4,8)     # obtains a transition matrix

[
    [ 0  0  0  0  0  0  0  0]
    [ 0  0  1  0  0  0  0  0]
    [ 0  0  1  3  4  6  0  0]
    [ 0  1  4 10 15  5  0  0]
    [ 0  1  8 12 11  4  2  0]
    [ 0  1  4  2  3  1  0  0]
    [ 0  0  0  0  1  0  0  0]
    [ 0  0  0  0  0  0  0  0]
]
```

16.3. Basic PDL data type

The basic data type of PDL is a multidimensional array of numbers of identical type stored in logically contiguous memory locations, that can be manipulated directly with with fast low-level languages like C, FORTRAN, or Perl itself. In Perl, arrays can store any type of scalar. This flexibility is achieved at the expense of efficiency, by using pointers to scalars instead of scalars. PDL allows also to specify the range of types. For example PDL supports single and double precision floating point arithmetic, as well as several integer types, such as byte, short, and long.

As one of the most useful data structures for statistics are matrices, in which columns correspond to variables, and rows to observations of a dataset, in subsequent examples, we will

consider only one-and two dimensional arrays PDL objects, instead of more general multidimensional object.

16.4. Constructing PDL objects

PDL introduces a new data structure the "pdl numerical array," often referred to as a "piddle." A piddle is an object that can store efficiently a collection of numbers numbers for manipulation with normal mathematical expressions.

The statements below show how PDL objects can be constructed. PDL variables can be created in several ways.

```
use PDL;
$vec = pdl( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 );    # vector
$mat  = pdl [ [ 1, 2, 3, 4 ], [ 5, 6, 7, 8 ] ]; # 2x4 matrix
$mat2 = pdl $mat;                                # make a new copy
```

Several functions are available to construct special vectors and matrices. For example,

```
use PDL;
$0   = zeroes( 3, 3 );                          # null matrix of order 3
$one = ones( 2,   5 );                          # 5x2 matrix of ones
$I   = zeroes( 5, 5 );
$I->diagonal( 0, 1 ) .= 1;                      # identity matrix of order 5
$e = zeroes(9);
$e->slice('2') .= 1;                            # e_3
$i = ones(4);                                   # vector of ones
```

16.5. Indexing PDL objects

Indexing and slicing arrays in PDL requires particular care, as it is in many ways unconventional. A PDL vector data type is an ordered set of memory locations referenced by a name and an integer index, starting with zero. Negative indices work like the case of Perl arrays and return elements by counting backwards from the end. An element of a two-dimensional PDL object, is indexed by two indices. In PDL, contrary to what most Statisticians are used to, the first index represents the column and the second the row. This index order is common practice in image processing, the computationally intensive field of application for which PDL was originally developed.

Individual elements of a PDL object can be accessed through the *at* function, as the following example using `perldl` illustrates.

```
perldl> p $A = sequence(12,3);  # initializes 3x12 matrix $A

[
 [ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31 32 33 34 35]
]

perldl> p $A->at(8,1);  # prints element at nineth column and second row
```

```
20

perldl> p $A->at(-2,-1); # prints element at the next to last column and last row

34
```

Submatrices and subvectors of a PDL object can be extracted through the *slice* function. Consider the following examples.

```
perldl> p $col = $A->slice('6,:');  # 5th column assigned to PDL object $col

[
 [ 6]
 [18]
 [30]
]

perldl> p $sub = $A->slice('-2:-1,-2:-1');  # extracts 2x2 submatrix

[
 [22 23]
 [34 35]
]
```

16.6. Basic vector and matrix operations

In general, with matrices conformable for addition, mathematical operators such as `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `%` (module division), and `**` (exponentiation), in PDL refer to element-by-element vector and matrix operators. For example, the usual multiplication symbol `*`, a scalar multiplication, if a matrix is multiplied by a scalar, the Hadamard product of two matrices, if the matrices have identical dimensions, or a special matrix/vector multiplication.

```
use PDL;
$s = pdl 1 / 2;           # scalar
$A = sequence( 3, 2 );    # 2x3 matrix
$B = sequence( 7, 3 );    # 3x7 matrix
$s * $A;                  # scalar multiplication
$A * $A;                  # Hadamard product
$mu = matmult( $A, $B );  # matrix multiplication
$A x $B;          # matrix multiplication with overloaded operator 'x'
$A->transpose;    #transpose of a matrix
```

Relational operators such as `==` (equal) `!=` (not equal) `>` (greater than), and `>=` (greater equal than),

```
perldl> p $R = random(6,3)

[
 [ 0.16018677  0.50411987   0.9630127  0.69573975  0.92477417  0.18994141]
 [  0.3359375  0.17834473  0.99514771  0.45742798  0.99798584 0.097503662]
 [ 0.62515259 0.094390869  0.43771362  0.93148804 0.048431396  0.89459229]
]
```

```
perldl> p $R > .5

[
 [0 1 1 1 1 0]
 [0 0 1 0 1 0]
 [1 0 0 1 0 1]
]
```

To extract elements that satisfy particular conditions, we can use the *where* function, as the following example demonstrates.

```
perldl> p $R -> where($R>.7)

[ 0.9630127 0.92477417 0.99514771 0.99798584 0.93148804 0.89459229]
```

16.7. Manipulating PDL objects

The *dims* method returns the dimensions of the PDL object as a Perl list. The *nelem* method returns the number of elements in an array. As an example consider the matrix

```
perldl> p $C = sequence(5,5)

[
 [ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]
]
```

then

```
$n = nelem($C);   # assigns the number of elements in $C to the variable $n
$n = $C->nelem;   # same as above
$mean = sum($C) / nelem($C);    # computes average of the elements $C
$mean = avg $C;                 # same as above
( $ncols, $nrows ) =
  $C->dims;    # assigns the number of rows and columns to variables
```

Functions such as *maximum*, *minimum*, *average*, and *sumover* produce information on the rows in a matrix. These functions return vectors with the number of elements equal to the number of rows in the matrix. For example, *maximum* returns a vector whose elements are the maximum numbers in the corresponding rows of the matrix.

```
perldl> p average $C

[2 7 12 17 22]

perldl> p medover $C

[2 7 12 17 22]
```

```
perldl> p minimum $C

 [0 5 10 15 20]

perldl> p maximum $C

 [4 9 14 19 24]
```

The same operations can be applied to columns using the *xchg* function which can be used to exchange the two dimensions of the matrix.

```
perldl> p average $C->xchg(0,1)

 [10 11 12 13 14]

perldl> p medover $C->xchg(0,1)

 [10 11 12 13 14]

perldl> p minimum $C->xchg(0,1)

 [0 1 2 3 4]
```

Note that because of the (column,row) addressing order, one-dimensional PDL objects are treated as row vectors.

The analogous function *max*, *min*, *avg* and *sum*, perform the same operations only including all the elements of the matrix. It is possible to obtain the location of a maximum or a minimum with the functions *maximum_ind*, *minimum_ind*. For example,

```
perldl> p maximum_ind $C;

 [4 4 4 4 4]
```

16.8. Regression computations

One of the most common and important applications in statistic of linear algebric computation is in the fitting of linear models by least-squares. From numerical analysis we know that there are several methods for the solution of the linear least-squares problem. PDL As an example, we will use the test econometric model of Longley (1967), which makes use of a dataset consisting of observed economic variables to illustrate hoe PDL can be used to do statistical computations. Because of the near collinearity of the variables, this estimation problem is also considered a difficult test for regression routines. The Longley dataset is part of StRD by NIST. Here we show the first 6 (out of 17) records of the file named *longley.dat*:

```
Obs TOTEMP  X1      X2      X3      X4      X5      X6
1   60323   83      234289  2356    1590    107608  1947
2   61122   88.5    259426  2325    1456    108632  1948
3   60171   88.2    258054  3682    1616    109773  1949
4   61187   89.5    284599  3351    1650    110929  1950
5   63221   96.2    328975  2099    3099    112075  1951
```

The first line contains the variable names; the actual data follows. The following script shows how the dataset can be read and least square estimates of the Longley model obtained using PDL.

```
use PDL;
use PDL::Slatec;    # to access matinv function
( $TOTEMP, $X1, $X2, $X3, $X4, $X5, $X6 ) = rcols 'longley.dat', 1, 2, 3, 4, 5,
  6, 7, { LINES => '1:-1' };    # reads longley dataset
$X0 = ones(16);                                    # creates a vector of ones
$x = transpose cat $X0, $X1, $X2, $X3, $X4, $X5, $X6;  # concatenates rows
$y        = transpose $TOTEMP;    # obtains a column vector y
$xt       = $x->transpose;        # obtains column vectors x
$ssc      = $xt x $x;             # sums of sq. and cross-prod. matrix
$inv      = matinv($ssc);         # inverts the ssc matrix
$beta_hat = $inv x $xt x $y;      # computes beta estimates
```

The statement `print $beta_hat` produces:

```
[
    [  -3482258.7]
    [   15.061873]
    [-0.035819188]
    [  -2.0202298]
    [  -1.0332269]
    [-0.051104065]
    [   1829.1515]
]
```

The following block of code shows how the standard errors of the estimates can be obtained.

```
( $k, $n ) = $x->dims;    # extracts dimensions of $x
$df         = $n - $k;                      #  degrees of freedom
$y_hat      = $x x $beta_hat;               # fitted values
$u_hat      = $y - $y_hat;                  #  least squares residuals
$ssr        = transpose($u_hat) x $u_hat;   #  sums of squared residuals
$sigma_hat2 = $ssr / $df;                   #  error variance estimate
$covb       = $sigma_hat2 * $inv;           #  var-cov matrix estimate
$seb = sqrt( $covb->diagonal( 0, 1 ) );   # standard errors of the estimates
```

The statement `print $covb` produces:

```
[890420.39 84.914926 0.033491008 0.48839968 0.21427416 0.2260732 455.4785]
```

More accurate estimates can be obtained by solving the least squares problem by factoring the design matrix using the singular value decomposition (SVD).

```
use PDL::Math;    # to access SVD routine
( $u, $s, $v ) = svd($x);
$dw = zeroes( $k, $k );
$dw->diagonal( 0, 1 ) .= 1 / $s;    # creates diagonal matrix
$beta_hat = $v x $dw x ( transpose($u) x $y );
```

Table XII: Benchmark regression results for Longley's model

Variable	Estimate $(\boldsymbol{X}'\boldsymbol{X})^{-1}$	Estimate SVD	Certified value	LRE $(\boldsymbol{X}'\boldsymbol{X})^{-1}$	LRE SVD
TOTEMP	−3482258·65178986	−3482258·63459535	−3482258·63459582	8·306	12·869
X1	15·0618725479599	15·0618722713199	15·0618722713733	7·736	11·450
X2	−0·0358191882359904	−0·0358191792925778	−0·035819179292591	6·603	12·434
X3	−2·0202297665921	−2·02022980381764	−2·02022980381683	7·735	12·397
X4	−1·03322687441975	−1·03322686717374	−1·03322686717359	8·154	12·838
X5	−0·0511040653404145	−0·0511041056533019	−0·0511041056535807	6·103	11·263
X6	1829·15147282475	1829·15146461331	1829·15146461355	8·348	12·882

Table XII summarizes the benchmark regressions results.

16.9. Numerical optimization

Optimization is an important tool in statistical computing. Though, in most statistical applications solutions to optimization problems, such as linear least squares, can be expressed in matrix notation, and are routinely solved using linear algebraic computations as seen, for example, in Section 16.8 starting on page 62, in more general settings, numerical optimization methods are needed. Maximizing non-tractable likelihood functions, fitting non-linear models, and applying robust methods are just a few examples of statistical application where numerical optimization is an essential tool. The `PDL::Opt::Simplex` module provides the general purpose nonlinear programming simplex method for locating the minimum of a multi-parameter function. The following code illustrates how the module can be used in finding a minimum of the standard difficult test function of a minimization routine introduced by Rosenbrock (1960) given by the equation

$$f(x, y) = 100\,(x - y)^2 + (1 - x)^2.$$

starting from the point $(0, 0)$. The function has a global minimum at the point $(1, 1)$ where it attains a value of 0.

```
use PDL;
use PDL::Opt::Simplex;
$minsize = 1.e-6;
$maxiter = 100;

sub rosenbrock {
    ($xv) = @_;
    $x = $xv->slice("(0)");
    $y = $xv->slice("(1)");
    return 100 * ( $x - $y )**2 + ( 1 - $x )**2;
}

$init = pdl [ 0, 0 ];
$initsize = 2;
( $optimum, $ssize ) =
  simplex( $init, $initsize, $minsize, $maxiter, \&rosenbrock );

print "OPTIMUM = $optimum \n";
print "SSIZE   = $ssize\n";
```

The output returned is

```
OPTIMUM =
[
 [0.99999962 0.99999961]
]

SSIZE   = 9.90704080273217e-007
```

For statistical computing purposes the default information provided by the function *simplex* might not be enough. For instance, from a numerical analysis point of view, to assess whether convergence has occurred, the number of iterations performed by the optimization routine is required whereas, from a statistical point of view, the value of the likelihood function might be necessary for testing purposes. To obtain a more informative output that includes also the coordinates of the vertices of the simplex for each iteration with the associated function values and the iteration number, we can add to the above script the function *logs*, defined as

```
sub logs {
    print "Vertices: $_[0]\n";
    print "Values:   $_[1]\n";
    print "Distance: $_[2]\n";
    print
      "Iteration: $_[3]\n\n";   # needs slight change in the simplex routine
}
```

The function call needs to be modified by including the optional parameter `\&logs` as last input. Note also that to get the iteration count the file containing the source code of the module was slightly modified.[40] In our example, the last two steps taken by the simplex method are shown below.

```
Vertices:
[
 [ 1.0000003  1.0000003]
 [ 1.0000002  1.0000001]
 [  0.999999   0.999999]
]

Values: [1.61655e-013 4.2647055e-013 1.0071831e-012]
Distance: 1.86295117510571e-006
Iteration: 55


Vertices:
[
 [ 1.0000003  1.0000003]
 [ 1.0000002  1.0000001]
 [0.99999962 0.99999961]
]

Values: [1.61655e-013 4.2647055e-013 1.4865345e-013]
Distance: 9.90704080273217e-007
Iteration: 56
```

[40] The extra parameter `100-$maxiter` was added to the function call `&{$logsub}($simp,$vals,$ssize)` at line 189 of the source code file *simplex.pm*.

Other modules useful for optimization include `Math::Brent`, an implementation of Brent's method for one-dimensional minimization of a function without using derivatives, and `Math::LP` which provides an object oriented interface to defining and solving mixed linear-integer programs, using Berkelaar's *lp_solve* library as the underlying solver.

17. Embedding statistical software in Perl

Perl offers wide range of ways to "communicate" with other applications. We have seen, for instance, how files and pipes can be used for this purpose. These approaches, though powerful, are a limited form of "one-way" communication. More advanced form of communication are also supported by Perl. The main disadvantage is that their implementation is often platform dependent. Consider extending Perl with procedures from a more specialized statistical or numerical application. We would like, for instance, to use R's advanced statistical procedures and visualizations, in our Perl program, within a Win32 operating system.

DCOM, an acronym for *Distributed Component Object Model*, is a an object-based distributed system developed by Microsoft to build object-based applications. DCOM objects expose interfaces that allow other DCOM-enabled applications to access, remotely, their functionality. Perl can be DCOM-enabled using the `Win32::OLE` module. This module provides the means to control many Win32 applications, including DCOM-enabled statistical and numerical software, from within a Perl scripts.

An implementation of R as a Microsoft DCOM server is available from the Comprehensive R Archive Network (CRAN) (http://CRAN.R-project.org/). Consider the following Perl script that: 1) downloads the Hooker's dataset (see, e.g, Weisberg 1995) from STATLIB, 2) extracts and prepares the data, and finally 3) uses R to fit a regression model and plot the residuals.

```
# downloads from Web file containing data examples from Weisberg (1985)
use LWP::Simple;
$data = get('http://www.stat.unipg.it/stat/statlib/datasets/alr');

# extracts data of example on page 28 of book from file to Perl variables
@lines = split( /\n/, $data );
while ( $_ = shift @lines ) {
    if ( /alr28/ .. /alr29/ ) {
        if ( ( ( $temp, $pres ) = split( /\s+/, $_ ) ) == 2 ) {
            push( @temp, $temp );    # stores temperature
            push( @pres, $pres );    # stores pressure
        }

    }
}

# creates DCOM server object
use Win32::OLE;
$R =
  Win32::OLE->new('StatConnectorSrv.StatConnector')
  ;
$R->Init('R');
```

```
# runs regression in R
$R->EvaluateNoReturn('reg<-lm(pres~temp)');

# gets residuals from R into Perl array
$ref_to_residuals = $R->Evaluate('reg$res');

# prints comma separated residuals
print join( ',', @$ref_to_residuals );

# plots residuals in R
$R->EvaluateNoReturn(
    'plot( reg$res, pch=19, xlab="Observations", ylab="Residuals" )');
$R->EvaluateNoReturn('abline(h=0)');

# closes connection with R
$R->Close;
```

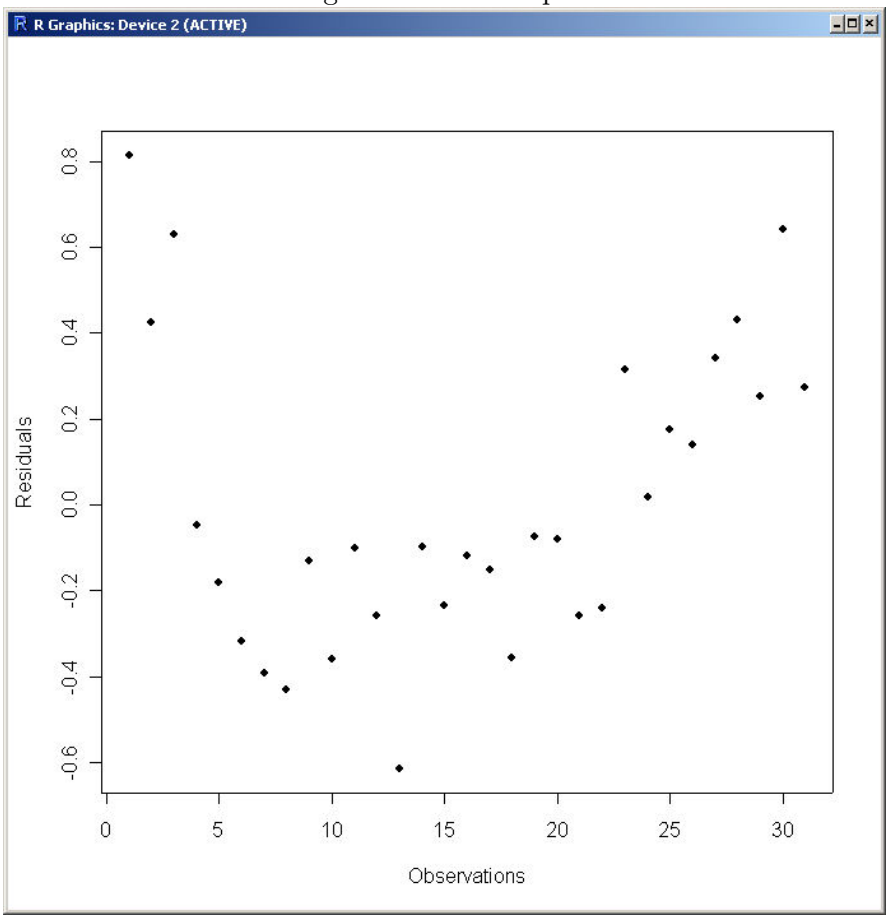
When executed, the script produces the following textual output

```
0.812316790922684,0.424485959040852,0.629993463395295,-0.0473430501095586,...
```

and creates a window (see Figure 4) containing a residual plot.

Information and examples on how to use S-PLUS are provided in Venables and Ripley (2000). Other approaches are available to connect Perl with other useful applications. For instance, the module `Inline::Octave` allows the matrix language Octave to be used from within Perl programs.

Figure 4: Residual plot



18. Conclusion

Perl is a free, general purpose programming language, that can be employed in the solution of a multiplicity of tasks faced regularly by statisticians. In Part I of the paper, we have shown, by means of practical examples, how Perl can assist the statistician in activities ranging from data collection and preparation to the typesetting and dissemination of the final results. Perl can automate what would otherwise be tedious, repetitive, and error prone activities. We also described how modules, that enhance Perl's functionality, can be employed effectively by statisticians to process data. In Part II we investigated how Perl can be used for statistical computations. We have demonstrated how Perl can be used to perform a variety computational tasks, from simple statistical analyses to more complex statistical computations, by extending it with modules and specialized applications. We have also extensively tested Perl's statistical and numerical modules and shown how they can be effectively used, in conjunction with Perl's semi-numerical and non-numerical functionality.

Throughout the paper, we have tried to emphasize the advantage of Perl, used as a scripting language, combined with its more advanced features, such as network programming support, object-orientation, and extensibility, in solving statistical problems. We have demonstrated how Perl can be used as a platform to make statistical computation projects more easily reproducible.

Acknowledgements

The author would like to thank Jan de Leeuw, Colin McClean, Jan Minx, and three anonymous referees for helpful comments and suggestions.

Appendices

A. Perl code for LRE routine

```
sub re {
    my ( $est, $cert ) = @_;
    return abs( $cert - $est ) / abs($cert);
}

sub log10 {
    my $n = shift;
    return log($n) / log(10);
}

sub lre {
    my ( $est, $cert, $nosd ) = @_;
    my $aest = abs(est);
    if ( $cert == 0 ) {
        if ( abs($est) > 1 ) {
            return 0;
        }
        else {
            ( -log10($aest) < $nosd ) ? return -log10($aest) : return $nosd;
        }
    }
    elsif ( $cert == $est ) {
        return $nosd;
    }
    elsif ( abs( $est / $cert ) > 2 || abs( $est / $cert ) < 1 / 2 ) {
        return 0;
    }
    else {
        ( -log10( re( $est, $cert ) ) < $nosd )
          ? return -log10( re( $est, $cert ) )
          : return $nosd;
    }
}
```

B. Perl code for DIEHARD test hex input files

```
# Generates hex input file for DIEHARD test of \proglang{Perl}'s rand()
$seed = 4343434;
srand($seed);
open( OUT, ">rand.hex" );
select(OUT);    # selects OUT as default output filehandle
for ( $i = 0 ; $i < 3e6 ; $i++ ) {
    printf "%08x", int( rand(4294967296) );    # prints random 32-bit integer
    if ( $i % 10 == 9 ) { printf "\n" };       # starts a new line every 10 no.
}

# Generates hex input file for DIEHARD test of the Math::Random module
```

```
use Math::Random;
$seed_1 = 1234567890;
$seed_2 = 1234567890;
@seed   = ( $seed_1, $seed_2 );
random_set_seed(@seed);
open( OUT, ">randlib.hex" );
select(OUT);    # selects OUT as default output filehandle
for ( $i = 0 ; $i < 3e6 ; $i++ ) {
    printf "%08x",
      int( random_uniform( 1, 0, 4294967297 ) );   # prints random 32-bit integer
    if ( $i % 10 == 9 ) { printf "\n" }
    ;    # starts a new line every 10 no.
}

# Generates hex input file for DIEHARD test of the Math::Random::TT800 module
use Math::Random::TT800;
open( OUT, ">TT800.hex" );
select(OUT);    # selects OUT as default output filehandle
@seed = (
    913860295,  1086204226, 1322218135, 2107674887, 1421458520, 2141267188,
    575078061,  1796978786, 476959775,  129791043,  2047223682, 1572134678,
    571817061,  629482166,  694818294,  1914617414, 2018740633, 234577687,
    1795180530, 1071903645, 821640312,  456869517,  829823942,  1469601523,
    2145855977
);
$rand = new Math::Random::TT800 @seed;
for ( $i = 0 ; $i < 3e6 ; $i++ ) {
    printf "%08x", $rand->next_int();    # prints random 32-bit integer
    if ( $i % 10 == 9 ) { printf "\n" }; # starts a new line every 10 no.
}

# Generates hex input file for DIEHARD test of the Math::Random::MT module
use Math::Random::MT;
$seed = 4321;                           # sets the seed
$gen  = Math::Random::MT->new($seed);   # creates generator
open( OUT, ">mt.hex" );
select(OUT);    # selects OUT as default output filehandle
for ( $i = 0 ; $i < 3e6 ; $i++ ) {
    printf "%08x", $gen->rand( 2**32 );    # prints random 32-bit integer
    if ( $i % 10 == 9 ) { printf "\n" };   # starts a new line every 10 no.
}
```

C. Output of speed benchmarking of random modules

```
Benchmark: timing 3000000 iterations of MT, RANDLIB, TT800, rand...
    MT: 20 wallclock secs (19.92 usr +  0.00 sys = 19.92 CPU) @ 150587.29/s
(n=3000000)
   RANDLIB: 11 wallclock secs (10.80 usr +  0.00 sys = 10.80 CPU) @ 277829.23/s
(n=3000000)
     TT800:  6 wallclock secs ( 6.17 usr +  0.00 sys =  6.17 CPU) @ 486066.10/s
(n=3000000)
      rand:  0 wallclock secs ( 0.67 usr +  0.00 sys =  0.67 CPU) @ 4464285.71/s
 (n=3000000)
```

D. Example of **Perl** code for reproducible results

```
# downloads and decompresses diehard programs
use LWP::Simple;
getstore( "http://stat.fsu.edu/pub/diehard/diehard.zip", "diehard.zip" );
system( unzip diehard );    #  requires unzip program

# generates hex test file using TT800 pseudorandom number generator
use Math::Random::TT800;
open( OUT, ">TT800.hex" );
select(OUT);                # selects OUT as default output filehandle
for ( $i = 0 ; $i < 3e6 ; $i++ ) {
    printf "%08x", $rand->next_int();     # prints random 32-bit integer
    if ( $i % 10 == 9 ) { printf "\n" };  # starts a new line every 10 no.
}

# converts hex file to binary file using utility program asc2bin
open( INPUT, "| asc2bin" );
select(INPUT);
print "\n";
print "\n";
print "\n";
print "TT800.hex\n";                    # supplies hex input file name
print "TT800.bin\n";                    # supplies binary output file name
close(INPUT);

# performs DIEHARD test
open( INPUT, "| diehard" );
select(INPUT);
print "TT800.bin\n";                    # supplies binary input file name
print "TT800.out\n";                    # supplies test output file name
print "111111111111111\n";              # ask for all 15 tests
close(INPUT);

# extracts p-values from output file
open( IN, "TT800.out" );
while ( $line = <IN> ) {
    if ( $line =~ m/p-?(?:values?)?[:=]?\s*(\.\d+)/ ) {
    push ( @p_vals, $1 );            # stores p-values
    }
}

# performs statistical analysis on p-values
use Statistics::Descriptive;
$stat = Statistics::Descriptive::Full->new();
$stat->add_data(@p_vals);               # sends data to object
$count = $stat->count();                     # gets number of p-values from object
%f     = $stat->frequency_distribution(10);    # returns a frequency table

# prints final result in LaTeX table format
open( TEX, ">table.tex" );
select(TEX);    # selects OUT as default output filehandle
print "Value & Total matches \\\\\n";
for ( sort { $a <=> $b } keys %f ) {
    $co = 100 * $f{$_} / $count;
    printf "%5.1f &   %11.0f \\\\\n", $_, $co;
}
```

References

Abramowitz M, Stegun I (1965). *Handbook of Mathematical Functions*. Dover, New York.

Anscombe F (1973). "Graphs in Statistical Analysis." *American Statistican*, **27**, 17–21.

Barron D (2000). *The World of Scripting Languages*. John Wiley & Sons, New York.

Belsey DA, Kuh E, Welsch RE (1980). *Regression Diagnostics*. John Wiley & Sons, New York.

Brown BW, Lovato J, Russell K, Venier J (1997). "**RANDLIB**: Library of FORTRAN Routines for Random Number Generation." Available at http://odin.mdacc.tmc.edu/anonftp/.

Buckheit JB, Donoho DL (1995). "Wavelab and Reproducible Research." In A Antoniadis, G Oppenheim (eds.), "Wavelets and Statistics," pp. 55–81. Springer-Verlag, Berlin, New York.

Burke SM (2002). *Perl and LWP: Fetching Web Pages, Parsing HTML, Writing Spiders and More*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. ISBN 0-596-00178-9.

Christiansen T, Torkington N (2003). *Perl Cookbook: Solutions and Examples for Perl Programmers*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition.

Devroye D (1986). *Non-Uniform Random Variate Generation*. Springer-Verlag, New York.

Feller W (1968). *An Introduction to Probability Theory, Vol. 1*. Wiley, New York, third edition.

Friedl JEF (2002). *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition.

Galton F (1880). "Visualised Numerals." *Nature*, **21**, 252–256.

Guelich S, Gundavaram S, Birznieks G (2000). *CGI Programming with Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition.

Hill TP (1996). "A Statistical Derivation of the Significant-Digit Law." *Statistical Science*, **10**, 354–363.

Knüsel L (1989). "Computergestützte Berechnung Statistischer Verteilungen." *Technical report*, Oldenburg, München-Wien, (an English version of the program is available from http://www.stat.uni-muenchen.de/~knuesel/elv).

Knuth DE (1984). *The TEXbook*. Addison-Wesley Publishing Company.

L'Ecuyer P, Côté S (1991). "Implementing a Random Number Package with Splitting Facilities." *ACM Transactions on Mathematical Software*, **17**(1), 98–111.

Lewis TG, Payne WH (1973). "Generalized Feedback Shift Register Pseudorandom Number Algorithm." *Journal of the ACM*, **20**(3), 456–468.

Longley JR (1967). "An Appraisal of Least Squares Programs for the Electronic Computer from the Point of View of the User." *Journal of the American Statistical Association*, **62**(319), 819–841.

Marsaglia G (1996). "**DIEHARD**: A Battery of Tests of Randomness." Available at http://stat.fsu.edu/pub/diehard/.

Matsumoto M, Kurita Y (1992). "Twisted GFSR Generators." *ACM Transactions on Modeling and Computer Simulations*, **2**, 179–194.

Matsumoto M, Kurita Y (1994). "Twisted GFSR Generators II." *ACM Transactions on Modeling and Computer Simulations*, **4**, 254–266.

Matsumoto M, Nishimura T (1998). "A 623-Dimensionally Equidistributed Uniform Pseudo-random Number Generator." *ACM Transactions on Modeling and Computer Simulations*, **8**(1), 3–30.

McCullough B (1998). "Assessing the Reliability of Statistical Software." *The American Statistician*, **52**, 358–366.

McCullough B (1999). "Assessing the Reliability of Statistical Software: Part II." *The American Statistician*, **53**, 149–159.

Orwant J, Hietaniemi J, Macdonald J (1999). *Mastering Algorithms with Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Prechelt L (2000). "An Empirical Comparison of Seven Programming Languages." *Computer*, **33**(10), 23–29.

Press WH, Teukolsky SA, Vetterling WT, Flannery BP (1995). *Numerical Recipes in C*. Cambridge University Press, Cambridge, second edition.

Rogers J, Filliben J, Gill L, Guthrie W, Lagergren E, Vangel M (1998). "Statistical Reference Datasets for Assessing the Numerical Accuracy of Statistical Software." *NIST 1396*, National Institute of Standards and Technology, Bethesda.

Rosenbrock H (1960). "An Automatic Method for Finding the Greatest or Least Value of a Function." *Computer Journal*, **3**, 175–184.

Schwartz RL, Phoenix T (2001). *Learning Perl: Making Easy Things Easy and Hard Things Possible*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, third edition.

Sebesta RW (1999). *A Little Book on Perl*. Prentics-Hall, Inc., New Jersey.

Stein L (2001). *Network Programming with Perl*. Addison-Wesley Publishing Company, New Jersey.

Venables W, Ripley B (2000). *S Programming*. Springer-Verlag, New York.

Wall L, Christiansen T, Orwant J (2000). *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, third edition.

Weisberg S (1995). *Applied Linear Regression*. John Wiley & Sons, New York, second edition.

Affiliation:

Baiocchi Giovanni
Department of Economics and Finance
University of Durham
Durham, DH1 3HY, United Kingdom
E-mail: giovanni.baiocchi@durham.ac.uk
URL: http://www.dur.ac.uk/dbs/

Journal of Statistical Software
May 2004, Volume 11, Issue 1.
http://www.jstatsoft.org/

Submitted: 2003-02-21
Accepted: 2004-05-16