Measuring and Improving Code Performance

1 Measuring Performance

Matlab offers two main tools for measuring the performance of code and finding out its bottlenecks. The first are the timing functions, which can be used to measure how long pieces of code and functions take to run. The second is the profiling tool of Matlab, which can measure the performance of large pieces of code (several functions) and displays results in a nice interface.

1.1 Timing

We can time pieces of code with the functions `tic` and `toc`. When `tic` is executed, it starts an internal stopwatch and returns a number (optional) representing the time when `tic` was executed. The function `toc` displays how much time has elapsed in seconds since the function `tic` was executed. Subsequent calls to `toc` display how much time has elapsed since the first `tic` has been executed. We can start a new timer by calling `tic` again.

We can use `tic` and `toc` to time pieces of code.

```
% measure how long it takes to execute the code below
% start timer
tic
n = 10000;
a = rand(n, n);
b = rand(n, 1);
a*b;
toc

% we can store the time in a variable
t = toc;

% measure time depending on size of matrix
n_values = 1:500:10000;
t = zeros(length(n_values), 1);
for i = 1:length(n_values)
    tic;
    n = n_values(i);
    a = rand(n, n);
    b = rand(n, 1);
    a*b;
    t(i) = toc;
```

```
end
plot(n_values, t)
```

We can create more than one timer by saving the output of `tic`, which can then be passed to `toc` to measure the time difference with respect to the respective `tic`.

```
total_time = tic;
repetitions = 100;
times = zeros(length(repetitions), 1);
for rep = 1:repetitions
    t = tic;
    % do computation
    mean(randn(10000000, 1));
    % store time
    times(rep) = toc(t);
end
% average time
disp(sprintf('Average time: %.2f seconds (%d repetitions)', ...
     mean(times), length(times)));
% total time
disp(sprintf('Total time: %.2f seconds', toc(total_time)));
```

The functions `tic` and `toc` are good to time small pieces of code, but to get a more precise time estimation it is necessary to re-run the same code several times and average the `tic` and `toc` results.

A more rigorous timing measure can be obtained with the function `timeit`. We can use `timeit` to measure the typical time it takes to run a function. Given a function, `timeit` runs it several times and reports the median time it takes to complete its execution.

Create a test function:

```
% test_timeit.m
function [a, b] = test_timeit(X)
beta = rand(size(X, 2), 1);
a = X*beta;
if nargout > 1
    b = exp(a);
end
```

Use `timeit` to time the execution of the function:

```
% specify the input to test_timeit
X = rand(5000);
timeit(@() test_timeit(X))
% specify how many outputs the function being timed should
   return
timeit(@() test_timeit(X), 2)
```

1.2 Profiling

The command `profile` can be used to profile large pieces of code. We can start the profiler by executing `profile on`. Then, any code that we run afterwards will be automatically timed. We can then call `profile viewer` to stop the profiler and see the results in a new window. Let's first create a test code:

```
% test_profiler.m
clear all
clc
% Load stock data
data = readmatrix('AAPL.csv');
% Compute intraday log-returns
prices = reshape(data(:, 3), 78, []);
returns = diff(log(prices));
% Compute realized variance
RV = sum(returns.^2);
% Generate bootstrap samples to compute confidence interval
total_samples = 10000;
RVs = zeros(total_samples, length(RV));
pbar = waitbar(0, 'Bootstrapping');     % progress bar
for i = 1:total_samples
    % generate bootstrap indices
    indices = unidrnd(size(returns, 1), size(returns, 1), 1);
    % bootstrap sample of the returns
    returns_sample = returns(indices, :);
    % compute statistic for each day
    RVs(i, :)  = sum(returns_sample.^2);
    % generate progress bar
    waitbar(i/total_samples, pbar);
end
close(pbar);
% Compute confidence interval
CI = quantile(RVs, [0.005 0.995])';
% Generate timestamps for plotting
dates = data(1:78:end, 1);
years = floor(dates./10000);
months = floor((dates - years.*10^4)./100);
days = floor(dates - years.*10^4 - months.*10^2);
sdates = datenum(years, months, days);
% Plot realized variance with correct x-ticks
plot(sdates, RV, 'k');
datetick('x', 'yyyy');
grid on;
% Shade confidence interval region
hold on;
f = fill([sdates; flip(sdates)], [CI(:, 1); flip(CI(:, 2))],
   'k');
f.FaceAlpha = 0.2;
```

```
42 f.EdgeAlpha = 0;
```

Now, we can use the profiler to analyze the code for bottlenecks. Clear the workspace before running the profiler.

```
1 clear all
2 profile on
3 test_profiler
4 profile viewer
```

The command `profile viewer` displays the window shown in Figure 1.

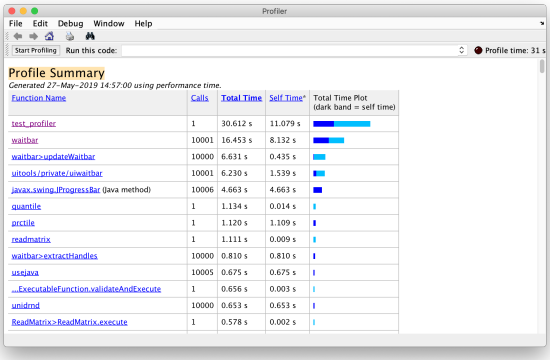

Figure 1: Profile Summary Window.

The profiler window shows all functions called while the profiler was on. The functions are sorted by total executing time. You can click on each function to see a more detailed break down of the execution. Click on the `test_profiler` function, and you should see a window similar to what Figure 2 displays.

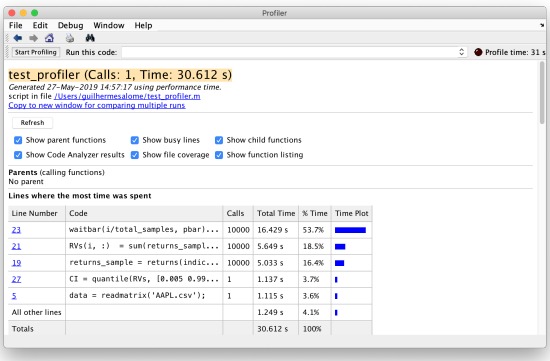

Figure 2: Profiler Detail Window.

We see that the `test_profiler` function took around 30 seconds to finish running. The window also shows the lines where most of the time was spent. We see that out of the 30 seconds, almost 5 seconds were spent on generating bootstrap samples from the original returns. That line was executed 10000 times. You can click on the line number to see the code around it. Click on the button for line `19`. You should see a window similar to what is displayed in Figure 3.

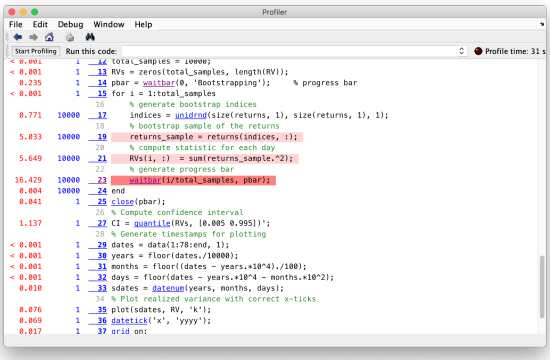

Figure 3: Profiler Line by Line Timing.

The lines displayed in darker red are the ones that take longer to execute. Lines 19 and 21 take about the same time to execute, while updating the progress bar takes the longest. We can use this information to identify lines that are inefficient, update the code and test again.

We can save the profiler results by executing the command `profsave`, which stores the results in a folder named `profile_results`. The file `file0.html` contains the same information as the profiler summary window, displayed in Figure 1.

Using the profiler is useful when working with a data set that is large. The idea is to use a small subset of the data to develop your code, then time it to see how long it takes. Then, you extrapolate how much time it would take to run your code on the entire data set. If the total time is acceptable then you are done. However, if it is not, then you should profile the code and see where you could improve it before running on the entire data set.

2 Efficiency Tips

We will now cover some general tips for efficiency when coding. Let's use the functions `tic` and `toc` to check the difference in execution time.

Avoid creating temporary objects:

```
times = zeros(1000, 1);
for i = 1:1000
    tic
    %% Code to time
    A = 2*randn(10000, 100);
    B = A.^2;
    % store time
    times(i) = toc;
end
% report median
disp(sprintf('Median execution time: %.4f seconds', median(
   times)));

times = zeros(1000, 1);
for i = 1:1000
```

```
    tic
    %% Code to time
    B = (2*randn(10000, 100)).^2;
    % store time
    times(i) = toc;
end
% report median
disp(sprintf('Median execution time: %.4f seconds', median(
   times)));
```

The object stored in `A` is temporary, created at the beginning of each loop, then used to compute `B`, and then reassigned when a new loop starts. We can avoid creating and deleting `A` by moving the `randn` command into `B`.

Pre-allocate matrices whenever possible.

```
%% Append to vector
tic
a = [];
for j = 1:10000
    a = vertcat(a, randn(1));
end
disp(sprintf('Time: %.4f seconds', toc));
%% Pre-allocate and then modify vector
tic
a = zeros(10000, 1);
for j = 1:10000
    a(j) = randn(1);
end
disp(sprintf('Time: %.4f seconds', toc));
```

Creating a matrix and then extending it is slower than pre-allocating the matrix. Extending the matrix means Matlab has to allocate more and more memory at each loop, recreating the vector `a` at each iteration. This reduces the efficiency of the code and makes it slower. Pre-allocating the matrix allows for a single block of memory to be allocated to the vector `a`.

Use sparse matrices to save memory when dealing with a large number of zeros.

```
A = eye(10000);                          % not sparse
B = sparse(eye(10000));                  % sparse
whos A B;
```

Sparse arrays only stores the nonzero elements in memory. It is possible to visualize the sparsity of a matrix with the function `spy`.

Vectorize operations if possible:

```
m_size = 1000000;
x = randn(m_size, 1);
y = rand(m_size, 1);
%% Not vectorized
z = zeros(m_size, 1);
tic
```

```
for i = 1:m_size
    z(i) = x(i)*y(i);
end
disp(sprintf('Time: %.4f seconds', toc));
%% Vectorized
clear z
tic
z = x.*y;
disp(sprintf('Time: %.4f seconds', toc));
```

A vectorized operation can run faster than a for-loop if it uses a built-in function. It is also more self-documenting than a for-loop and often leads to fewer lines of code.

3 Generating and Using C Code with Matlab

We can execute code written in C, C++ or Fortran with Matlab. To do so, we need to create a binary file named `MEX`, which can then be called in Matlab as a regular function. To create `MEX` files, we will use a Matlab tool that can convert Matlab code to some other language. For this section you will need to install Matlab Coder (or learn how to code in C or C++). You can also execute `coder` on Matlab to get a prompt to install it.

We will learn how to generate the `MEX` files with an example. Write the following function:

```
% euclidean_dist.m
function [distances, point_min, point_max] = euclidean_dist(x
   ,points)
% Compute Euclidean norm between x and the points
distances = sum((x-points).^2).^0.5;
% Obtain points that maximize and minimize the distance
[~, id_min] = min(distances);
[~, id_max] = max(distances);
% Save these points
point_min = points(:, id_min);
point_max = points(:, id_max);
end
```

This functions takes a column vector `x`, and a matrix containing multiple column vectors `points`, then computes the Euclidean distance between `x` and the other points. The functions has three outputs, the distance between all points, the vector closest to `x` and the vector farthest from `x`.

Now, we need to use Matlab's Code Analyzer. To do so, add the comment `%#codegen` right after the definition of the function `euclidean_dist`:

```
function [distances, point_min, point_max] = euclidean_dist(x
   ,points) %#codegen
```

This will trigger the Code Analyzer to check for any issues that could prevent the Matlab code from being converted to C/C++. If there are no errors, a green square will appear on the upper right-hand side of the Editor window. If there are errors, then the lines with issues will be underlined in red. You can hover the mouse on top of the line to see a

message explaining the issue and how to fix it. Most of the features of the programming language are supported by the converter, but some are not. For a full list of the supported and unsupported features, see this reference page.

Figure 4 displays the results of the Code Analyzer for the function `euclidean_dist`.

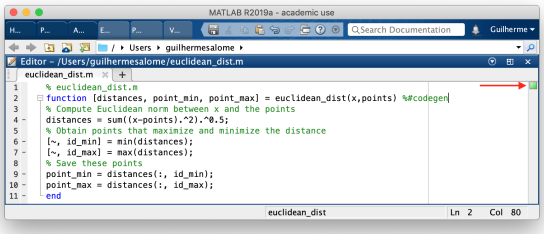

Figure 4: Code Analyzer Without Errors.

Now, we will use the function `coder.screener` to call the Code Generation Readiness Tool. This tool also checks for possible issues that can arise when generating the C/C++ code.

```
coder.screener('euclidean_dist')
```

Figure 5 shows the output of the code above. It indicates the function we created should not lead to issues when being converted to C/C++ code.

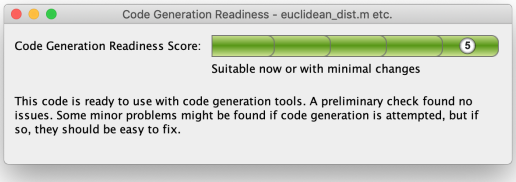

Figure 5: Output of Code Generation Readiness Tool.

We can now generate the `MEX` file with the `codegen` command. The `MEX` file is based on C code, which is statically typed. This means we need to define the type (class) and size of the inputs to the function when generating the lower-level code. We start with the case where the size of the inputs is fixed.

3.1 Fixed-Size Inputs

The type and size of inputs can be inferred when generating the lower-level code by giving an example to the command `codegen`.

```
% create example for codegen
x = [1; 2; 3];                          % size 3 x 1
points = rand(3, 10);                   % size 3 x 10
```

To create the `MEX` file, we call `codegen` by passing it the name of the file we want to convert and the example inputs:

```
codegen -report euclidean_dist.m -args {x, points}
```

The option `-report` generates a report that we can use to debug any issues that arise during the conversion. After the `-report`, we have the file name, in this case `euclidean_dist.m`, and then the option `-args` and a cell containing the example inputs. The `codegen` command infers the type and size of the inputs from the type and size of `x` and `points`.

When we execute the code above you will see an error. The error is depicted in Figure 6.

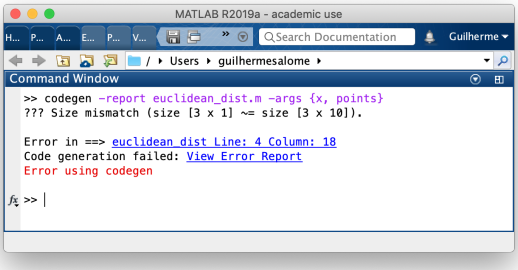

Figure 6: Codegen Error.

The error says there is some type of mismatch in the code. We can get more detail by clicking on `View Error Report`, which will open the window depicted in Figure 7.

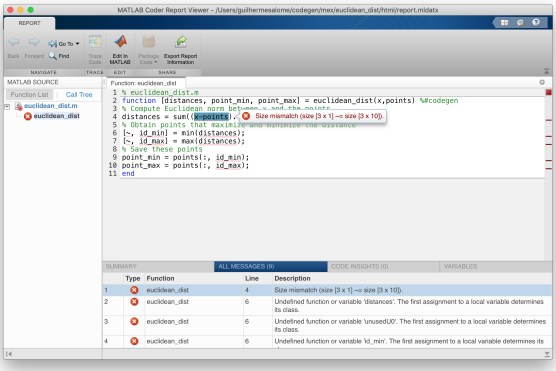

Figure 7: Codegen Report and Error Message.

The error report shows that there is a mismatch in size between `x` and `points`. It is not an issue in the Matlab code, because when we execute `x-points`, the vector `x` is broadcasted to have the same size of `points`. However, it leads to an issue when converting the code to C. We can fix it, by using `repmat` on the vector `x` to avoid the mismatch:

```
% Modify the line:
distances = sum((x-points).^2).^0.5;
% To:
distances = sum((repmat(x, 1, size(points, 2))-points).^2)
   .^0.5;
```

We can now compile the code by running:

```
codegen -report euclidean_dist.m -args {x, points}
```

The code generation should complete successfully. A file named `euclidean_dist_mex.mexmaci64` will be created. This file contains the binaries that can be executed by Matlab as a regular function. We can compare the time it takes to run the original function and the compiled function:

```
% use timeit to compare execution time of the functions
sprintf('Original Function: %.6f seconds', timeit(@()
   euclidean_dist(x, points), 3))
sprintf('MEX Function: %.6f seconds', timeit(@()
   euclidean_dist_mex(x, points), 3))
```

In this case, there is little gain since vectorized and simple operations in Matlab are already very fast. However, the gains in speed will increase when loops are involved.

Notice that if you try to execute `euclidean_dist_mex` with a vector `points` of a size different than the one specified, you will get an error.

```
points = rand(3, 5);
euclidean_dist_mex(x, points)            % error
```

Next, we modify the `MEX` file to accept an input that can vary in size.

3.2 Variable-Size Inputs

To have the `MEX` function accept inputs of varied sizes, we use the function `coder.typeof` to specify the size of the inputs. The function `coder.typeof` takes three inputs:

1. An example for the input (like `x`);
2. A vector that specifies an upper bound for each dimension of the input;
3. A vector that specified whether each dimension of the input is variable in size or fixed in size.

Suppose we want to allow `x` to have up to 10 rows, but always one column. And we want `points` to have up to 10 rows, and up to 10 columns.

```
x = [1; 2; 3];
example_x = coder.typeof(x, [10 1], [1 0]);
% [10 1] specifies the upper bound for the dimensions of x
% [1 0] specifies that the size of the first dimension is
   variable,
% while the second is fixed
points = rand(3, 10);
example_points = coder.typeof(points, [10 10], [1 1]);
% [10 10] specifies the upper bound for the dimensions of the
    points
% [1 1] specifies that both dimensions are variable
```

We can now compile the code as before with `codegen`, and use `example_x` and `example_points` as the examples, instead of `x` and `points`:

```
codegen -report euclidean_dist.m -args {example_x,
   example_points}
```

This will overwrite `euclidean_dist_mex` with the new version. We can test it:

```
x = rand(7, 1);
points = rand(7, 3);
euclidean_dist_mex(x, points)
```

We can further modify the code to allow `x` to have as many rows as we want, but always one column. And `points` to have as many rows and columns as we want.

```
example_x = coder.typeof(x, [Inf 1], [1 0]);
example_points = coder.typeof(points, [Inf Inf], [1 1]);
codegen -report euclidean_dist.m -args {example_x,
   example_points}
```

This will overwrite `euclidean_dist_mex` with the new version. We can test it:

```
x = rand(23, 1);
points = rand(23, 40);
euclidean_dist_mex(x, points)
```

3.3 Generating C Code

If you have a supported C compiler installed on your machine, you can also generate the C code from the Matlab files. You do so by adding the option `-config:lib` when calling `codegen`:

```
codegen -d matlab_to_c -report euclidean_dist.m -args {
   example_x, example_points}
```

Notice the use of the option `-d matlab_to_c`, which stores the resulting files in the folder named `matlab_to_c`.

3.4 Observations

The `MEX` files are platform specific, so if you compile a file in Windows, for example, then you will not be able to re-use it in Linux. However, if you have the original `.m` file you can re-compile it again in a different platform.

It is not possible to generate `MEX` files from Matlab scripts, only for functions. If you need to convert a script, then you must write it in a function format.

If you are often converting a file to `MEX`, it is good practice to create a build script that will do the conversion. For example:

```
% generate_mex_euclidean_dist.m
% Generates .mex file for: euclidean_dist.m
% Define inputs
example_x = coder.typeof(x, [Inf 1], [1 0]);
example_points = coder.typeof(points, [Inf Inf], [1 1]);
% Call codegen
```

```
codegen -report euclidean_dist.m -args {example_x,
   example_points}
```

Then, we can execute `generate_mex_euclidean_dist` on Matlab to generate the `MEX` file.

4 Parallelization

We can speed up the execution of code in Matlab by taking advantage of the multiple cores available in modern computers. Matlab offers a relatively easy way to make for-loops run in parallel on multiple cores using `parfor`. To make use of multiple cores, you will need to install the Parallel Computing Toolbox. It makes sense to make a for-loop run in parallel if:

- Each iteration of the loop takes a long time
- The loop iterations are independent from one another

We can use the `parfor` functionality to work with embarrassingly parallel problems. A problem is called embarrassingly parallel if it can be broken into smaller pieces that have little to no dependency between one another.

The syntax for `parfor` is similar to the syntax of for-loops:

```
parfor i = 1:10
    % code to be executed at each iteration
    disp(i)
end
```

However, the loop is executed in parallel and the order of execution is not deterministic. Additionally, the code executed within the loop must not depend on previous iterations of the loop.

If you execute the code above, Matlab will start a pool of workers and then execute the loop in parallel. The pool of workers takes some time when it is first initialized, but can be used afterwards without slow downs. The workers are associated to the cores in the computer's processor. The more cores available, the more workers can be added to the pool. It is possible to set the number of workers to be used in the settings panel for the Parallel Computing Toolbox.

The code above displays the value of the variable `i` on each iteration. Notice that the values are displayed out of order. This happens because the iterations are executed in a non-deterministic fashion.

4.1 Converting a For-Loop into a Parfor-Loop

As an example, let's convert a `for-loop` that bootstraps the confidence interval for the mean into a `parfor-loop`. First, we begin with the data generation and bootstrapping in an usual for-loop:

```
% true mean
mu = 10.35;
% generate random sample
sample = normrnd(mu, 2, 1000, 1);
```

```
% bootstrap
total_iter = 100;
means = zeros(total_iter, 1);
for i = 1:total_iter
    % create bootstrap sample
    b_sample = sample(randi(length(sample), length(sample),
       1));
    % compute statistic
    means(i) = mean(b_sample);
end
% display 99% confidence interval for the mean
CI = quantile(means, [0.005 0.995]);
sprintf('99%% Confidence Interval for Mean: [%.4f, %.4f]', CI
   )
```

Let's make the code above into a function, so that we can specify the sample size and the number of bootstrap samples.

```
% bootstrap_mean_CI.m
function CI = bootstrap_mean_CI(varargin)
parser = inputParser;
addOptional(parser, 'sample_size', 1000);
addOptional(parser, 'bootstrap_samples', 1000);
parse(parser, varargin{:});
sample_size = parser.Results.sample_size;
bootstrap_samples = parser.Results.bootstrap_samples;
% true mean
mu = 10.35;
% generate random sample
sample = normrnd(mu, 2, sample_size, 1);
% bootstrap loop
means = zeros(bootstrap_samples, 1);
for i = 1:bootstrap_samples
    % create bootstrap sample
    b_sample = sample(randi(length(sample), length(sample),
       1));
    % compute statistic
    means(i) = mean(b_sample);
end
% display 99% confidence interval for the mean
CI = quantile(means, [0.005 0.995]);
end
```

Time the function:

```
sprintf('Elapsed Time: %.4f seconds', timeit(@
   bootstrap_mean_CI))
% increase sample size
fun = @() bootstrap_mean_CI(10000);
sprintf('Elapsed Time: %.4f seconds', timeit(fun))
```

```
% increase number of bootstrap samples
fun = @() bootstrap_mean_CI(10000, 5000);
sprintf('Elapsed Time: %.4f seconds', timeit(fun))
```

When we increase the original sample size, the computation of the mean statistic starts taking more time. When we increase the number of bootstrap samples, we increase the number of iterations in the bootstrap loop.

Observe that the iterations in the bootstrap loop are independent from one another. Also, each iteration might take a while, specially if the sample size is big. Therefore, we have an embarrassingly parallel problem. We can use `parfor` to speed up this computation. In this case, all we need to do is change the for-loop into a parfor-loop:

```
% bootstrap_mean_CI_parallel.m
function CI = bootstrap_mean_CI_parallel(varargin)
% change only the line
for i = 1:bootstrap_samples
% into
parfor i = 1:bootstrap_samples
```

We can execute the code once to get the pool of workers started. Then, we time the execution and compare to the non-parallel version.

```
sprintf('Non-Parallel: %.4f seconds\nParallel: %.4f seconds',
    ...
        timeit(@bootstrap_mean_CI), timeit(@
           bootstrap_mean_CI_parallel))
% increase sample size
fun = @() bootstrap_mean_CI(10000);
pfun = @() bootstrap_mean_CI_parallel(10000);
sprintf('Non-Parallel: %.4f seconds\nParallel: %.4f seconds',
    ...
        timeit(fun), timeit(pfun))
% increase number of bootstrap samples
fun = @() bootstrap_mean_CI(10000, 5000);
pfun = @() bootstrap_mean_CI_parallel(10000, 5000);
sprintf('Non-Parallel: %.4f seconds\nParallel: %.4f seconds',
    ...
        timeit(fun), timeit(pfun))
```

Notice that there is no improvement in speed for the cases where the sample size is small and the number of bootstrap samples is small. This can happen because there is some overhead involved in making a for-loop run in parallel. Indeed, each worker must have access to the data being re-sampled. If copying this data over takes longer than the computations in the loop, then the parallel loop will run slower. When the sample size increases, the computation in the loop starts becoming slower than copying the data to the worker, and running the loop in parallel starts becoming quicker. When the number of bootstrap samples increases (number of iterations in the loop), the speed gain from the parfor-loop becomes relevant.

4.2 Requirements for Parfor-Loop

There are a few requirements to use `parfor`. First, the looping variable must take integer values, and the values must be consecutive and increasing:

```
% does not work: non-consecutive integers
parfor i = -5:2:5
    disp(i)
end
% does not work: not integers
parfor i = 0:0.2:1
    disp(i)
end
% does not work: not increasing
parfor i = 10:-1:1
    disp(i)
end
% works: consecutive and increasing integers
parfor i = -5:5
    disp(i)
end
```

Second, when Matlab encounters a `parfor`, it classifies all of the variables inside the loop in 5 different categories. The categories describe what each variable is doing inside the loop. The categories are:

- Loop variables: loop indices in the loop;
- Sliced variables: arrays that are sliced inside the loop and used by different iterations;
- Broadcast variables: variables defined outside the loop and used inside it, but that are not assigned inside the loop;
- Reduction variables: variables that accumulate values across iterations of the loop
- Temporary variables: variables created inside the loop and not used outside the loop

If a variable cannot be uniquely classified into one of the five categories, then Matlab will give an error. For example:

```
x = [1; 2];
out = zeros(10, 1);
parfor i = 1:10
    x(1) = 5;
    x(2) = 10;
    out(i) = sum(x);
end
```

In this case, it is not clear what is the role of the variable `x`. It is acting like a sliced variable, but at the same time as a temporary variable. The code above will not run and Matlab will display an error. We can fix it, by making `x` a temporary variable:

```
out = zeros(10, 1);
parfor i = 1:10
    x = [5 10];
    out(i) = sum(x);
end
```

Now, `x` is being created inside the loop and it is clear that different iterations of the loop do not depend on one another.

Third, parfor-loops cannot be nested:

```
% does not work
parfor i = 1:5
    sprintf('i=%d', i)
    parfor j = 1:3
        disp(j)
    end
end
% works
parfor i = 1:5
    sprintf('i=%d', i)
    for j = 1:3
        disp(j)
    end
end
```

Nesting `parfor` does not work because we are already using all the workers (cores) in the outermost `parfor`, so the loop inside should be treated as a regular loop. For a complete set of issues that may arise when using `parfor`, refer to the reference page for variables in parfor-loops.

4.3 Memory Management

The `parfor` command can lead to memory issues. Each worker gets a copy of the variables in the workspace, so if you are using 2 GB of memory in your workspace and have a parfor-loop with 4 workers, then Matlab will require access to 8 GB of memory. Therefore, if you are working with big matrices, or your computer has a big number of cores, then you need to be mindful of the memory being used.

5 Assignment

Problem 1 *Consider a deterministic growth model, where an agent decides between consumption (c_t) and investment in capital (k_t), while maximizing his utility. We can write this problem as:*

$$\max \sum_{t=0}^{\infty} \beta^t U(c_t)$$

$$\textit{subject to} \begin{cases} k_{t+1} = k_t^{\alpha} - c_t + (1-\delta)k_t, \forall t >= 0 \\ k_0 > 0 \end{cases}$$

Write the problem as a Bellman equation. Let $U(c;\sigma) = \frac{c^{1-\sigma}-1}{1-\sigma}$. *Obtain the Euler equation for this problem in terms of the consumption* c. *Solve the problem by Value Function Iteration. Consider* $\sigma = 2$, $\beta = 0.95$, $\delta = 0.1$ *and* $\alpha = 0.33$. *Use the steady state value of* k *to create a grid for the possible values of* k, *say* 100 *points between* $0.25k^*$ *and* $1.75k^*$. *Start with a guess for* V *over the grid, for example* $V(k) = 0$ *for all* k *in the grid. Use the Matlab minimization function to solve for* k. *You may want to add the constraint that* c *should always be positive.*

Use **`parfor`** *to speed up the solution of this problem. What can be easily run in parallel in this problem? Is this problem embarrassingly parallel? If not, are there subsets of the problem that are embarrassingly parallel?*

Graph the time it takes to find the solution of the problem when the number of grid points increases. Compare the time using regular loops and using **`parfor`** *loops.*