Notes on Algorithm Efficiency When Coding in Matlab

The purpose of this document is to discuss the most common mistakes we make when coding in Matlab, and how to fix them. These mistakes are not logical, but are mistakes in terms of loss of efficiency.

Imagine we have prices of a stock stored in the vector P:

$$P = \begin{bmatrix} P_1 \\ P_2 \\ \vdots \\ P_n \end{bmatrix}$$

And we want to compute its geometric returns in percentage:

$$R_i = 100 * \log\left(\frac{P_i}{P_{i-1}}\right)$$

A first try would be:

Script/Function 1: Inefficient Return Calculation

```
R = 0;
for i = 2:n
    R(i) = 100*[log(P(i)/P(i-1))];
end
```

The major problem here is that we have a vector of prices, and so we will have a vector of returns, but the variable R was not initialized as a vector, but rather as an integer. When Matlab encounters the line $R(i)$ it will have to perform a type change (from integer to vector of numbers). Matlab also does not know how big the vector should be, so in each iteration of the for loop it will have to increase the size of the vector and reallocate memory to it.

To fix these problems we declare R as a vector, and compute how big it should be: if P has n observations, then R will have $n-1$ returns. So we get the following code:

Script/Function 2: Better Return Calculation

```
R = zeros(n-1,1);
for i = 2:n
    R(i) = 100*[log(P(i)/P(i-1))];
end
```

Now the code is more efficient and runs faster. A good way to check this is to create a vector P of random numbers and time both scripts. Try varying the size of P (by changing n) and see what happens to the runtime.

Matlab has many built-in functions that were written in lower level languages. These built-in functions are very fast, and are even faster than writing the equivalent code in Matlab. Thus, to achieve maximum efficiency we should make use of Matlab's built-in functions.

For example, calculating the geometric return requires calculating the log of a fraction, which is the same as calculating the difference of the logs. Matlab has a built-in function that calculates differences, and we can use it to speed up the script:

Script/Function 3: Efficient Return Calculation

```
R = 100*diff(log(P));
```

In this case, the code not only runs faster, but also looks cleaner. Again, you should time these three scripts in Matlab and see which one runs faster, and how the speed varies with the size of P.

It is important to notice that usually there is a trade-off between efficiency and transparency. The more efficient a code becomes, the less transparent it is for others (other people that have to read your code, and you in the future when you have to re-use your code and have forgot everything about it). So, try to keep that in mind when coding a more complex problem.

Now, let's explore another coding issue: repeated creation of vectors inside a for loop. To do so, imagine we want to compute the realized variance:

$$RV_i = \sum_{j=1}^{m} R^2_{(i-1)*m+j}$$

where m represents the number of returns we observe in a day, and $i = 1, \ldots, T$ represents the day for which we are computing the realized variance.

A first try would be:

Script/Function 4: Inefficient RV Calculation

```
1 T = length(R)/m;
2 RV = zeros(T,1);
3
4 for i = 1:T
5     Rsquared = R.^2;
6     id = (i-1)*m+1:1:i*m;
7     Rid = Rsquared(id);
8     RV(i) = sum(Rid);
9 end
```

In the second line of the code we correctly initialize the vector RV of realized variances. Then, in line 5 inside the for loop, we get the vector of returns and square each value, and save them in the vector *Rsquared*. We then select the returns of the correct day in lines 6 and 7, and use these returns in line 8 to get the realized variance.

Can you guess what is the problem in this script? What makes it run slow? The problem is that we are creating a new vector (that is big) and then destroying it in each iteration of the for-loop. In line 5 we square each value of the vector R and copy them to *Rsquared*, we use these values, and then in the next iteration of the loop we do it all again.

This is a very common problem when working with vectors, and sometimes can be very subtle and hard to identify. When dealing with more complex coding problems it is very easy to run into this issue and end up with a code that takes a long time to finish running. It is important to remember to be careful with your code whenever you need to access a vector repeatedly.

To fix the code, we can just move the vector *Rsquared* outside the for-loop, since it only needs to be created once:

Script/Function 5: RV Calculation

```
1 T = length(R)/m;
2 RV = zeros(T,1);
3
4 Rsquared = R.^2;
5 for i = 1:T
6     id = (i-1)*m+1:1:i*m;
7     RV(i) = sum(Rsquared(id));
```

```
end
```

Now the code is much faster than before. You should not take my word for it, test both scripts and see which one is faster.

An alternative way to code it is to avoid creating the vector *Rsquared* at all. We just use what we need from R directly:

Script/Function 6: RV Calculation

```
T = length(R)/m;
RV = zeros(T,1);

for i = 1:T
    id = (i-1)*m+1:1:i*m;
    RV(i) = sum(R(id).^2);
end
```

There is not a big difference in efficiency between the two previous scripts, but it could be argued that the latter is more transparent.

Remember the trade-off between efficiency and transparency? We can make the previous script even more efficient by accessing the vector R in a faster way:

Script/Function 7: Efficient But Hard to Read RV Calculation

```
T = length(R)/m;
RV = zeros(T,1);

for i = 1:T
    Rsquared = 0;
    for j = 1:m
        Rsquared = Rsquared + R((i-1)*m+j)^2;
    end
    RV(i) = Rsquared;
end
```

However, it adds some extra lines to the code and makes it less readable. So, in this case, it is better to loose a little bit of efficiency to gain readability by using the previous script instead.

Conclusion

Coding in Matlab is a very practical way to solve the problems we will encounter in this course. Nevertheless, we should be aware of the common pitfalls that slow down our code. Avoiding these pitfalls becomes critical as we move to more and more complex problems. Important things to remember:

- Declare variables of the correct type;
- When declaring vectors, specify their correct size;
- Use efficient built-in functions;
- Be careful to not create and destroy big vectors inside a for-loop.