Supplementary material: Handling Data Files With Perl
Chuck Connor, Laura Connor January 24, 2014

Introduction

Geoscientists are awash in data. It is now possible to collect and use vast amounts of data in a typical geologic or geophysics project. Data collection is becoming highly automated, locations are determined using global positioning systems (GPS), sensors collect data at high rates. Furthermore, huge datasets are available on-line. Many projects require the use of data gathered on-line, such as bathymetric data, digital elevation models, or seismic hazard catalogs. All this is a bit of a double-edged sword. The quality of the science is generally improved by the increase in resolution that accompanies high volumes of data. On the other hand, it can be quite time-consuming and laborious to plow through these large data sets. What problems are associated with large data sets? A few are:

- large data sets make point and click style approaches to data analysis intractable. It does no one any good to try to highlight 30 000 measurements in a single file.
- these data sets come in an infinite variety of file formats. Different machines, different organizations, and different applications all save data sets in their unique ways. It is crucial to be able to extract information from these various formats.
- typical problems involve performing simple, repetitive calculations involving data extracted from large data sets. For instance, it may be appropriate to calculate a mean and standard deviation for some fraction of data stored in a file, or add a correction to each value in the file.
- preprocessing is often necessary to put data sets into a form in which they can be visualized using programs like GMT.

Today, these issues are ubiquitous in the discipline - a situation that simply did not exist in the recent past. Geoscientists are not alone. Most scientists are now faced with the need to handle large amounts of data in an efficient manner.

Fortunately, computer scientists have some of the largest data volumes, and consequently they have developed tools for handling large data sets that geoscientists can exploit. These tools primarily exist on Unix operating systems. They include a number of high-level scripting languages designed to manipulate data. These scripting languages include: perl, awk, nawk, gawk, sed, python, and others. In this class we will concentrate on using Perl for simple data manipulation tasks. Keep in mind that: 1) other similar tools are available, 2) each of these tools, especially Perl, has a great deal of power for manipulating and managing data beyond what we will be using. Perl is particularly adept at manipulating databases on the worldwide web.

Writing, Editing, and Running Perl Scripts

The process of editing and running Perl scripts is summarized in Figure 1.

Perl Examples

Many of the Perl scripts we will write and use involve manipulating data stored in a file.

Example 1

This first example gets data from a file, reads in the data one line at a time, transforms the data (does a little math), and sends the transformed data to a new file:

```
open (IN, "depth.dat") || die "Can't open depth.dat: $!\n";

$fileout = "out.dat";

open (HANDOUT,">$fileout") || die "problems!: $!\n";

while (<IN>){ $line = $_;
   @parse = split(" ", $line);
   $signed = -3.28* $parse[1];
   print HANDOUT "$signed $parse[2]\n";

}

close IN;
close Handout;
```

The syntax for is reasonably simple. In Example 1, the first line opens the data file called 'depth.dat'. After this point in the script, this file is referred to as 'IN'. If the script cannot locate this file (e.g., it does not exist, it is located in another directory), the script prints a simple error message and quits. The next two lines of the script create and name to output file – where the results will go – 'out.dat'. If this file already exists, the code will erase it and put the new results in the file (that is what the redirection symbol, >, does). The output file, 'out.dat', is referred to as 'HANDOUT' within the script. The elegance of Perl is seen in the next few lines. The couplet of lines:

```
while (<IN>){ $line = $_;
.
.
```

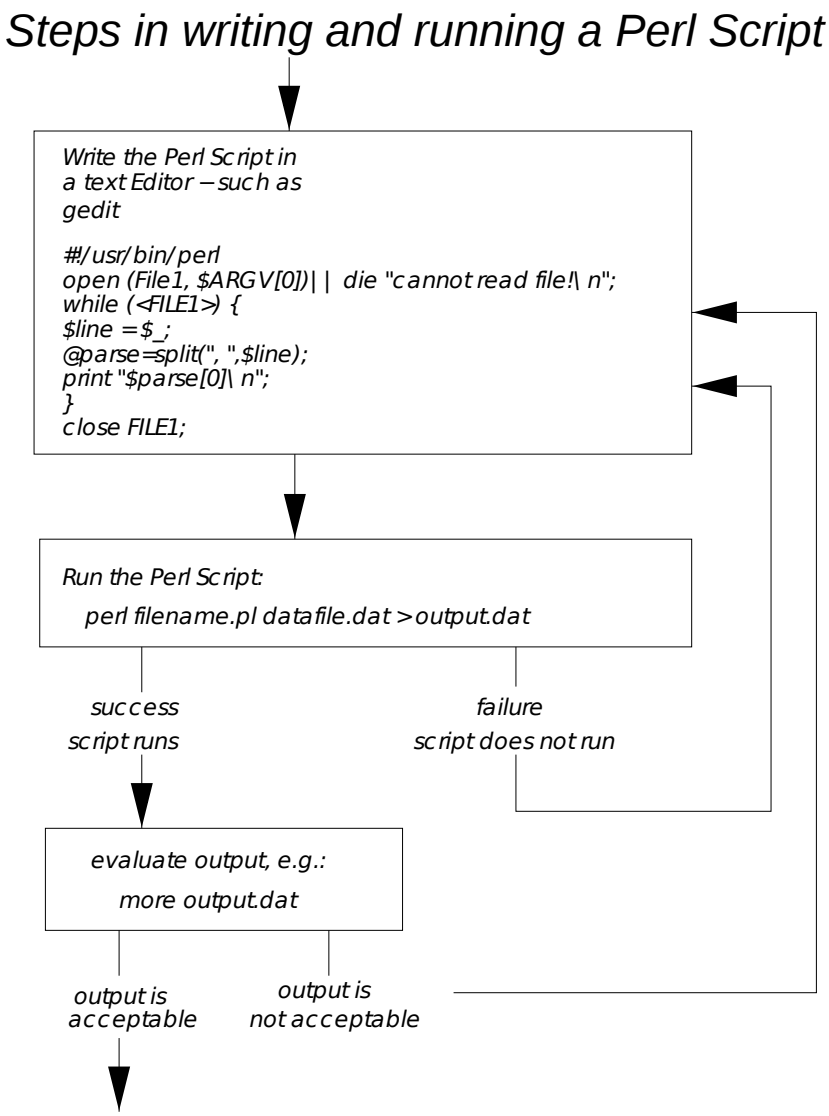

Figure 1: How to generate and run a Perl script

```
.
}
```

means:

- read through the entire file IN one line of data at a time
- each time a new line is read, $_, assign this entire line of data to the variable $line
- continue doing this until the entire file is read

This is an example of a loop. Every thing within the brackets is executed for each new line of data read from the file. Look at the lines in-between the brackets that are executed for each line of data. The first line is:

```
@parse = split(" ", $line);
```

This line parses the data line. It does this by creating an array of variables called 'parse'. Each element of the array parse contains information in $line, separated by white space. For example, if the input line looks like:

```
100100 200200 300300
```

then an array is created consisting of three elements. Each element in this new array can be treated as a variable. The three variables created in this case are:

```
$parse[0]=100100
$parse[1]=200200
$parse[2]=300300
```

Perl decides how to assign the variables values based on the white space in the input data line. Another example, the input looks like:

```
Rock1: 10.9 2.5 8.7
```

The array assignment looks like:

```
$parse[0]=Rock1:
$parse[1]=10.9
$parse[2]=2.5
$parse[3]=8.7
```

Suppose instead you have a data file separated by commas:

```
1,400234,3402202,43123.4
2,400235,3402201,43122.9
3,400236,3402199,43121.2
```

To read these values you would modify the line of the script to separate variables by commas rather than white space:

```
@data_array = split(",", $line);
```

The first time through the loop, 'data_array' is assigned the following values:

```
$data_array[0]=1
$data_array[1]=400234
$data_array[2]=3402202
$data_array[3]=43123.4
```

The second time through the loop the array data_array is assigned new values:

```
$data_array[0]=2
$data_array[1]=400235
$data_array[2]=3402201
$data_array[3]=43122.9
```

The third time through the loop parse....you get the idea. There is no magic in the rest of the script.

```
$signed = -3.28* $parse[1];
```

assigns a new variable a value based on one of the input lines, and the results are printed out:

```
print HANDOUT "$signed $parse[2]\n";
```

After the loop is completed (every line of the input file is read), the script artfully closes the two files.

Example 2

The following script has exactly the same structure. It handles the file names differently. It expects that the name of the input file will be provided as an argument on the command line. It prints the output to 'standard out', which is your computer screen, unless the output is redirected to a file.

```
open (FILE1, $ARGV[0]) || die "cannot read this file!\n";
while (<FILE1>) {
$line = $_;
@parse = split(" ", $line);
print "$parse[2]\n";
}
close (FILE1);
```

The execute this script, the following sort of line would be typed on the command line:

```
perl input.dat > output.dat
```

where: input.dat is the name of the file containing the input data (referred to as '$ARGV[0]' in the script. The results are directed to an output file, 'output.dat', using the redirection symbol, >. Note that the output file will only contain one column of input values – values that were in the third column of the original input file.

Example 3

Geophysicists are always rearranging columns in data files. For example, suppose you have a file of gravity values. The first column is the latitude, the second column longitude, the third column the gravity reading, the fourth column is the free air correction, and the last column is the simple Bouguer anomaly.

For example:

```
24.093 -87.003 23.22 20.78 20.99
```

You want to contour the simple Bouguer anomaly in GMT (note that GMT would prefer to have the longitude in the first column and the latitude in the second column). A perl script slightly modified from example 2 makes short work of this:

```
open (FILE1, $ARGV[0]) || die "cannot read this file! \n";

while (<FILE1>) {
$line = $_;
@parse = split(" ", $line);
print "$parse[1] $parse[0] $parse[4]\n";

}
close (FILE1);
```

One line of the output file will look like:

```
-87.003 24.093 20.99
```

Example 4

Suppose you would rather count the number of gravity readings in the file (say to make sure you have all of the data you collected in the actual file, and no more.

```
open (FILE1, $ARGV[0]) || die "cannot read this file!\n";
while (<FILE1>) {
$line = $_;
@parse = split(" ", $line);
$n=$n + 1;
```

```
}
print "number of reading = $n\n ";
close (FILE1);
```

An abbreviated way to write this summation:

```
open (FILE1, $ARGV[0]) || die "cannot read this file!\n";
while (<FILE1>) {
$line = $_;
@parse = split(" ", $line);
$n++;
}
print "number of reading = $n\n ";
close (FILE1);
```

where '$n++' is a shorthand way to increment the value of $n by one - this happens each time a new line of the file is read.

Example 5

Now take a file, extract one column of numbers, and sort the numbers into ascending order before they are printed out. Print the percentile at the same time. This script is used to make input files for plotting survivor functions in GMT.

```
open (FILE1, $ARGV[0]) || die "cannot read this file!\n";

#define an array that will be used to store the
#data while the script is executing
@accumulate=();

while (<FILE1>) {
$line = $_;
@parse = split(" ", $line); #parses on the white space
$accumulate[$n]=$parse[2];
$n++;
}

@prob = sort {$a <=> $b} @accumulate;
$decrement=$n;

foreach (@prob) {

$proportion = $decrement/$n;
$decrement--;
```

```
print "$_ $proportion\n";
}
close (FILE1);
```