Program 1 - matrices

Submit Assignment

Due Oct 14 by 11:59pm **Points** 160 **Submitting** a file upload
Available Sep 11 at 12am - Oct 16 at 11:59pm about 1 month

Program 1 – CS 344

This assignment asks you to write bash shell scripts to compute matrix operations. The purpose is to get you familiar with the Unix shell, shell programming, Unix utilities, standard input, output, and error, pipelines, process ids, exit values, and signals (at a basic level).

What you're going to submit is your script, called simply "matrix".

Overview

In this assignment, you will write a bash shell script that calculates basic matrix operations using input from either a file or stdin. The input will be whole number values separated by tabs into a rectangular matrix. Your script should be able to print the dimensions of a matrix, transpose a matrix, calculate the mean vector of a matrix, add two matrices, and multiply two matrices.

You will be using bash builtins and Unix utilities to complete the assignment. Some commands to read up on are `while`, `cat`, `read`, `expr`, `cut`, `head`, `tail`, `wc`, and `sort`.

Your script must be called simply "matrix". The general format of the matrix command is:

```
matrix OPERATION [ARGUMENT]...
```

Refer to man(1) (You can do this with the command `man 1 man`) for an explanation of the conventional notation regarding command syntax, to understand the line above. Note that many terminals render italic font style as an underline:

```
matrix OPERATION [ARGUMENT]...
```

Specifications

Your program must perform the following operations: dims, transpose, mean, add, and multiply. Usage is as follows:

```
matrix dims [MATRIX]
matrix transpose [MATRIX]
matrix mean [MATRIX]
```

```
matrix add MATRIX_LEFT MATRIX_RIGHT
matrix multiply MATRIX_LEFT MATRIX_RIGHT
```

The dims, transpose, and mean operations should either perform their respective operations on the file named *MATRIX*, or on a matrix provided via stdin. The add and multiply operations do not need to process input via stdin.

- dims should print the dimensions of the matrix as the number of rows, followed by a space, then the number of columns.
- transpose should reflect the elements of the matrix along the main diagonal. Thus, an MxN matrix will become an NxM matrix and the values along the main diagonal will remain unchanged.
- mean should take an MxN matrix and return an 1xN row vector, where the first element is the mean of column one, the second element is the mean of column two, and so on.
- add should take two MxN matrices and add them together element-wise to produce an MxN matrix. add should return an error if the matrices do not have the same dimensions.
- multiply should take an MxN and NxP matrix and produce an MxP matrix. Note that, unlike addition, matrix multiplication is not commutative. That is A*B != B*A.

Here is a brief example of what the output should look like.

```
$ cat m1
1       2       3       4
5       6       7       8
$ cat m2
1       2
3       4
5       6
7       8
$ ./matrix dims m1
2 4
$ cat m2 | ./matrix dims
4 2
$ ./matrix add m1 m1
2       4       6       8
10      12      14      16
$ ./matrix add m2 m2
2       4
6       8
10      12
14      16
$ ./matrix mean m1
3       4       5       6
$ ./matrix transpose m1
1       5
2       6
3       7
4       8
$ ./matrix multiply m1 m2
50      60
114     140
```

You must check for the right number and format of arguments to matrix. This means that, for example, you must check that a given input file is readable, before attempting to read it. You are not required to test if the input file *itself* is valid. In other words, the behavior of matrix is undefined when the matrix input is not a valid matrix. for the purposes of this assignment, a valid matrix is a tab-delimited table containing at least one element, where each element is a signed integer, every entry is defined, and the table is rectangular.

The following are examples of invalid matrices. Our grading script (see below) does not send these as *input* into your matrix program. Similarly, you aren't allowed to send matrices with these characteristics as *output* from your script either, and we will test your output to make sure it doesn't. If any of these are found in your output, the grading script will deduct points:

- An empty matrix.
- A matrix where the final entry on a row is followed by a tab character.
- A matrix with empty lines.
- A matrix with any element that is blank or not an integer.

Here is a valid matrix file, m1:

```
$ cat m1
8	5	6
3	2	2
1	6	7
5	0	7
2	2	4
$ cat -A m1   # The '-A' flag shows tabs as '^I' and newlines as '$'. This is a good way to check corre
ctness.
8^I5^I6$
3^I2^I2$
1^I6^I7$
5^I0^I7$
2^I2^I4$
$
```

If the inputs are valid -- your program should output only to stdout, and nothing to stderr. The return value should be 0.

If the inputs are invalid -- your program should output only to stderr, and nothing to stdout. The return value should be any number except 0. The error message you print is up to you; you will receive points as long as you print *something* to stderr and return a non-zero value.

Your program will be tested with matrices up to dimension 100 x 100.

Though optional, I do recommend that you use temporary files; arrays are not recommended. For this assignment, anytemporary files you use should be put in the current working directory. (A more standard place for temporary files is in /tmp but don't do that for this assignment; it makes grading easier if they are in the current directory.) *Be sure any temporary file you create uses the process id as part of its name, so that there will not be conflicts if the program is running more than once.* Be sure you remove any temporary files when your program is done. You should also use the trap command to catch interrupt, hangup, and terminate signals to remove the temporary files if the program is terminated unexpectedly.

All values and results are and must be integers. You may use the `expr` command to do your calculations, or any other bash shell scripting method, such as `((expr))`. Do not use any other languages other than bash shell scripting: this means that, among others, **awk, sed, tcl, bc, perl, & the python languages and tools are off-limits for this assignment.** Note that `expr` only works with whole numbers. When you calculate the average you must round to the nearest integer, where half values round away from 0 (i.e. 7.5 rounds up to 8, but -7.5 rounds down to -8). This is the most common form of rounding. When doing truncating integer division (as bash does), this formula works to divide two numbers and end up with the proper rounding:

```
(a + (b/2)*( (a>0)*2-1 )) / b
```

You can learn more about rounding methods here:

Rounding – Wikipedia (https://en.wikipedia.org/wiki/Rounding#Round_half_away_from_zero)

Grading With a Script

To make it easy to see how you're doing, you can download the actual grading script here:

p1gradingscript

To use the script, just place it in the same directory as your matrix script and run it like this:

```
$ ./p1gradingscript matrix
```

Be aware that this script might take a minute or two to run, depending on how speedy your algorithms and code are, since there's a lot of matrix math in there. The grading script has generous time-outs for each test -- if your script takes too long to complete a command, the test will time out and you will receive 0 points for that particular test. If you have any doubt about your running time being acceptable, just add a note into your code as a comment at the top to warn the TA. When we run your script for grading, we will do this to put your results into a file we can examine more easily:

```
$ ./p1gradingscript matrix > grading_result.username
```

To compare yours to a perfect solution, you can download here a completely correct run of the script that shows what you should get if everything is working correctly:

p1perfectoutput

The p1gradingscript itself is a good resource for seeing how some of the more complex shell scripting commands work, too. The script uses two functions for most of the work; expect_error and expect_success. The first is used to test commands that are expected to fail. The second is used if the command is expected to succeed.

Several options have been added to help with debugging and to speed testing. Here is the list of flags available.

This first section processes the flags used in while calling the grading script.
h - will list the proper syntax and list of flags, then exits the code
d - enter debug mode. Failures will send outpipe or errpipe to log. Matrices will be saved.
u - enters unit testing mode. Grading script will exit after first error.
e - The flag for error tests
i - The flag for dims tests
t - The flag for transpose tests
m - The flag for mean tests
a - The flag for add tests
l - The flag for multiply tests
g - The flag for all tests

```
./p1gradingscript -duel matrix
```

will run only error and multiply test sections, exiting upon the first error, and storing any errors in a log while also saving any matrices used by a failed test.

Summary

Your script must support the following operations:

- **matrix dims** [*MATRIX*]
  - Prints error message to stderr, nothing to stdout and return value != 0 if:
    - Argument count is greater than 1 (e.g. `matrix dims m1 m2`).
    - Argument count is 1 but the file named by argument 1 is not readable (e.g. `matrix dims no_such_file`).
  - Otherwise, prints "*ROWS COLS*" (Space separated!) to stdout, nothing to stderr, and returns 0.
- **matrix transpose** [*MATRIX*]
  - Prints error message to stderr, nothing to stdout and return value != 0 if:
    - Argument count is greater than 1 (e.g. `matrix transpose m1 m2`).
    - Argument count is 1 but the file named by argument 1 is not readable (e.g. `matrix transpose no_such_file`).
  - Otherwise, prints the transpose of the input, in a valid matrix format to stdout, nothing to stderr, and returns 0.
- **matrix mean** [*MATRIX*]
  - Prints error message to stderr, nothing to stdout and return value != 0 if:
    - Argument count is greater than 1 (e.g. `matrix mean m1 m2`).
    - Argument count is 1 but the file named by argument 1 is not readable (e.g. `matrix mean no_such_file`).
  - Otherwise, prints a row vector mean of the input matrix, in a valid matrix format to stdout, nothing to stderr, and returns 0. All values must round to the nearest integer, with ***.5 values rounded away from zero.
- **matrix add** *MATRIX_LEFT MATRIX_RIGHT*
  - Prints error message to stderr, nothing to stdout and return value != 0 if:
    - Argument count does not equal 2 (e.g. `matrix add m1 m2 m3` or `matrix add m1`).
    - Argument count is 2 but the file named by either argument is not readable (e.g. `matrix add m1 no_such_file`).
    - The dimensions of the input matrices do not allow them to be added together following the rules of matrix addition.
  - Otherwise, prints the sum of both matrices, in a valid matrix format to stdout, nothing to stderr, and returns 0.

- **matrix multiply** *MATRIX_LEFT MATRIX_RIGHT*
  - Prints error message to stderr, nothing to stdout and return value != 0 if:
    - Argument count does not equal 2 (e.g. `matrix multiply m1 m2 m3` or `matrix multiply m1`).
    - Argument count is 2 but the file named by either argument is not readable (e.g. `matrix multiply m1 no_such_file`).
    - The dimensions of the input matrices do not allow them to be multiplied together following the rules of matrix multiplication.
  - Otherwise, prints the product of both matrices, with the first argument as the left matrix and the second argument as the right matrix, in a valid matrix format to stdout, nothing to stderr, and returns 0. (`matrix multiply A B` should return A*B, not B*A)

`

An invalid command must result in an error message to stderr, nothing to stdout, and a return value != 0.

Hints

- Try writing each part as a separate function (see the **1.4 bash Functions** reading). You can use `$1 "${@:2}"` to call the function named by argument 1, with the remaining arguments passed to it. For example, `matrix multiply m1 m2` will expand `$1 "${@:2}"` to `multiply m1 m2` inside your script, which will call the function named "multiply". This fancy bash method of calling functions would go at the bottom of your script, after the functions are defined.
- You'll need to use the read command extensively to read in data from a file. Note that it reads in one line at a time *from the stdin buffer* and stores the line in a variable called REPLY, unless specified. Generally, read is used in a while loop, where a file is redirected to stdin *of the while loop*. Calling `read < myfile` multiple times will repeatedly read the *first* line of myfile.
- The expr command and the shell can have conflicts over special characters. If you try expr 5 * (4 + 2), the shell will think * is a filename wild card and the parentheses mean command grouping. You have to use backslashes, like this:

```
expr 5 \* \( 4 + 2 \)
```

- I ***highly*** recommend that you develop this program directly on the class server (os1). Doing so will prevent you from having problems transferring the program back and forth, which can cause compatibility issues. Keep in mind that students caught developing on flip risk losing points on the assignment. If you are using Windows as your operating system, windows may replace newlines in files with newline-carriage returns. These will be displayed as ^M characters in vim. To remove them, use the utility `dos2unix`.

```
$ dos2unix bustedFile
```

- Consider the following snippet of code. This will allow you to use one variable to hold the path to a file with passed-in contents, no matter if they came via stdin or via a file specified on the command line:

```
datafilepath="datafile$$"
if [ "$#" = "1" ]
then
    cat > "$datafilepath"
elif [ "$#" = "2" ]
then
```

```
    datafilepath=$2
fi
```

- Want to remove the last character from a string? Try this:

```
$ myvar="abcde"
$ myvar=${myvar%?}
$ echo $myvar
abcd
```

- Want to display all the characters in a file or string, even the non-printing tabs and newlines? Try `cat -A`

What to Turn in and When

Simply submit your "matrix" script file. Your script must be entirely contained in this one file. Do not split this assignment into multiple files or programs. As our Syllabus says, please be aware that neither the Instructor nor the TA(s) are alerted to comments added to the text boxes in Canvas that are alongside your assignment submissions, and they may not be seen. No notifications (email or otherwise) are sent out when these comments are added, so we aren't aware that you have added content! If you need to make a meta-comment about this assignment, please include it in a #comment near the top of the script itself, or email the person directly who will be grading it (see the **Home** page for grading responsibilities).

The due date given below is the last minute that this can be turned in for full credit. The "available until" date is NOT the due date, but instead closes off submissions for this assignment automatically once 48 hours past the due date has been reached, in accordance with our **Syllabus Grading Policies**.

Grading

148 points are available for your script (which is the only file you'll submit) successfully passing the grading script, while the final 12 points will be based on your style, readability, and commenting. Comment well, often, and verbosely: we want to see that you are telling us WHY you are doing things, in addition to telling us WHAT you are doing.

The TAs will use this exact set of instructions: **Program1 Grading.pdf** to grade your submission.