Bash shell scripting tutorial

Scott T. Milner

September 9, 2020

1 Introduction

The shell is the program we interact with when we type at a Unix command line prompt. There are actually several different Unix shell programs; the most commonly used is `bash`. `bash` and other shells include facilities for writing programs, called "shell scripts". (Different shells have overlapping but distinct capabilities and syntax; in this tutorial, we assume the shell is `bash`.)

Shell scripts can be used to automate tasks involving Unix, as well as command-line programs that run under Unix. The shell scripting language reflects an accretion of good ideas and convenient tricks, in contrast to coherently designed, full-featured programming languages such as Python, which can also be used to write scripts for automating tasks. However, shell scripting is convenient because of its direct relationship to the well-conceived set of command-line tools that Unix comprises. All Unix commands can be used directly in shell scripts; indeed, many useful shell scripts consist of little more than a sequence of commands that could be typed at the command line.

Shell scripting benefits immensely from the Unix concept of small, well-defined command line utilities that can be combined to perform larger tasks by use of pipes and redirection of standard input and output. Familiarity with this approach is a prerequisite for writing good scripts. Likewise, powerful and versatile command line tools such as `grep, sed,` and `awk` greatly extend the range of what simple scripts can do.

The disadvantages of shell scripting for tasks beyond a certain complexity arise as well from its simple design. First among these limitations is the lack of convenient facilities for mathematical operations on real numbers, or data structures beyond integers, strings, and simple arrays. As a result, shell scripts are preferred for automation tasks, but not for "number crunching".

Of course, other shell scripting tutorials exist; many are rather brief and thus of limited use. One extensive tutorial is `http://tldp.org/LDP/abs/html/`, which is a useful reference but formidable to digest. This tutorial presents those features of shell scripting that commonly appear in scripts written for my research group.

The elements of the language are introduced here with brief but functional sample code. By themselves, these examples are too limited to give a clear indication of how real scripts are constructed.

However, similar tasks and consequently similar techniques occur again and again in our scripts. Therefore, in the final part of this tutorial we present several complete scripts from real applications, annotated to highlight techniques we have found useful.

2 Language elements

2.1 Variables

The traditional first program in any language prints out "Hello world". Here is a shell script that illustrates the use of variables:

```
#!/bin/bash

# to use a variable, just assign it
sayIt="Hello"
sayit=$sayIt" world"
echo $
```

In bash scripts, `#` denotes a comment, and blank lines are allowed. The first line above is a comment to Unix, that the script should be executed with `bash`.

In the script above, `sayIt` is first assigned the value "Hello". Then, `sayIt` is assigned its old value, concatenated with " world". The value of a variable is accessed with `$`.

Shell variables need not be "declared" or "defined" first; just start using them. Shell variables do not have "types" (integer, real, string, etc.). More precisely, all shell variables are strings, but can be interpreted and used as integers (discussed below).

2.2 Arguments

Values can be passed to shell scripts by command-line arguments. Here is an example (`script1.sh`):

```
#!/bin/bash
echo Hello $1
```

`script1.sh Scott` prints out "Hello Scott". Up to 9 command-line arguments are accessed with `$1` and so forth. `$#` is the number of arguments passed.

2.3 Strings

Shell variables hold strings, which are assigned with `=` (no space on either side):

```
myString=Hello
```

To assign a string containing spaces, enclose it in double quotes:

```
myString="Hello world"
```

To assign a string containing special characters that should not be "processed", enclose it in single quotes:

```
myString='echo $1'
```

Strings are concatenated by placing them together:

```
string1="Hello "
string2="world!"
bothStrings=$string1$string2
```

2.4 Catching output from commands

Shell scripts can contain any Unix command. Often, we want to save the output of the command in a variable. This is done with backquotes (`` `...` ``), as follows:

```
myFiles=`ls`
```

myFiles now contains a string with all the filenames separated by spaces.

An equivalent form is `$(...)`:

```
myFiles=$(ls)
```

As usual in Unix, the command inside `` `...` `` or `$(...)` can be a sequence of commands separated by semicolons, or a compound command involving pipes.

2.5 Integers

Shell variables are fundamentally strings; however, shell scripting includes syntax that allows us to treat variables as integers.

One way is to use `let`, which allows us to do math on variables that contain integers, using operations `+, -, *, /, **` (exponentiation), and parentheses:

```
z=4
let z=($z/2 + 4**2)*$z
```

which replaces `z` with 72. Note that division (`/`) here is *integer*, i.e., 5/2=2. Spaces are not permitted around the `=`, but can appear in the expression.

A more convenient way to do math on integer-valued variables is with `((...))`. Expressions inside double parentheses act as they would in C/C++, with no need to use `$` to obtain the value of variables. For example:

```
((e=4*e+3**2))
```

The right side evaluates without typing `$e`, and the assignment is made. Operators like `++` (increment) and `%` (mod) also work inside `((...))`.

2.6 Loops

Automation scripts typically involve looping. We may loop over values of some parameter, or a list of files to be processed. Or, we may continue a process while or until some condition is met.

Shell scripting includes three constructs for looping: `for, while,` and `until`. All have a similar syntax, with the body of the loop enclosed by `do` and `done`. `while` and `until` loops are controlled by a logical test (described below):

```
while <logical test>
do
<command1>
<command2>
...
done
```

`for` loops in shell scripts are different from those in C/C++; they iterate over a list of choices:

```
for fruit in apple pear orange
do
     echo $fruit
done
```

The list of choices can be contained in a variable, or the output of a command:

```
choiceList="apple pear orange"
for fruit in $choiceList
...
```

```
for file in ‘ls‘
...
```

To get a C-like `for` loop, use the shell command `seq` to generate a sequence to iterate over:

```
for i in ‘seq 1 10‘
...
```

The general form of `seq` is `seq <start> <step> <end>`, where `<start>` and so forth are integers (or expressions that evaluate to integers), and `<step>` is optional (with default 1).

2.7 Logical tests

To compare two variables, we use a logical test (note the spaces between the brackets, variables, and `==`):

```
[ $string1 == $string2 ]
```

To test for inequality, use `!=`.

If the variables contain integers (see below), comparison operators `-eq, -ne, -le, -ge, -lt, -gt` can be used.

A more convenient form of logical test for integer-valued variables is to use `((...))`. In the same way as arithmetic operations on integers (see above), expressions inside `((...))` using the usual comparison operators `<, >, <=, >=, != ==, &&` (and), and `||` (or) will evaluate to “true” or “false”.

Thus we can write logical tests in loops like

```
while ((x**2+y**2 < 100))
...
```

and in if statements (see below) like

```
if ((x < 2*y));
then
...
```

2.8 Control statements

In addition to looping, shell scripting provides control statements to direct the execution. The simplest is the if-statement:

```
if [ $string1 == $string2 ];
then
    echo "equal"
else
    echo "not equal"
fi
```

The logical test `[...]` takes the form described in the previous section, and must be followed by a semicolon. The `then` and `else` sections can contain multiple commands. The `else` clause may be omitted.

An if-statement with an "else-if" secondary test can be written as

```
if [ $string1 == $string2 ];
then
    echo "equal"
elif [ $string1 == "default" ];
then
    echo "default"
fi
```

In the above example, for simplicity the `else` cases have been omitted from the primary and secondary tests.

2.9 Random choices

The shell provides a facility for generating 16-bit random integers (ranging from 0 to 2^{15}-1 = 32767). Each time the shell variable `$RANDOM` is evaluated, a different pseudorandom number is obtained.

These can be used to make random choices among a relatively small number of alternatives, in conjunction with the mod (`%`) operator and integer arithmetic.

The mod operator computes the remainder of an integer divided by the modulus: `echo $((5 % 2))` gives 1, while `echo $((-5 % 2))` gives -1.

For example, to choose between two possibilities with equal probability,

```
if (($RANDOM % 2 == 0));
...
```

2.10 Case statements

Multiway branching can be performed with a `case` statement. The comparison is different from `if`; here, the argument is matched to a succession of "patterns", each of which is terminated with a closing parenthesis `)`:

```
case $percent in
[1-4]*)
     echo "less than 50 percent" ;;
[5-8]*)
     echo "less than 90 percent" ;;
*)
    echo "90 percent or greater" ;;
esac
```

Note the double semicolon terminating each command (which as usual can be multiple commands on multiple lines).

The patterns to be matched use the same syntax as in other Unix operations (commonly used in `vi, grep, ls, rm`, and the like). For completeness, patterns are described briefly in the next section.

2.11 Patterns

Patterns (or "regular expressions") are used by several Unix command-line programs and utilities. File management programs like `ls`, `cp`, `mv`, `rm`, and `tar` use patterns to select files to operate on.

The Unix utility `grep` selects lines from its input (stdin) that match a pattern, and prints those lines to stdout. The Unix visual editor `vi`, and the stream editor `sed` use patterns in its searching and substitution.

Patterns are strings, possibly containing special characters, which match part or all of other strings. A simple example is `ls *.sh`, which lists all files with names ending in `.sh` (i.e., all shell scripts, if we adhere to that naming convention). Because effective use of patterns extends the capabilities of Unix tools like `grep` and `sed` that frequently appear in scripts, we include here a table of the special characters used in patterns and their meanings.

Table 1: Special characters used in regular expressions and their meanings.

pattern	matches
`^`	beginning of string
`$`	end of string
`.`	any single character
`*`	zero or more repeats of previous character
`[...]`	any one of the enclosed characters
`[a-k]`	any one of the range of characters
`[2-5]`	any one of the range of digits
`[^...]`	any character not in ...
`\<...\>`	pattern ... only matches a word
`\(...\)`	saves pattern matched by ...
`\1, \2, ...`	the first, second, ... pattern saved

Pattern matching is useful in many contexts, but can be tricky — it is often a good idea to test patterns in a safe situation before relying on them where errors can do damage (as with `rm`).

2.12 Functions

In longer scripts it is often convenient to define functions, to perform a well-define subtask that is called upon multiple times.

Command-line arguments can be passed to functions in the same way as for scripts generally, with variables `$1`, `$2` and so forth. Variables declared with `local` are protected from meddling by the main program. Shell script functions do not return values, so functions must communicate with the calling program through global variables.

The syntax for defining a function is simple:

```
function newFcn {
...
}
```

in which ... are one or more commands.

2.13 Return status

Functions can return a value that indicates the status of the function call; by convention, 0 indicates success and nonzero values indicate some error. For a user-defined function, the return value is set with `return <value>`.

All built-in shell functions return such values. The return value of the most recently called function can be accessed with `$?`.

Return values of functions can be used as logical tests. For example:

```
while read nextLine
do
   echo $nextLine
done
```

reads successive lines and echoes them until the file is empty, at which point `read` returns a nonzero value, and the `while` loop ends.

Another way to use return status of functions and commands, is together with the operators `&&` (logical and) and `||` (logical or). The syntax

```
<cmd1> && <cmd2>
```

executes the second command only if the first one *succeeds* (i.e., returns status 0). This construct can be used to run a "check" (for the existence of a file) before executing a command.

Similarly,

```
<cmd1> || <cmd2>
```

executes the second command only if the first one *fails* (i.e., returns status 1). This construct can be used to run a "fallback" command if the first one fails.

2.14 Writing to files

Output to files can be written using Unix command line tools, including `echo`, `cat`, and `printf`. `echo` writes strings; `cat` concatenates one or more files; and `printf` prints one or more variables in a specified format, similar to print statements in C/C++.

For example,

```
echo "var1 equals" $var1
```

writes the concatenated string consisting of the text strings and variable.

`printf` writes one or more variables using a C-style format string, which provides a convenient way of controlling the output format:

```
printf "%5s %10.5f%12.4g  %5i\n" $s1 $f1 $f2 $i1
```

writes string `$s1` in 5 spaces, followed by floating-point variable `$f1` in 10 spaces with 5 decimal places, `$f2` in scientific notation in 12 spaces with 4 decimal places, and integer `$i1` in five spaces.

Finally, `cat` concatenates a list of files into a single file directed to stdout, as in:

```
cat file1 file2 file3
```

2.15 Output redirection

All three utilities `echo`, `cat`, and `print` write to stdout (standard output). However, this output can be redirected to write or append to a file with `>` or `>>`. If a script is invoked with redirection, as in

```
myScript.sh > myFile.out
```

all commands in the script that write to stdout are redirected to `myFile.out`. This is convenient if the script only produces one output file, since redirection can be used to save the file with any desired name.

Sometimes, a script writes more than one output file. Then, it is convenient to redirect output of commands within the script. This is done with the same `>` and `>>` syntax. For example,

```
cat file1 file2  > file3
```

writes `file3` (overwriting if it exists); whereas,

```
cat file1 file2 >> file3
```

appends to `file3` if it exists (and creates it otherwise).

2.16 "here" documents

Sometimes, it is convenient to store a small file within a script, for use with a utility like `cat` that requires a file as input. This can be accomplished with a "here" document:

```
cat << EOF
first line
second line
third line
EOF
```

In this example, `cat` will write the file (consisting of three lines) to stdout. Here documents can be combined with output redirection, by replacing the first line with

```
cat > myFile.out << EOF
```

Here documents can be used with any command that requires a file as input, as an alternative to maintaining a separate "database file" that is redirected to input, like this:

```
cat < database.in > myFile.out
```

2.17 Reading input

The Unix utility `read` is used to get input from files. By default, `read` takes input from stdin. To read a single line of text:

```
read newLine
```

To read multiple arguments (separated by spaces) from a single line of text:

```
read arg1 arg2 arg3
```

Often, we want to read all the lines in a file, processing each line in some way until the file is empty. This can be done with a `while` loop:

```
while read newLine
do
   echo $newLine
done
```

The `read` command returns status 0 ("true") if a line is read, and nonzero ("false") if the end of file is reached. This serves as a logical test to control the `while` loop. If input is read from stdin (the keyboard), ctrl-D acts as an end-of-file to terminate input.

2.18 Reading from a string

Sometimes it is convenient to read from a string; for example, we can use read to parse a string into arguments. This may be regarded as a form of input redirection, or a variation on a "here" document.

To read from a string:

```
read arg1 arg2 arg3 <<< $mystring
```

2.19 Input redirection

If a script is invoked with input redirection, as in

```
myScript.sh < myFile.in
```

all `read` commands in the script that have not been redirected then take input from `myFile.in`. This is convenient if the script only reads from one file, which we specify when we invoke the script.

Sometimes, a script reads from more than one file. Then, we can also invoke input redirection within the script. For example:

```
while read newLine
do
echo $newLine
done < myFile.txt
```

Note how the redirection follows the entire while loop construct; all reads inside the `while` loop come from `myFile.txt`.

However, if we want to read two successive lines from a file, the obvious code

```
read line1 < myFile.txt
read line2 < myFile.txt
```

does not work as intended; both reads take the *first* line from `myFile.txt`. To achieve the desired result requires a different approach, described in the next section.

2.20 Open a file

In languages like C/C++ or Python, facilities exist for "opening" a file, which provides a file "pointer" that advances with each successive read. When reading is done, the file must be "closed".

The following syntax accomplishes this in a shell script:

```
# open the file
exec 3< inFile
# read from the file (first line)
read 0<&3 arg1 arg2
# read again (second line)
read 0<&3 arg3 arg4
# close the file
exec 3<&-
```

Here 3 is a file "channel", where 0 is standard input (stdin), 1 is standard output (stdout), and 2 is standard error (stderr). If more than one file must be opened at once, channels 4, 5, and so forth can be used.

2.21 Arrays

Full-featured programming languages offer data structures more complex than a single variable. The simplest of these are arrays. The shell provides array variables, which are lists of simple variables that can be accessed by index and iterated over.

An array can be created by reading a list of arguments separated by spaces from a single line (the flag `-a` directs read to create an array):

```
read -a myArray
```

Arrays can also be assigned by listing the elements within parentheses:

```
myArray=("apple" "pear" "orange" $otherFruit)
```

Elements of the array can be accessed by index, as

```
firstElement=${myArray[0]}
```

(The array index starts at 0.) The length of the array is accessible with the same syntax as the length of a string:

```
${#myArray[@]}
```

The entire array can be accessed as a single string (with elements separated by spaces):

```
${myArray[@]}
```

Arrays can be iterated over with a `for` loop:

```
for i in ${myArray[@]}
do
...
done
```

2.22 Substrings

Often we need to take part of a string. Substrings can be accessed as follows:

```
myString="abcdefg"
newString=${myString:2:3}
```

newString then equals `bcd`, i.e., a substring starting at character 2, of length 3.

If the `2` above is omitted, the substring starts at 1; thus `${myString::3}` equals `abc`.

Likewise, if the `:3` is omitted, the substring goes to the end; thus `${myString:2}` equals `bcdefg`.

Characters can be removed from the end of a string with

```
${myString%<pattern>}
```

where `<pattern>` is a string possibly including "wildcard" characters. For example, `?` matches any single character, so that

```
${myString%???}
```

drops the last three characters from `myString`.

Finally, the length of a string is given by

```
${#myString}
```

2.23 Floating point math

Floating point math operations in shell scripting are limited and awkward, which constrains its use in programs that crunch numbers. However, many shell scripts require only a few floating-point calculations. For this purpose, the Unix utility `bc` (basic calculator) can be used.

`bc` implements a rather modest calculator, with arithmetic operations `+, -, *, /`. When invoked with option `-l`, a very few scientific functions are available (`s(x)` = sine, `c(x)` = cosine, `a(x)` = arctan, `l(x)` = natural log, `e(x)` = exponential, `sqrt(x)` = square root).

`bc` is normally an interactive program; launched from the command line, it responds to typed input. To use `bc` in a script, input can be supplied from `echo` via a pipe:

```
echo "c(3.14159/2)" | bc -l
```

The output can be saved in a variable using backquotes `‘...‘`:

```
myResult=‘echo "c(3.14159/2)" | bc -l‘
```

Variables can be included in the input expression:

```
echo "$var1*$var2" | bc -l
```

The number of output digits can be controlled with the command `scale=<n>`. This only works if the expression includes division, which can be worked around by dividing by 1 somewhere.

```
echo "scale=3;3.14*7/2+1" | bc -l
```

2.24 Automating interactive programs

Using `echo` to supply what would have been typed input to `bc` is an example of a general approach to make interactive programs "scriptable".

Another way to do this is with input redirection from a string, which we encountered with `read` in constructions like `read arg1 arg2 <<< $inputLine`.

The same syntax can be used with `bc`, and any other program or utility that expects typed input from stdin:

```
myResult=‘bc -l <<< "c(3.14159/2)"‘
```

2.25 Auxiliary programs

Unix provides a number of very powerful utility programs, chief among them `grep`, `sed`, and `awk`, which can be used to great advantage in shell scripts.

`grep` (Globally search for Regular Expression and Print) searches for patterns ("regular expressions") on each line of its input, and writes the lines matching the pattern to stdout. The basic format is

```
grep <pattern> <file>
```

in which `<pattern>` is any string, possibly containing one or more regular expressions (see "Patterns" above). If `<file>` is not specified, `grep` takes its input from stdin.

`awk` (named for its co-creators Aho, Weinberger, Kernihan) acts on each line of its input, splitting it into fields, doing simple calculations, and printing output.

Finally, `sed` (Stream EDitor) acts with editing commands on each line of its input, producing a stream of output lines. An important use of `sed` is to produce a sequence of files from a common "template file", in which only a few parameters are replaced by some iterated value. A typical use of `sed` looks like this:

```
sed "s/ANG1/$myAng1/; s/ANG2/$myAng2/" < infile > outfile
```

Here all occurrences in `infile` of the "template parameter" `ANG1` are replaced by the value of the variable `$myAng1`, and likewise for `ANG2` and `$myAng2`. The commands within quotes are standard `vi` substitution commands.

3 Example scripts

In this section, we present several scripts that automate various tasks arising in my research. Although the particular applications pertain to my group, the categories of tasks they represent and the scripting techniques they employ are relevant to any computational researcher.

The scripts fall roughly into three categories:

1. extracting data from output or log files of computational programs;
2. automating compute jobs, submitted to PBS or other queue systems;
3. constructing long and repetitive input files.

3.1 Extracting results from files

```
...
 SCF Done:  E(RB3LYP) =  -552.938806292     A.U. after   10 cycles
...
 Alpha  occ. eigenvalues --  -88.90122 -10.22991 -10.22991 -10.21007 -10.20983
 Alpha  occ. eigenvalues --   -7.97827  -5.94170  -5.93770  -5.93284  -0.87872
 Alpha  occ. eigenvalues --   -0.73873  -0.73631  -0.57462  -0.56304  -0.52033
 Alpha  occ. eigenvalues --   -0.43087  -0.40782  -0.38726  -0.37642  -0.34632
 Alpha  occ. eigenvalues --   -0.25864  -0.24890
 Alpha virt. eigenvalues --   -0.02966   0.00606   0.01137   0.03129   0.03259
 Alpha virt. eigenvalues --    0.03474   0.04011   0.06137   0.06196   0.06943
 Alpha virt. eigenvalues --    0.09110   0.09179   0.10201   0.10545   0.11620
 Alpha virt. eigenvalues --    0.14021   0.14641   0.15230   0.15333   0.16807
 Alpha virt. eigenvalues --    0.16835   0.20141   0.20394   0.21203   0.21387
 Alpha virt. eigenvalues --    0.25377   0.26555   0.27470   0.31453   0.36792
 Alpha virt. eigenvalues --    0.45975   0.49534   0.51152   0.53956   0.58259
 Alpha virt. eigenvalues --    0.61709   0.67887   0.69572   0.69799   0.70991
 Alpha virt. eigenvalues --    0.73149   0.74013   0.77902   0.87953   0.92346
 Alpha virt. eigenvalues --    0.97386   0.98760   1.00561   1.07732   1.08930
 Alpha virt. eigenvalues --    1.17629   1.19135   1.22912   1.32897   1.42423
...
```

Figure 1: Excerpt from Gaussian log file.

A common task for scripts is to extract values of interest from the log file of some computational program. Such log files are typically long text files, full of comments and information. Shell text processing commands are helpful in finding the relevant lines and extracting the desired fields from those lines.

Above is an example of log file output from the quantum chemistry program Gaussian, where the energy levels are printed sequentially from most to least strongly bound. The occupied and unoccupied orbitals are denoted `Alpha occ.` and `Alpha virt.` respectively. The HOMO (highest occupied molecular orbital) is thus the last `Alpha occ.` entry, and the LUMO (lowest unoccupied molecular orbital) is the first `Alpha virt.` entry. Separately, the ground state energy is reported on the line beginning `SCF Done:`.

The script below (`analyz.sh`) extracts the HOMO, LUMO, and ground state energy from a list of files read from stdin ((CHANGE THIS IN SCRIPT)), and computes the difference of these values from those in a reference calculation. (The purpose here is to analyze how the HOMO, LUMO, and ground state energy change with molecular distortion corresponding to different amplitudes of a normal mode. The set of Gaussian jobs with different imposed normal mode amplitudes were generated manually from a reference job in which normal modes were computed, by examining the normal mode, setting the amplitude manually, and saving a new compute job.)

The script uses command-line tools `grep`, `head`, and `tail` to extract the lines containing the ground state energy, HOMO, and LUMO ((FIX THIS — no need to use head and tail both on each line. And put SCF Done first.)). Briefly, `grep` searches stdin for all lines containing a given pattern. Usually, the desired line in a file can be uniquely specified by a well-chosen `grep` (as here for the ground state). But for the HOMO and LUMO, the obvious `grep` (of "Alpha occ" and "Alpha virt") finds multiple lines. So we use `head -n 1` and `Alpha occ.` to print the first and last lines of stdin, to obtain the relevant line in each case. The commands are strung together with pipes `|`), which directs the output (stdout) of one command into the input (stdin) of the next.

The relevant field is extracted and printed with `awk`, which is a command-line tool useful for this purpose. The same thing could be accomplished using `read -a` to fill an array `args` with the successive fields in the extracted line, and extracting the desired field by indexing the array, as indicated in the comments.

The difference between the extracted and reference values are computed using `bc`, and printed to stdout in a desired format with `printf`.

```
#!/bin/bash

# awk is convenient here to select fields; this could also be done as
# read -a args <<< `grep "Alpha occ" thiophene.log | tail -1`
# eh0=${args[5]}

eg0=`grep "SCF Done" thiophene.log | awk '{ print $5 }'`
eh0=`grep "Alpha  occ" thiophene.log | tail -n 1 | awk '{ print $6 }'`
el0=`grep "Alpha virt" thiophene.log | head -n 1 | awk '{ print $5 }'`
printf "%12s%12s%12s\n" "Delta E_H" "Delta E_L" "Delta E_g"
while read file
do
   eg=`grep "SCF Done" $file | awk '{ print $5 }'`
   eh=`grep "Alpha  occ" $file | tail -n 1 | awk '{ print $6 }'`
   el=`grep "Alpha virt" $file | head -n 1 | awk '{ print $5 }'`
   deh=`echo "$eh - $eh0" | bc`
   del=`echo "$el - $el0" | bc`
   deg=`echo "$eg - $eg0" | bc`
   printf "%12.5f%12.5f%12.5f\n" $deh $del $deg
done
```

Figure 2: `analyz.sh`, a script to extract results from a Gaussian log file.

The script is used with input redirection, with a list of files like `mode11.txt` below. (The names indicate the mode amplitude: "m4" equals minus 0.4Å, and so forth.)

```
mode11m4.log
mode11m3.log
mode11m2.log
mode11m1.log
thiophene.log
mode11p1.log
mode11p2.log
mode11p3.log
mode11p4.log
```

Figure 3: Input file list for `analyz.sh`.

3.2 Batch job

```
#PBS -A open
#PBS -l nodes=1:ppn=8
#PBS -l walltime=8:00:00
#PBS -l pmem=1gb
#PBS -m aeb
#PBS -M stm9@psu.edu
#PBS -j oe

# load group local copy of Gromacs
source /gpfs/group/stm9/default/SOFTWARES/gromacs-5.1.4STM/single/bin/GMXRC

# cd to working directory, define directories
cd $PBS_O_WORKDIR
MDP=$PBS_O_WORKDIR/MDP
SYSTEM=$PBS_O_WORKDIR/System

# energy minimization
mkdir EM
cd EM
gmx grompp -f $MDP/em.mdp -c $SYSTEM/system.gro -p $SYSTEM/topol.top -o em.tpr -maxwarn 20
gmx mdrun -nt 8 -deffnm em

# NVT equilibration
cd ../
mkdir NVT
cd NVT
gmx grompp -f $MDP/nvt.mdp -c ../EM/em.gro -p $SYSTEM/topol.top -o nvt.tpr -maxwarn 20
gmx mdrun -nt 8 -deffnm nvt

# NPT equilibration
cd ../
mkdir NPT
cd NPT
gmx grompp -f $MDP/npt.mdp -c ../NVT/nvt.gro -p $SYSTEM/topol.top -o npt.tpr -maxwarn 20
gmx mdrun -nt 8 -deffnm npt

# MD production run
cd ../
mkdir MD
cd MD
gmx grompp -f $MDP/md.mdp -c ../NPT/npt.gro -p $SYSTEM/topol.top -o md.tpr -maxwarn 20
gmx mdrun -nt 8 -deffnm md
```

Figure 4: `job.sh`, a typical Gromacs batch job script.

A very common use of shell scripts is for submitting batch jobs to PBS (Portable Batch System). Batch scripts are ordinary shell scripts, plus a set of comments at the beginning that contain directives to PBS. These directives request resources (nodes, cores, memory, and time), direct the job to a particular queue or allocation, request that mail be sent when the job starts or ends, and so forth. For details on PBS directives, see `https://albertsk.files.wordpress.com/2011/12/pbs.pdf`.

The script below (`job.sh`) is a typical Gromacs batch job, with no control statements or other automation features. It consists essentially of a set of commands that could have been entered sequentially at the command line.

The only difference is the line `cd $PBS_O_WORKDIR`, which navigates to the folder from which the job was launched. The shell variable `$PBS_O_WORKDIR` (Original WORKing DIRectory) is the full

path to the directory from which the command `qsub job.sh` was executed. For convenience, variables `$MDP` and `$SYSTEM` are defined as paths to subfolders of `$PBS_O_WORKDIR`.

3.3 Morphing simulation scripts

```
#!/bin/bash
#PBS -A open
#PBS -l nodes=1:ppn=8
#PBS -l walltime=24:00:00
#PBS -l pmem=1gb
#PBS -j oe

source /gpfs/group/stm9/default/SOFTWARES/gromacs-5.1.4STM/single/bin/GMXRC

echo $PBS_O_WORKDIR
cd $PBS_O_WORKDIR
MDP=$PBS_O_WORKDIR/MDP
SYSTEM=$PBS_O_WORKDIR/System

TEMP=/gpfs/scratch/stm9/vanish
mkdir $TEMP

INIT=$SYSTEM/equil.gro

for LAMBDA in `seq 0 20`
do
    mkdir Lambda_$LAMBDA
    cd Lambda_$LAMBDA

    cp ../Tables/table_${LAMBDA}.xvg md_${LAMBDA}_A_B.xvg
    cp ../Tables/table_0.xvg md_${LAMBDA}_A_A.xvg
    cp ../Tables/table_0.xvg md_${LAMBDA}_B_B.xvg
    cp ../Tables/table_0.xvg md_${LAMBDA}.xvg
    gmx grompp -f $MDP/md.mdp -c $INIT -p $SYSTEM/system.top -n $SYSTEM/system.ndx -o
md_$LAMBDA.tpr -maxwarn 20
    gmx mdrun -nt 8 -deffnm md_$LAMBDA
    mv md_$LAMBDA.trr $TEMP

    cd ..
    INIT=../Lambda_$LAMBDA/md_$LAMBDA.gro
done
```

Figure 5: `morphAll.sh`, a typical "morphing" simulation batch script.

The above script (`morphAll.sh`) is a typical batch script with automation. Here a sequence of `gromacs` simulation jobs are prepared and executed, each placed in its own folder `Lambda_1`, `Lambda_2` and so forth.

(In this "morphing" simulation, the interactions between groups A and B are progressively weakened. This is achieved by using a series of tabulated potentials `table_1.xvg` (original), `table_1.xvg`, `table_2.xvg` and so forth, which are copied into each job as needed.)

The initial configuration for the ith job is obtained from the final configuration of the previous job, the path to which is stored in the variable `$INIT`.

The second part of the "morphing" simulation uses a similar automated workflow, in which previously saved trajectory files `md_<k>.trr` are "rerun" with slightly different interactions, to produce energy data files `rerun_<k-1>.edr` and `rerun_<k+1>.edr`.

```
#PBS -A open
#PBS -l nodes=1:ppn=8
#PBS -l walltime=48:00:00
#PBS -l pmem=1gb
#PBS -j oe

source /gpfs/group/stm9/default/SOFTWARES/gromacs-5.1.4STM/single/bin/GMXRC

cd $PBS_O_WORKDIR
SYSTEM=$PBS_O_WORKDIR/System
MDP=$PBS_O_WORKDIR/MDP
TEMP=/gpfs/scratch/stm9/vanish

for LAMBDA in `seq 1 20`
do
    let LP=$LAMBDA+1
    let LM=$LAMBDA-1

    cd Lambda_$LAMBDA

    cp ../Tables/table_${LP}.xvg rerun_${LP}_A_B.xvg
    cp ../Tables/table_0.xvg rerun_${LP}_A_A.xvg
    cp ../Tables/table_0.xvg rerun_${LP}_B_B.xvg
    cp ../Tables/table_0.xvg rerun_${LP}.xvg
    gmx grompp -f $MDP/md.mdp -c md_$LAMBDA.gro -p $SYSTEM/system.top -n $SYSTEM/system.ndx
-o rerun_$LP.tpr -maxwarn 20
    gmx mdrun -nt 8 -deffnm rerun_$LP -rerun $TEMP/md_$LAMBDA.trr
    rm rerun_$LP.trr

    cp ../Tables/table_${LM}.xvg rerun_${LM}_A_B.xvg
    cp ../Tables/table_0.xvg rerun_${LM}_A_A.xvg
    cp ../Tables/table_0.xvg rerun_${LM}_B_B.xvg
    cp ../Tables/table_0.xvg rerun_${LM}.xvg
    gmx grompp -f $MDP/md.mdp -c md_$LAMBDA.gro -p $SYSTEM/system.top -n $SYSTEM/system.ndx
-o rerun_$LM.tpr -maxwarn 20
    gmx mdrun -nt 8 -deffnm rerun_$LM -rerun $TEMP/md_$LAMBDA.trr
    rm rerun_$LM.trr

    cd ..
done
```

Figure 6: `remorphAll.sh`, a "morphing" simulation batch script for rerunning the trajectory files.

The `.edr` files produced by the script `remorphAll.sh` are analyzed in turn by a third script `getDhDl.sh`, which uses `gmx energy` to extract time series for the potential energy, computes the difference between the two time series, and uses these data to evaluate the average $\langle \Delta H/\Delta\lambda \rangle$ and its uncertainty.

`getDhDl.sh` takes a single argument, the value of `$LAMBDA` to be processed. A small driver script (not shown) loops over all the desired `$LAMBDA` values. Separating this looping leads to a simpler and more versatile main script.

In the script, note the use of `echo` to automate the interactive utility `gmx energy`. `echo '8'` supplies the input that would otherwise be typed in.

```
#!/bin/bash
# argument is LAMBDA

# adjacent lambda values
LAMBDA=$1
LP=$((LAMBDA+1))
LM=$((LAMBDA-1))

# go to directory
cd Lambda_$LAMBDA

# run gmx energy on LP and LM output
# field 8 is potential energy (rerun has no KE)
echo '8' | gmx energy -f rerun_${LP}.edr -o energy_${LP}.xvg &> energy_${LP}.out
echo '8' | gmx energy -f rerun_${LM}.edr -o energy_${LM}.xvg &> energy_${LM}.out

# read comments atop .xvg files
exec 3< energy_${LP}.xvg
exec 4< energy_${LM}.xvg
for n in `seq 1 12`
do
    read 0<&3 line1
    read 0<&4 line2
done

# read the xvg header, echo to diff.xvg
for n in `seq 1 11`
do
    read 0<&3 line1
    read 0<&4 line2
    echo $line1 > diff.xvg
done

# read 100 data lines, compute avg and variance
# write difference to diff.xvg
avg=0.
var=0.
for n in `seq 1 100`
do
    read 0<&3 t e1
    read 0<&4 t e2
    diff=`echo "$e1 - $e2" | bc -l`
    printf "%15.5f%15.5f\n" $t $diff > diff.xvg
    avg=`echo "$avg + $diff" | bc -l`
    var=`echo "$var + $diff*$diff" | bc -l`
done
# normalize
avg=`echo "$avg/100." | bc -l`
var=`echo "$var/100." | bc -l`

# write avg and error estimate to stdout
# /0.1 because dH/d lambda
dhdlam=`echo "$avg/0.1" | bc -l`
err=`echo "sqrt($var/100.)/0.1" | bc -l`
printf "%15.5f%15.5f" $dhdlam $err

# close files
exec 3<&-
exec 4<&-
```

Figure 7: `getDhDl.sh`, a script for extracting and analyzing the morphing results.

The other noteworthy feature of this script is the use of `exec 3<` and so forth to open multiple files for reading, The file is then read with `read 0<&3`; successive reads from the same "file handle" read successive lines in the file. The file is finally closed with `exec 3<&-`.

3.4 Index file construction

```
#!/bin/bash --login

chains=32       #Total chains in the box
monomers=20     #Residues in a given chains
backbone="aC1|aC2"  #List all atoms that are part of the monomer backbone separated by |
and prefix with a

nGroup=5        # index on group created (groups 0-4 exist a priori)
chain=1         # index on chain
for chain in `seq 1 $chains`
do
   r1=$((monomers*(chain-1)+1))
   r2=$((monomers*chain))

   echo $backbone"&r$r1-$r2" >> make_ndx_cmds
   echo "name ${nGroup} chain${chain}" >> make_ndx_cmds
   nGroup=$((nGroup+1))

   echo $backbone"&!r$r1-$r2" >> make_ndx_cmds
   echo "name ${nGroup} notchain${chain}" >> make_ndx_cmds
   nGroup=$((nGroup+1))
done
echo "q" >> make_ndx_cmds

cat make_ndx_cmds | gmx make_ndx -f newBox.pdb -o PS.ndx
rm "#"*"#"
```

Figure 8: `packing_ndx.sh`, a script to drive the interactive utility `gmx make_ndx` to produce an index file of backbone carbons for a polymer melt.

The above script (`packing_ndx.sh`) is a more elaborate example of automating an interactive utility, in this case `gmx make_ndx`, to produce an index file that identifies the backbone carbons on all the chains in a polymer melt. (The system here consists of 32 chains of 20 monomers each, which values are "hard coded" into the script, but could have been input as command-line arguments.)

`gmx make_ndx` ordinarily responds to a sequence of typed queries and commands to create groups of atoms satisfying certain criteria, rename those groups as desired, and ultimately write an index file. Here we create a group for each chain, consisting of those atoms of type `C1` or `C2` (i.e., the main-chain carbons) with atom numbers within range for that chain.

The automation approach here is to assemble a file `make_ndx_cmds` that contains all the input we would ordinarily type. When the file is complete, we `cat` the file piped to `gmx make_ndx`.

3.5 Building a polymer `.pdb` file

```
1  #!/bin/bash
2  # reads monomer list from stdin
3  # arguments $1 $2 $3 are box dimensions
4  read monomer
5  gmx editconf -f $monomer.pdb -box $1 $2 $3 -noc -o newChain.pdb &> insert.log
6  read a b dx dy dz < $monomer.pdb
7  zOff=`echo "scale=4;$dz/10" | bc -l`
8  echo "0. 0. $zOff" > pos.dat
9
10 # loop over monomers
11 while read monomer
12 do
13         mv newChain.pdb oldChain.pdb
14         gmx insert-molecules -f oldChain.pdb -ci $monomer.pdb -ip pos.dat \
15            -rot none -scale 0. -o newChain.pdb &> insert.log
16         read a b dx dy dz < $monomer.pdb
17         zOff=`echo "scale=4;$zOff + $dz/10" | bc -l`
18         echo "0. 0. $zOff" > pos.dat
19 done
20 rm oldChain.pdb
```

Figure 9: `buildChain.sh`, a script to drive `gmx insert-molecules` to assemble a polymer chain from monomer `.pdb` files.

The above script (`buildChain.sh`) uses the utility `gmx insert-molecules` to construct a `.pdb` file for a polymer chain from `.pdb` files for the constituent monomers. The list of monomers is read from stdin.

The script leverages the ability of `gmx insert-molecules` to renumber the atoms and residues of each monomer it adds, so that the entire molecule is appropriately labeled.

The script needs to know where to place each successive monomer. The monomer `.pdb` files for this purpose are specially constructed in "standard orientation" along the chain (z) axis, with the first atom to be bonded at the origin.

Furthermore, the displacement vector from the bonded atom to the next bonded atom is included as a comment at the top of the file. This comment is read (lines 6 and 16) and converted to a z offset, which is added to the variable `zOff`. The current value of `zOff` is written to a short file `pos.dat`, which tells `gmx insert-molecules` where to insert the next monomer.

```
#!/bin/bash

nMon=$1
echo "PSB"
for i in `seq 1 $((nMon-2))`
do
   if (($RANDOM % 2));
   then
      echo "PSML"
   else
      echo "PSMR"
   fi
done
echo "PST"
```

Figure 10: `buildChainPS.sh`, a script to drive `buildChain.sh` to assemble an atactic polystyrene chain.

The above script (`buildChainPS.sh`) can be used with `buildChain.sh` to build an atactic polystyrene chain. The script simply constructs a list of monomer names to be piped to `buildChain.sh`:

```
buildChainPS.sh 25 | buildChain.sh
```

The number of monomers `nMon` is passed as a command-line argument. The random tacticity is accomplished by making a random choice in the loop over the number of monomers, with the use of `$RANDOM % 2`, which randomly evaluates to 0 or 1 each time it is accessed. The structure of the end monomers is different from the internal monomers, so they are written separately outside the loop.

3.6 Building a polymer `.itp` file

The script below (`itpMake.sh`) assembles an `.itp` file for a polymer from a set of specially constructed "monomer .itp files". `itpMake.sh` reads a list of monomer names from stdin, in the same way as `buildChain.sh` above. This script serves as an alternative to `gmx pdb2gmx`, which works well for proteins (and all-atom polymers, for which new residues can be defined and added to `aminoacids.rtp`).

The work is done by a (long) function `addMonITP`, defined at the top of the script. It is often convenient to define the function to be iterated separately from the iteration (as we did for the script `getDhDl.sh` above).

The actual body of the script is rather short (lines 108–129); it consists of writing headers to temporary files for the different declaration sections, looping over the monomers (lines 119–124), and using `cat` to write the completed declaration sections to stdout.

`addMonITP` reads through the monomer `.itp` file, incrementing the atom numbers, residue numbers, and charge group numbers throughout, and using `printf` to write the incremented values in a convenient format to a set of temporary output files for the various declaration sections (`[ atoms ]`, `[ bonds ]`, and so forth). Global variables `cAtom`, `cRes`, and `cChgrp` store the current "offset values" of these numberings.

Finally, note again the use of `exec 3&< $line.itp` to open the monomer `.itp` files for sequential access, so that the multiple `read` commands within `addMonITP` access successive lines in the file.

```
#!/bin/bash

# function to add one monomer
# all variables are global
function addMonITP {
itpFile=$1
# read header line; read counts
read 0<&3 line
read 0<&3 nAtom nChgrp nCons nVsite nExcl nBond nAng nDih
# read atom header
read 0<&3 line
read 0<&3 line
# copy atoms (iAtom is 1, res is 3, chgrp is 6)
for i in `seq 1 $nAtom`
do
   read 0<&3 f0 f1 f2 f3 f4 f5 f6 f7
   let f0=$f0+$cAtom
   let f2=$f2+$cRes
   let f5=$f5+$cChgrp
   printf "%5s%5s%5s%5s%5s%5s%10s%10s\n" $f0 $f1 $f2 $f3 $f4 $f5 $f6 $f7 >> fAtom.temp
done

# read constraint header
read 0<&3 line
read 0<&3 line
# copy constraints (atom nums are 1,2)
for i in `seq 1 $nCons`
do
   read 0<&3 f0 f1 f2 f3
   let f0=$f0+$cAtom
   let f1=$f1+$cAtom
   printf "%5s%5s%5s%10s\n" $f0 $f1 $f2 $f3 >> fCons.temp
done

#read vsites header
read 0<&3 line
read 0<&3 line
# copy vsites (atom nums are 1, 2, 3, 4)
for i in `seq 1 $nVsite`
do
   read 0<&3 f0 f1 f2 f3 f4 f5 f6 f7
   let f0=$f0+$cAtom
   let f1=$f1+$cAtom
   let f2=$f2+$cAtom
   let f3=$f3+$cAtom
   printf "%5s%5s%5s%5s%5s%10s%10s\n" $f0 $f1 $f2 $f3 $f4 $f5 $f6 $f7 >> fVsite.temp
done

# read exclusions header
read 0<&3 line
read 0<&3 line
# copy exclusions (atom nums are 1, 2)
for i in `seq 1 $nExcl`
do
   read 0<&3 f0 f1
   let f0=$f0+$cAtom
   let f1=$f1+$cAtom
   printf "%5s%5s\n" $f0 $f1 >> fExcl.temp
done

# read bonds header
read 0<&3 line
read 0<&3 line
# copy bonds (atom nums are 1, 2)
for i in `seq 1 $nBond`
do
   read 0<&3 f0 f1
   let f0=$f0+$cAtom
   let f1=$f1+$cAtom
   printf "%5s%5s\n" $f0 $f1 >> fBond.temp
done
```

Figure 11: First page of `itpMake.sh`, a script to assemble a polymer `.itp` file from `.itp` files for the constituent monomers.

```
72
73  # read angles header
74  read 0<&3 line
75  read 0<&3 line
76  # copy angles (atom nums are 1, 2, 3)
77  for i in `seq 1 $nAng`
78  do
79     read 0<&3 f0 f1 f2
80     let f0=$f0+$cAtom
81     let f1=$f1+$cAtom
82     let f2=$f2+$cAtom
83     printf "%5s%5s%5s\n" $f0 $f1 $f2 >> fAng.temp
84  done
85
86  # read dihedrals header
87  read 0<&3 line
88  read 0<&3 line
89  # copy dihedrals (atom nums are 1, 2, 3, 4)
90  for i in `seq 1 $nDih`
91  do
92     read 0<&3 f0 f1 f2 f3
93     let f0=$f0+$cAtom
94     let f1=$f1+$cAtom
95     let f2=$f2+$cAtom
96     let f3=$f3+$cAtom
97     printf "%5s%5s%5s%5s\n" $f0 $f1 $f2 $f3 >> fDih.temp
98  done
99
100 # increment counts
101 let cAtom=$cAtom+$nAtom
102 let cRes=$cRes+1
103 let cChgrp=$cChgrp+$nChgrp
104 }
105
106 # main program
107 # write headers into each temp file
108 echo "[ atoms ]" > fAtom.temp
109 echo "[ constraints ]" > fCons.temp
110 echo "[ virtual_sites3 ]" > fVsite.temp
111 echo "[ exclusions ]" > fExcl.temp
112 echo "[ bonds ]" > fBond.temp
113 echo "[ angles ]" > fAng.temp
114 echo "[ dihedrals ]" > fDih.temp
115
116 cAtom=0; cRes=0; cChgrp=0
117 # read monomers from stdin;
118 # open file descriptor for each monomer
119 while read line
120 do
121     exec 3< $line.itp
122     addMonITP
123     exec 3<&-
124 done
125
126 # concatenate files
127 cat fAtom.temp fCons.temp fVsite.temp fExcl.temp fBond.temp fAng.temp fDih.temp
128 # delete temp files
129 rm fAtom.temp fCons.temp fVsite.temp fExcl.temp fBond.temp fAng.temp fDih.temp
```

Figure 12: Second page of `itpMake.sh`.

3.7 Building a salt solution

As a final example of a job script, `buildSystem.sh` prepares the set of files needed to simulate a salt solution in which the ions are confined by a harmonic "umbrella potential". (The purpose of the simulation is to measure the resulting concentration profile; from the profile, the osmotic pressure as a function of concentration can be obtained, by analyzing the balance of forces on each slice of the concentration profile.) The script takes as input parameters the number of ions of each kind, the linear dimension of the system, and the "spring constant" (curvature) of the confining potential.

The script is unremarkable until the final portion (lines 37–60), in which the index (`.ndx`) and MD parameters (`.mdp`) files are generated. Because the confining potential acts independently on every ion, the index file must contain a group for every ion, and the `.mdp` file must contain a set of "pulling options" specifying the (identical) force acting on each of these groups.

The entries to the index file are generated by a brief script `writeIndex.sh`, which iterates over the number of ions to write the necessary lines to stdout. The main script `buildSystem.sh` calls `writeIndex.sh`, and appends its output by redirection (`>>`) to the existing index file.

The repetitive "pulling options" for all the ion groups are generated by a script `writePulls.sh`. This script uses the stream editor `sed` to modify a "template file", replacing keywords within the file with the corresponding variable values. Here, `pullHeader.mdp` is the template file for the initial lines in the section of `.mdp` that describe the pulling, with `NIONS` as the variable to be replaced. Similarly, `pullTemplate.mdp` is the template file for the pulling options for each ion group, with `ION` (the ion number), `CTR` (the center of the harmonic potential), and `KVAL` (the potential "spring constant") to be replaced by the corresponding variable values `$i`, `$kVal`, and `$ctr`.

```
1  #!/bin/bash
2  # arguments are # ions, spring constant, center position
3
4  nIons=$1
5  kVal=$2
6  ctr=$3
7
8  # copy the non-pull .mdp lines
9  cat mdTop.mdp
10 # copy the top pull .mdp lines, replace NIONS
11 sed "s/NIONS/${nIons}/g" < pullHeader.mdp
12
13 # loop over the ions
14 for i in `seq 1 $nIons`
15 do
16     # replace the template .mdp lines with correct values
17     sed "s/ION/${i}/g;s/KVAL/${kVal}/g;s/CTR/${ctr}/g" < pullTemplate.mdp
18 done
```

Figure 13: `writePulls.sh`, a script to construct the repetitive "pulling options" for all the ions.

```
; pull settings
pull = yes
pull-ngroups = NIONS
pull-ncoords = NIONS
pull-nstxout = 0
pull-nstfout = 0
```

Figure 14: `pullHeader.mdp`, a "template file" used by `writePulls.sh`.

```
pull-groupION-name=ionION
pull-coordION-type = umbrella
pull-coordION-geometry = direction-periodic
pull-coordION-vec = 0 0 1
pull-coordION-groups = 0 ION
pull-coordION-origin = CTR CTR CTR
pull-coordION-dim = N N Y
pull-coordION-k = KVAL
```

Figure 15: `pullTemplate.mdp`, a "template file" used by `writePulls.sh`.

```
#!/bin/bash

nOffset=$1
nIons=$2
for i in `seq 1 $nIons`
do
    echo "[ ion"$i" ]"
    echo $((i+nOffset))
done
```

Figure 16: `writeIndex.sh`, a script to construct the repetitive "index file" entries for all the ions.

```
#!/bin/bash
# nEach = # of + and - ions, lBox = box dimension, kVal = spring constant
nEach=$1
lBox=$2
kVal=$3

# make job folder
JOB=${nEach}ions_${lBox}box
mkdir $JOB

# copy job scripts into folder
cp equil.sh job.sh $JOB

# make the system folder
mkdir $JOB/System

# copy .mdp files
cp em.mdp nvt.mdp npt.mdp mdTop.mdp $JOB/System

# start the .top file
cp systemTop.top $JOB/System/system.top
cd $JOB/System

# add water
gmx solvate -box $lBox $lBox $lBox -cs spc216.gro -o solvate.gro -p system.top &>
solvate.log

# make a tpr file for genion to use
gmx grompp -f em.mdp -c solvate.gro -p system.top -o system.tpr -maxwarn 2 &> grompp.log

# add ions
cp solvate.gro system.gro
echo "2" | gmx genion -s system.tpr -o system.gro -p system.top -neutral -pname NA -pq 1
-np $nEach -nname CL -nq -1 -nn $nEach -rmin 0.45 &> genion.log

#create ndx file #
echo "q" | gmx make_ndx -f system.gro -o system.ndx &> make_ndx.log

# compute nOffset (index of last non-ion atom)
# get second line from solvate.gro
inputLine=`head -n 2 solvate.gro | tail -n 1`
# parse as input (eliminates stray spaces)
read -a args <<< "$inputLine"
nAtoms=${args[0]}
let nOffset=$nAtoms-2*$nEach

# write index file lines
cd ../..
writeIndex.sh $nOffset $nIons >> $JOB/System/system.ndx

# write pullCount.txt
let nIons=2*$nEach
newLine="pull-ngroups = "$nIons
echo $newLine > pullCount.txt
newLine="pull-ncoords = "$nIons
echo $newLine >> pullCount.txt

# write pull lines
writePulls.sh $nIons $kVal > pullLines.txt

# write .mdp file
cat mdTop.mdp pullCount.txt pullLines.txt > $JOB/System/md.mdp

# clean up
rm pullCount.txt pullLines.txt
rm $JOB/System/"#"*"#"
```

Figure 17: `buildSystem.sh`, a script to construct the set of input files to simulate a salt solution confined by an external "umbrella" potential.