CS2035 - Assignment 2 - 2018

Arrays and Efficiency

Out: February 12th, 2018

In: March 4th, 2018 at 11:55 pm via Owl

Introduction

This MATLAB assignment requires you to write six (6) MATLAB functions, each contained in its own script file. The first three (avg1.m, avg2.m, avg3.m) compute the average values of an input matrix, and the second three functions (stderr1.m, stderr2.m, stderr3.m) compute the standard error values of an input matrix.

You will learn about:

- vectorized operations on matrices;
- just-in-time (JIT) compilation;
- writing your own vectorized operations;
- timing runtime execution of your code;
- plotting your data;
- how to automate tests on variable sized input.

This assignment is worth $11\frac{2}{3}\%$ of the course mark.

Part I: Computing Averages Efficiently

For the first part of this assignment you will write several MATLAB functions that compute the average of an input matrix in different ways.

First note how the built-in MATLAB function `mean` works for an `m` by `n` input matrix `A=rand(m,n)`. For example, if `m=100` and `n=5`, we might obtain the output

```
A = rand(100,5);
mean(A)
ans =
    0.5212    0.5810    0.4844    0.5070    0.4944
```

Thus, the `mean` function treats the input matrix as a sequence of 5 columns of length 100 and computes the average of each column. Recall that for a set of m samples of a random variable x, its average $\overline{x}$ (an estimate of the mean) is given by

$$\overline{x} = \frac{1}{m}\sum_{i=1}^{m} x_i \,.$$

You are to write three MATLAB functions that replicate this behaviour. The first function (`avg1.m`) should take an input matrix `X`, compute `[m,n] = size(X)`, and then compute the output vector `avg` using a pair of nested loops. The outer loop computes a single element of `avg`, and the inner loop computes the sum of the elements of the corresponding column of `X` needed to compute the given element of `avg`. For this function, the size of `avg` should increase with each iteration through the loop, i.e., **do not** allocate space for `avg` before starting the loop. The intention here is that MATLAB will not invoke JIT compilation for this function.

Make sure that the output of your function `avg1.m` replicates the behaviour of `mean`. It should compute the same vector of values that `mean` returns, and it should produce a single value if the input is a (row or column) vector. You can ensure this latter behaviour by checking to see if the input array `X` is a row vector, and if it is by then converting it into a column vector.

The second function (`avg2.m`) modifies `avg1.m` so that MATLAB's JIT compilation will be used. Make sure that space for any arrays is allocated before any loops begin and that any variables (including the loop indices!) are defined before the loops begin.

The third function (`avg3.m`) computes the same thing as your first two functions, but uses vectorization. For this function you can use the built-in MATLAB function `sum` to compute the average of the input matrix `X`. Note that, aside from the code from the previous functions used to convert an input row vector to a column, this function can compute the average of a matrix in one or two lines.

Part II: Computing Standard Errors Efficiently

For the second part of this assignment you will repeat what you did for the first part for standard error calculations rather than averages.

First note that the built-in MATLAB function `std` works in the same way as the `mean` function for an input matrix, in that it will compute the standard error of each column of the input matrix. For example, we might obtain the output

```
A = rand(100,5);
std(A)
ans =
    0.2752    0.2669    0.2741    0.2861    0.2754
```

Thus, the functions you write should replicate this behaviour. Recall that for a set of m samples of a random variable x, the standard error s_x (estimate of the standard deviation) is given by

$$s_x = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(\overline{x} - x_i)^2}\,.$$

Note that for each of your functions, you will have to detect when the input is a row vector and convert it to a column, just as for your functions from the previous part. You can reuse the same code for this.

The first function you write for this part (`stderr1.m`) should take an input matrix `X`, compute `[m,n] = size(X)`, compute the average `avg` using your first average function `avg1.m`, followed by a pair of nested loops to compute the output vector `stderr`. The outer loop computes a single element of `stderr`, and the inner loop computes the sum in the expression for s_x above for the corresponding column of `X`. The outer loop then completes by evaluating the remainder of the expression for s_x to compute the value of the given element of `stderr`. Once again, ensure that `stderr1` **does not** allocate space for `stderr` before starting the loop.

The second function for this part (stderr2.m) modifies stderr2.m so that MATLAB's JIT compilation will be invoked. Again, make sure that space for any arrays is allocated before any loops begin and that any variables (including the loop indices!) are defined before the loops begin. To make for a more interesting comparison later, use a vectorized computation for `avg` (using either your `avg3` or `mean`), i.e., do not use `avg2` to compute the averages.

The third function (stderr3.m) for this part computes the same thing as your first two functions for this part, but uses vectorization. For this function you can use the built-in MATLAB functions `sum` and `sqrt` to compute the standard error of the input matrix `X`. Note that, aside from the code from the previous functions used to convert an input row vector to a column, this function can compute the standard error of a matrix in one or two lines. Use either your `avg3` or `mean` to compute the averages.

Part III: Program Validation

Write a test script file `tests.m` that shows that your functions compute the correct results by comparing them to the result of the corresponding built-in MATLAB function.

This script should consist of two parts, each preceded by a section title, i.e., a commented line starting with `%%` indicating the title of the section. First construct a matrix `X=rand(100,6)` and a row vector `y=rand(1,100)`. The first section computes the result of calling `avg1`, `avg2`, and `avg3` on both `X` and `y` and compares them to the output of `mean(X)` and `mean(y)` All four function calls should produce the same result. Make sure not to suppress the output of your function calls, and use `fprintf` or `disp` to label the output of each function call.

The second section computes the result of calling `stderr1`, `stderr2` and `stderr3` on the same `X` and `y` and compares them to the output of `std(X)` and `std(y)`. All four function calls should produce the same result. Again, label the output of each function call.

Part IV: Measuring Runtime Performance

For this part of the assignment you will write code to test the runtime performance of the functions that you wrote. This will involve writing a script file called timing.m that will test many runs of your code and plot the runtime for different sizes of input matrix.

You will build this file in a few stages. To begin with, start with the lines

```
m = 1000;
n = 1e4;

X = rand(m,n);
```

This will construct a random 1000×10000 matrix to test your functions with. Then, for each of the standard error functions you wrote, write a code segment like

```
tic
stderr*(X);
elapsed = toc;
fprintf('stderr* on %dx%d array: %f s\n',m,n,elapsed);
```

to compute the execution time, where `*` gets replaced by the number of the function you are testing. Then add one more line that computes and prints out the runtime for `std(X)`. When this is working, you should find that `stderr1` is the slowest, and either `stderr3` or `std` is the fastest, with `stderr2` in between. The results will vary when you run your script multiple times.

The second stage for building your timing script is to have your script time several runs of each of your functions and print out only the average runtime (this is to give a fair comparison of the algorithms). To accomplish this, add a line `runs=10` to the beginning of your script file. Then before each `tic` statement (four of them) start a for loop with index `i` running from `1` to `runs`, and replace

```
elapsed = toc;
```

with

```
elapsed(i) = toc;
```

This will store the runtime of each run in a vector `elapsed`. Then, in the `fprintf` statement, replace '`elapsed`' with '`mean(elapsed)`'. When you have done this for all four functions (`stderr1`, `stderr2`, `stderr3` and `std`), your code should print out four lines as before, but now with the average runtime of 10 runs of each function.

The next stage of building your timing script is to have your script test different sizes of input matrix. To start this process, add the following two lines at the top of your script file:

```
pow = 4;
points = pow+1;
```

and replace the line reading

```
n = 1e4;
```

with

```
n = logspace(0,pow,points);
```

This will create a vector `n`, the entries of which are the number of columns of the input matrix `X` on a given run. This way the number of columns increases by a factor of 10 each time the functions are tested. Then add another for loop with loop index `k`, which starts before the line that assigns the value to `X` and ends at the bottom. To get the loop to work, wherever `n` appears within the loop you must change it to `n(k)` (there should be five of places where `n` appears). Now when you run your script, it will test all four functions 5 times, i.e., `pow+1` times, with exponentially increasing input size.

The last stage of building your timing script is to plot the results of the runtime tests. The easiest way to do this is the following. Before the first for loop add a line

```
T = zeros(4,points);
```

that initializes a variable `T` to hold the timing data. It has 4 rows, one for each of the functions you are testing, and `points` columns, one for each of the different input sizes you are testing. To fill this array, you only need to add a line of the form

```
T(*,k) = mean(elapsed);
```

before each `fprintf` statement, with `*` replaced by the numbers 1 through 4 (1 for `stderr1`, 2 for `stderr2`, etc.). Once you have done this, you can plot your results on a log-log scale simply by calling `loglog(n,T)`. Add a line at the end of your file to do this. Then annotate your plot with a legend (including the four titles `'Variable Array Loops'`, `'JITC'`, `'Vectorized'` and `'MATLAB'`), a title (`'Runtime Comparison for Standard Error Algorithms'`), and label the x (`'Number of Size 1000x1 Input Columns'`) and y (`'runtime'`) axes.

Once you have all of this working, increase the value of `pow` to 5, which will give another data point on your plot.

Submitting the Assignment

To prepare your assignment for submission, make a wrapper script `a2.m` that starts with the code

```
%% CS2035B Assignment 2: Arrays and Efficiency

%% Identification
% Your Name:
% Your Student Number
```

and then has the following content:

- Include a section for each of your functions (6 of them) with a section heading of the form

  ```
  %% FunctionName Source Code
  ```

  followed by a line reading

  ```
  % <include>FunctionName.m</include>
  ```

  where ‘`FunctionName`’ is replaced with the name of your function. This will include the source code in your published output.

- Include a section for each of your scripts (tests.m and timing.m). Recall that you can call a script file simply with the name of the file. Make sure to add **before each script call** a section heading (preceded with %%) to describe what the script does and a line

  ```
  % <include>ScriptName.m</include>
  ```

  to include the source code.

Note that when you run A2.m it may take some time to compute all of the runtimes for your code, and that when you call publish on A2.m it will have to recompute the runtimes, so you may wish to reduce the value of pow in timing.m while you are getting this set up. But make sure to **increase pow back to 5 before you submit**!

To complete the assignment, make a new folder in your OWL Drop Box called A2 and place in that folder your nine (9) MATLAB script files together with the file a2.pdf obtained by running MATLAB's Publish on a2.m. **You must include the pdf output to receive full marks for your assignment.**.