Geological Computing - Homework 2
Flowcharts and Scripting

Connor

Due

This homework assignment is due in class on **Wednesday, January 25th**. This homework assignment will be started in class on Wednesday, January 18th.

Please note: A late assignment will not be accepted! Turn in a hard (paper) copy of your answers; do **not** email your answers. Answers should be typed; drawings may be hand-drawn. Please be legible and organized! Multiple pages should be stapled together. Some additional explanation is helpful and often necessary for the person reading your assignment to understand exactly what you are doing and why. Assume that the person grading your assignment does not have a copy of the assignment. What is important here is that *(1)* you understand what you are doing, and *(2)* that you can communicate what you are doing to others.

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 often highly automated. Locations are determined using a global positioning system (GPS) and sensors collect data at high rates. Huge datasets are also available from the internet. Many projects require the use of *on-line* data, such as bathymetric data, digital elevation models, or seismic hazard catalogs. This is all 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 the *point and click* style approach to data analysis intractable. It does no one any good to try to *highlight* 30,000 measurements in a single file.
- Data sets come in an infinite variety of file formats. Different machines, different organizations, and different applications often save data sets in unique ways. It is crucial to be able to extract information from these various formats.
- Problems arise 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 to add a correction to each data value in a file.
- Pre-processing is often necessary to transform data sets into a form that 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 science has developed tools for manipulting large data sets that geoscientists can exploit. These include a number of high-level scripting languages designed to manipulate data files. A small subset of scripting languages include: awk, nawk, gawk, sed, Python, Perl (Practical Data Extraction Language); there are many others. These scripting tools have a great deal of power for manipulating and managing data beyond what we will be using. Perl and Python are particularly adept at manipulating databases on the worldwide web (WWW) and processing data, line-by-line, from files. For this class you are encouraged to explore the use of Python and/or Perl. Each of these scripting languages is extensively documented with howto's and code examples via the WWW. Start your exploration by looking up Perl and Python in Wikipedia or just *google* Perl or Python.

Perl & Python Examples

Most Perl or Python scripts written in this class will involve the manipulation of a data file.

Example 1

This first example *opens* a data file, *reads* from the data file one line at a time, *splits* each line into separate data elements, *manipulates* one of the data elements (does a little math), and *prints* out the transformed and original data. Try this and see if you understand the output. The output will be printed to the computer screen. Notice the similarities and the differences between the Python and the Perl scripts. Each language has its own unique syntax.

Perl script: example1.pl

```
open (IN, "<$ARGV[0]") || die ("Cannot open $ARGV[0]: $!");
@MyData = <IN>;

foreach $line (@MyData) {
  ($data1, $data2) = split (" ", $line);
  $new_value = -3.28 * $data1;
  print ("$new_value $data2\n");
}
```

Python script: example1.py

```
import sys

with open (sys.argv[1], "r") as file:
  MyData = file.readlines()
  for line in MyData:
    line = line.strip()
    data1, data2 = line.split(" ")
    value = float(data1)
    new_value = -3.28 * value
    print (new_value, data2)
```

Let us review the code:

import Command line parameters or arguments, as they are often called, are identified by the *argv* variable. The Python script requires an additional code library (sys) to interpret command line arguments. The *import* directive loads this library into your python working space. Perl, on the other hand, identifies ARGV as a special variable to keep track of the command line arguments. In both cases this variable is actually a list, known as an array, with individual array elements accessed by a numerical index. In Perl, the first command line argument after the script name is indexed as 0, in Python this same argument is indexed as 1. It is possible to have multiple command line arguments; their names would be referenced with increasing index numbers following the appropriate syntax for each scripting language (sys.argv[2], $ARGV[3], etc).

open The first action, *open*, expects the name of the input file to be provided as an argument on the command line. Note that in Perl, the special variable *$ARGV[0]* refers to the name of this file; in Python this file is referenced by sys.argv[1]. The

```
|| die ("Cannot open $ARGV[0]: $!");
```

symbols indicate that the Perl script should stop and output a message to the terminal, if the expected data file could not be opened. After a successful *open*, Perl can now access the file by using its *file handle*, *IN*. The Perl code

```
@MyData = <IN>;
```

causes each line from the file to be read into the array, @MyData. Notice that in Perl an array variable begins with the symbol **@**; a regular variable begins with the symbol **$**. Python uses the statement *with* to handle the opening and closing of files; Python has a set of file *methods* that perform various file operations. The method *file.readline()* reads each line of data from a file and stores them in the array named *MyData*.

for loop In Perl, a loop structure begins with the keyword *foreach* and indicates that a series of identical steps will be performed on a series of variables, in this case the variables contained in the data array, *@MyData*. The series of actions performed by the *foreach* loop are contained within the symbols, **{}**. In Python the loop structure begins with the keywork *for*, the data array is called *MyData*, and the series of loop steps follow the symbol **:**. For each pass through the loop the variable, *$line* (*line* in Python), will hold one complete *line* of data from the data file and the code in the loop will operate on or transform this data.

split Each line of data must then be split into individual data elements that the script will act upon. Here, each line is split into 2 data elements, *$data1* and *$data2* (*data1* and *data2* in Python). In this case a *white space* or non-printing character separates each data element. This *white space* could be represented by spaces or tabs or a newline. In Python a newline is special and must be removed by the string method *line.strip()*. Try deleting this line from the python script and see what happens to the output when you run the code. Python is sensitive to indentation and new lines. Notice that each step in the for loop must be on one line and indented to the same tab-stop. Perl uses the semi-colon (;) to indicate the end of a code action. Your Perl script will fail to to run if you forget any semi-colons.

math step Now some math, each data1 element will be multiplied by -3.28 and stored in a new variable, *$new_value*.

print The last loop action is to print out a line of data, the transformed data, *$new_value* or *new_value* in Python), and the original data (*$data2* or *data2* in Python). Notice that in Perl the data to be printed is enclosed in quotes, " " and a new line symbol,

```
\n
```

appears at the end of the line. This symbol places the cursor at a new line *(**very important**, otherwise all the data will be printed on one line)*. In Python, each data value to be printed is just separated by a comma.

Two more steps remain, running the script and saving the printed output to a file. These steps happen together on the command line, typed as follows:

```
perl myscript.pl mydatafile.txt > myoutputfile.txt
```

The keyword, **perl**, invokes the Perl interpreter. The name of the Perl script is typed next, in this case, *myscript.pl*, followed by the name of the data file, *mydatafile.txt*. The following symbol, >, is called a redirection symbol. The action of this symbol causes output from the Perl script (*i.e.* the stuff after the *print* action, that is listed between the quotes) to be redirected and saved to a file. The name of the file, in this case, is *myoutputfile.txt*. To perform these steps using the Python script and the python interpretor type as follows:

```
python3 example1.py mydatafile.txt > myoutputfile.txt
```

Note: we are using the newer, *python3* interpretor, hence **python3** is the correct syntax for invoking this interpretor.

Let's look at a supposed line of input from our data file, *mydatafile.txt*:

```
5 4
20.1 40
54 78
3 0.7
99 305
```

The array assignment in PERL(or Python) looks like this. The first pass through the *foreach (or for)* loop gives

```
$data1 (or data1) = 5
$data2 (or data2) = 4
```

The second pass through the loop gives:

```
$data1 (or data1) = 20.1
$data2 (or data2) = 40
```

and so on for each line in the input date file. Suppose, instead, the values in the input data file are separated by commas, such as in a *.csv* file:

```
400234,3402202
400235,3402201
400236,340219
```

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

```
 ($data1, $data2) = split (",", $line);
```

or Python syntax:

```
 data1, data2 = line.split(",")
```

The split operation works exactly the same as with spaces, except that the line is split at the commas. The loop acts on each line of the input data file until the end of the file and then the script stops.

Example 2

Geophysicists are always rearranging columns in their 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.

Consider these 2 lines of data from a text file:

```
24.093 -87.003 23.22 20.78 20.99
24.165 -86.954 23.41 20.98 21.03
```

Suppose you want to contour the simple Bouguer anomaly in GMT (Generic Mapping Tools). Note that GMT would prefer to have the longitude, or x, in the first column and the latitude, or y in the second column.

Perl script: example2.pl

A Perl script that is slightly modified from Example 1 makes short work of this:

```
open (IN, "<$ARGV[0]") || die ("Cannot open $ARGV[0]: $!");
@MyData = <IN>;

foreach $line (@MyData) {
  ($latitude, $longitude, $grav, $free_air, $bouguer) = split " ", $line;
  print "$longitude $latitude $bouguer\n";
}
```

Python script: example2.py

Similarly, the Python script that would perform this task is as follows:

```
import sys

with open (sys.argv[1], "r") as file:
  MyData = file.readlines()
  for line in MyData:
    line = line.strip()
    latitude, longitude, obs_grav, free_air_grav, bouguer_grav = line.split(" ")
    print (longitude, latitude, bouguer_grav)
```

The output file contains three columns of data *(longitude latitude bouguer)*. One line of the output file would look this:

```
-87.003 24.093 20.99
```

Example 3

Suppose you need to count the number of gravity readings in a file.

Perl script: example3.pl

```
open (IN, "<$ARGV[0]") || die ("Cannot open $ARGV[0]: $!");
@MyData = <IN>;
$total = 0;
foreach $line (@MyData) {
  $total++;
}
print "Total number of readings = $total\n";
```

Python script: example3.py

```
import sys

total = 0
with open (sys.argv[1], "r") as file:
   MyData = file.readlines()
   for line in MyData:
        total += 1
   print ("total lines in file = ",total)
```

In Perl, ++, the increment operator, increases the value of the variable *$total*, by one, each pass through the loop, thus, counting each line in the file. The total number of lines counted is maintained by the variable *$total*. In Python, the variable *total* is incremented by += 1.

Writing, Editing, and Running Scripts

The process of wrinting, editing and running a script is summarized by the flowchart in Figure 1. First the script is created with a text editor. A Perl script is saved with a *.pl* file extension; a Python script is saved with a *.py* file extension. The file extension is a visual clue, even before the file is opened, indicating that the file is a particular type of script.

Scripts can be executed, or run, from the command line using the syntax shown previously. At this point, there are two possible outcomes: 1) the script executs without any errors, or 2) the script fails to execute and gives one or more error messages. If there were errors, lots of text usually gets printed out to the terminal and words like *error* or *warning* will appear. The best plan of attack, is to scroll back to the first error listed and note the line number the interpreter gives for the error location in the file. Sometimes warnings can be ignored, but errors need to be fixed. Fixing errors involves opening the script with a text editor and fixing the lines that are causing the errors. It is usually best to fix the first error listed and then try to run your script again. Often, multiple errors are all related to the first error listed, so just fix the first error and try running the script again. This sequence can take some time. It is often difficult to discover errors when first beginning to write programs. Persistence and practice is the key. All errors can be fixed.

Once the script executes cleanly, without any errors, the next step is to examine the output. Ask yourself, *Is this the output I expected?*. Remember, the script is under your direction and will only execute what you program. If the output is correct and what you intended, then you are finished. If the script gives unacceptable output, then the script needs to be edited and executed until the desired output is printed out. Good Luck!

Flowchart Toolbox

Design and create your flowcharts using the following suggested design objects. Using consistent design objects will make the flowchart easier to understand, easier to discuss, and easier to code. Each action, decision structure, loop structure, array, and variable can be specifically coded in Perl or Python. An example flowchart is shown on the last page in Figure (2).

Code Action Use boxes to indicate a code action. Specify the action with a verb, such as **Initialize**, **Read**, **Increment**, **Assign**, **Calculate**, **Input**, **Output**. If the code action includes a mathematical operation, then show the math. Specify the types of data *inputs* and data *outputs*.

Decision Structure A decision structure poses a mathematical question. Use diamonds to indicate decision structures. Use operators such as, *less than* (<), *greater than* (>), *equivalence* (==), *not equal* (! =), *AND*, *OR*, etc., to indicate the question. A decision structure defines two paths, a *true* path and a *false* path. Use **TRUE** and **FALSE** to identify the appropriate path. Each path should eventually lead back to the main line of the code.

Loop Structure Use long arrows to represent loop structures. A loop structure performs the same sequence of actions for different variables. Loops should include a decision structure that indicates when the loop should continue or stop.

Variables Within the code environment, data are accessible via variables. It is often helpful to define known variables and to initialize them. These variables can then be used to make the flowchart more accurate and readable.

Array An array is a special variable that indicates a sequential arrangement of values. A numerical series of: *0, 1, 2 . . . . . N* can be used to indicate an array. An array value is accessed via an *index*. Be sure to indicate the data being stored and the total number of elements in the array. The exact total may not be known, this value can be a variable, such as N in this example.

Problem 1

We return to the volume problem. Using a Perl or Python script, we want to find the volume of a prism, given its three dimensions (width, depth, height). Given that the three dimensions of the prism are $3002\,\mathrm{m} \times 2456\,\mathrm{m} \times 1901\,\mathrm{m}$, calculate the volume of the prism in cubic meters and in cubic kilometers. Use the following steps:

- **Develop** a flowchart for this problem, using the *flowchart toolbox* provided. Your flowchart needs to show: what are the inputs, what are the outputs, what are the procedures. Be as specific as possible and use the *tools* described in the *toolbox*!
- **Discuss** the syntax needed to create a script based on your flowchart design (as a group). The steps in your flowchart should suggest the layout of your script.
- **Create** a script using a text editor (*e.g.* gedit, on linux systems) and execute (*i.e.* run) it.
- **Turn in** the problem you are trying to solve, your flowchart for solving this problem, the script developed from your flowchart, the command line you used to run your script, and the output from your script. Your code should output a complete sentence or mathematical phrase. The output should clearly show the solution to the problem. *Units are important!* Don't forget to list them.

Problem 2

Suppose this volume estimate is for the amount of tephra (volcanic ash) erupted from a volcano. Given that the density of the tephra is $1000\,\mathrm{kg\,m^{-3}}$. What is the volume of magma (density = $2650\,\mathrm{kg\,m^{-3}}$) that erupted to form this tephra? What is the total mass of magma erupted? Use the same steps as before:

- **Develop** a flowchart for this problem, using the *flowchart toolbox* provided. Your flowchart needs to show the inputs and the outputs, what are the procedures. Be as specific as possible and use the *tools* described in the *toolbox*!
- **Discuss** the syntax needed to create a script based on your flowchart design (as a group). The steps in your flowchart should suggest the layout of your script.
- **Create** a script using a text editor (*e.g.* gedit, on linux systems) and execute (*i.e.* run) it.
- **Turn in** the problem you are trying to solve, your flowchart for solving this problem, the script developed from your flowchart, the command line you used to run your perl script, and the output from your script. Your code should output a complete sentence or mathematical phrase. The output should clearly show the solution to the problem. *Units are important!* Don't forget to list them.

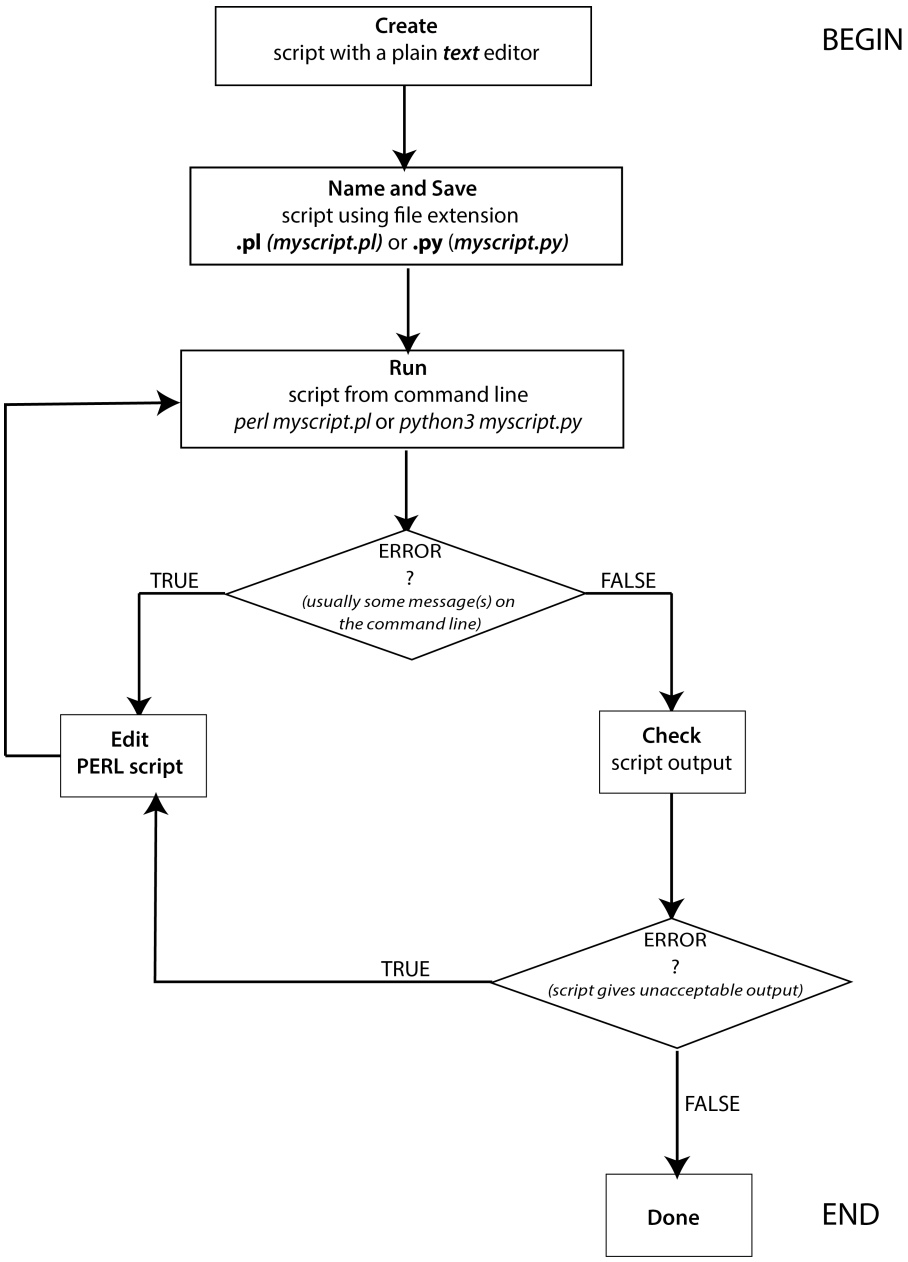

Figure 1: Flow chart for creating and running a Perl or Python script.

Problem 3

Next we return to the integration problem. Suppose there are ten prisms of equal width and depth, but these prisms vary in height. For each prism the width = depth = 90 m. The set, or array, of heights is 234.3, 233.5, 220.5, 200.5, 210.7, 222.3, 245.6, 254.5 270.6, 289.9 meters. What is the cumulative volume of the the prisms? If the heights correspond to elevation above sea level, and the rock is granite (density = $2670\,\mathrm{kg\,m^{-3}}$), what is the mass of rock above sea-level, represented by the prisms? Follow the same steps listed for Problem 1 and Problem 2.

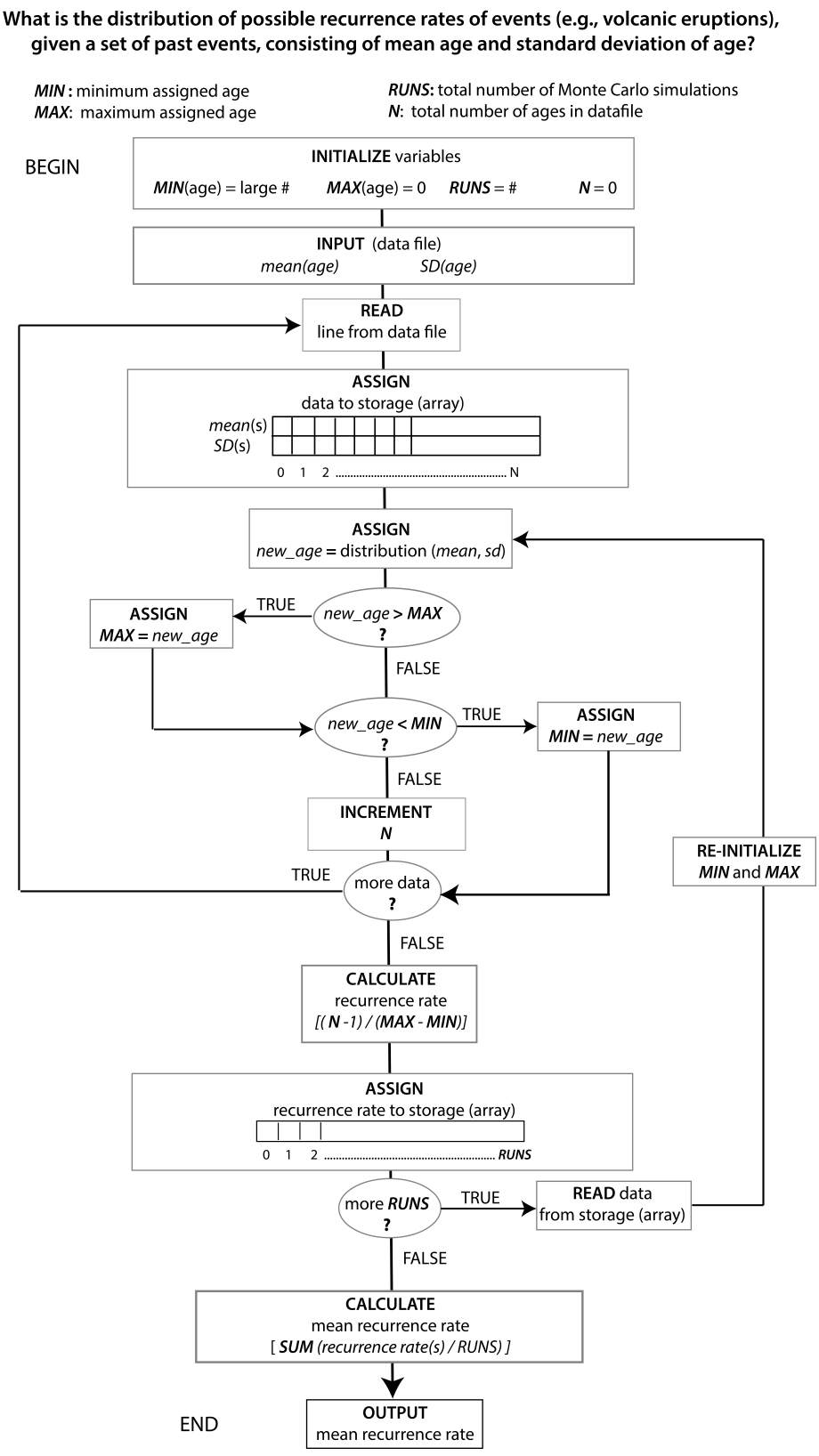

Figure 2: Example flowchart