Ex. 7, Vectorization, timing and Debugging

Stefan Karlsson

March 28, 2014

1 Vectorization

Consider the following simple function(make a function in your matlab environment):

```
function A = sumA1(A)
  A = A + 1;
```

The input `A` can be anything, as long as it is defined what happens when we add the value 1. Lets consider vectors, so that `A` will be a long vector of scalar values. In that case, an equivalent function to `sumA1`, is `sumA2`:

```
function A = sumA2(A)
  for ii = 1:length(A)
    A(ii) = A(ii) + 1;
  end
```

The two functions `sumA1` and `sumA2` are very different in appearance, yet the final output will be identical. Lets test this, by making a large random vector that we send to the functions:

```
A = rand(10^6,1);

%call the first function, time it
tic;
A1 = sumA1(A);
toc;

%call the second function, time it
tic;
A2 = sumA2(A);
toc;

bCheckForSimilarity = isequal(A1, A2)
```

On my system, the first function executes in approximately 2.2 ms and the second function takes 57 ms (roughly 26 times slower)[1]. We find that `sumA1` has less code, is more readable with much faster run time.

[1]On earlier versions of Matlab, this could be thousand times slower.

Why this huge difference in run-times?

The answer lies with how Matlab can make use of effecient algorithms, that are mapped onto the parallelism of the hardware. The process of converting code from `sumA2` into the form in `sumA1` is called vectorization. Vectorization is very important in high performance computing, and there are many languages aside from Matlab that allows for it natively[2]. We have seen some nice tricks with indexation of vectors and matrices, both in assignment and calculations. These tricks all fall within the scope of vectorized implementations. In Matlab, the vectorization is especially elegant. When it comes to vectorization in Matlab, it is mostly a question of how we remove loops, and re-write the same operations on fewer lines and more readable code. We should usually vectorize if and when we can. There are some drawbacks for some problems, however, and we will investigate some of those here.

2 Sums

Lets investigate an example of the sum:

$$S_P = \sum_{n=1}^{P} \frac{1}{n^2} = 1 + \frac{1}{2^2} + ... + \frac{1}{P^2}$$

In the limit, we have that $S_\infty = \frac{\pi^2}{6}$, but we are interested in the behavior as we input higher P. First, we do it by a loop:

```
function s = sumP1(P)
s = 0;
for n = 1:P
  s = s+1/(n^2);
end
```

The same sum is easily implemented in a vectorized fashion:

```
function s = sumP2(P)
n = 1:P;
s = sum( 1./(n.^2) );
```

Now, for some timing:

```
tic;sumP1(10^6);toc

%you may want to run sumP2 twice, the first time will likely be slower.
tic;sumP2(10^6);toc
```

The difference between these two functions is not that big in newer versions of Matlab. I get 19 ms for `sumP1` and 8 ms for `sumP2`. This would indicate that the vectorized version is more than twice as fast, but there is more to the story.

[2]I.e. its part of the core language

For the vectorized version sumP2, we create a big vector n = 1:P, the size is of course dependent on P (was 10^6 in example above). It takes time for Matlab to allocate memory and set the vector, and the loop version of the sum does not require this over-head. While the vectorized version may be faster, especially for smaller Ps, there is a problem we encounter for large P. Try this(warning, Matlab will run out of memory):

```
sumP2(10^11)
```

This makes Matlab scream in pain, and wont be able to finish the job. The equivalent call:

```
sumP1(10^11)
```

will not crash, however it takes a very long time to finish.

2.1 When to use loops?

Matlab has evolved quite a bit over the last 10 years. One thing that is much improved is the way that native for-loops are handled. Sometimes, you should use loops, lets investigate two common cases:

2.1.1 Memory considerations

Vectorising usually means a gain in speed, but sometimes a loss in memory(as above example sumP2 illustrates).

2.1.2 Dependency on previous iterations

An example of this would be so called "recursive filters"[3]. The following is an example, it's a simple way to estimate the variation locally in a signal x. It is not possible to vectorize, because it has dependency on the previous iteration:

```
N = 150;
x = randn(N,1);
y = zeros(N,1);

%y(t-1) is from previous iteration, gives the new value at t:
for t = 2:N
  y(t) = 0.8*y(t-1) + 0.2*x(t)^2;
end

plot(y);hold on; plot(x,'r')
```

2.1.3 Access intermediate results

Explicit loops can access results on every iteration. As a simple example, the following function cannot be vectorized (assuming that we want the output in the correct sequence)

[3]linear filters with infinite impulse response

```
for n = 1:P
  s = s+1/(n^2);
disp(s);           %echo the value of the sum, on this iteration
end
```

3 Debugging

Everybody makes errors during coding. It could be simple syntactic errors or more complicated ones such as logical errors. Matlab is equipped with a debugger to help the programmer to find and fix errors.

Errors come in two major forms: compile-time and run-time. Compile time errors will be noted by Matlab just before your functions run, and are usually easy to find. Among compile-time errors are common mistakes such as forgotten parenthesis or operators, and Matlab not only tells you there is an error, but nicely gives you the file, the line and often suggestion on how to fix it. You also get dynamic checking from matlab(like a spell checker) as you are writing the code, giving you hints of potential errors.

Run-time errors are the worse kind, they are bugs that have infected your code due to misses either in syntax(typos) or worse: your logic and understanding. I have touched on some common mistakes that cause problematic run-time errors. Here are is some valuable advice:

- **Avoid using the global workspace!**. The power to change variables anywhere in code, in any file, will make finding run-time errors extremely difficult.
- **Be careful with y-x indexation order.** Matlab indexes matrices as `A(y,x)`, not as `A(x,y)` as we are used to.
- **Be careful with one-based indexation.** The first element of a vector is indexed as `a(1)`, not as `a(0)`. Most languages start indexation at 0.
- **Keep track of element-wise operations, vs matrix operations**. The forgotten dot! If you wish to multiply two matrices `A` and `B` you write `A*B`, if you wish to multiply them ***element-wise*** you write `A.*B`. If both matrices are square, then both operations are allowed and you get hard to find bugs in your code.

Now let us move on and see how we can use the debugger[4].

[4]You do not need to view this video in order to complete the tasks, but it will help you to finish all the coming tasks more easily

Tasks

Solutions that do not fill the following requirements **EXACTLY, PRECISELY AND TO THE LETTER** will not be considered:

- Send your solutions by email to:

  stefan.karlsson@hh.se

  subject: Matlab, Exercise `X`, `YourNames`

- Send all the files that are requested, no more and no less, in one single zip file per exercise, with NO sub-folders in the zip file. All the files, for all the tasks should be bundled into one zip file.

- Put the Names of the authors, in remarks, at the top of every m-file.

- Send the solutions within 2 weeks of every exercise session. That is, you have a two week deadline to hand it in.

- You will get 2 chances to send it in to me correctly.

Task 1

We will make an experiment with dices. We will throw two dices `P` times and decide how often they add up to 6.

The function `e7_2(P)` emulates throwing 2 dices `P` times, and should produce a plot which shows the percentage of throws that sum to six plotted versus number of throws. (x −axis: the number of throws, y-axis: the percentage of throws that have summed to 6). The x-axis should number from 1, up until `P`.

Add a horizontal line to the plot where one sees the probability of this Experiment (what probablitiy is there of getting the sum to 6?). For a very large number of throws this experiment should validate the true probability.

Hand in ***ONLY*** the file `e7_1.m`

Hints, Task 1

- First thing to do is to calculate[5] by ***hand*** the true probability of 2 dices summing up to 6. This way, you can just put this value as the horizontal line in the plot, and be sure that you will see the output of the experiment converging to the right value.

  To calculate that probability, think about how many possible outcomes there are, when you throw two dices. Then think about how many of those total possibilities sum to 6. This gives you two values of interest (total outcomes, total positive outcomes). At this point you know what to do.

- Implement this function most easily by first using `randi` (search documentation).

[5]No need to hand in any calculation

Task 2

Write an m-file `[s1,s2] = e7_2(A,B)` that takes square matrices `A` and `B` of equal size and calculates the sum:

$$\sum\sum(A(i,j) - B(j,i))^2$$

!!! NOTICE !!! the elements of the matrices are indexed differently over the sum!!!
(i and j are iterated over the valid positions in the matrices only) The file should implement this both in a vectorized way, and a regular sequential way by for loops. Implement 2 local functions `e7_2a` and `e7_2b` to do the loop version and the vectorized version respectively. Let the function `e7_2(A,B)` have the following form:

```
function [s1,s2] = e7_2(A,B)

if nargin < 2
  error('require 2 args');
end

if ~isequal(size(A),size(B))
  error('A and B must be equal size')
end

if (size(A,1)~=size(A,2))  ||  (size(B,1)~=size(B,2))
  error('A and B must be both be square matrices')
end

s1 = e7_2a(A,B);
s2 = e7_2b(A,B);
if s1 ~= s2
  error('Student hasnt implemented the sum correctly')
end

%%  Define the local functions:
function s = e7_2a(A,B)
%implement the sum by nested for loop

function s = e7_2b(A,B)
%implement the sum in a vectorized way (no loops)
```

Hints, Task 2

For the vectorized method, consider what it means to do a matrix transpose. What kind of sum will you get for symmetric matrices?

Task 3 (optional, for grade 4)

Function `e7_3.m` should measure the time it takes to execute the two approaches of task 2. It should generate different sizes of random matrices `A` and `B` (still they should be square and of equal size), where height and width is varied $\in [1, 300]$. The first `A` and `B` matrices will be of size 1×1, the second 2×2, then 3×3, 4×4, ... , 300×300.
For each size of the matrices (totally 300 different ones) the sums of task 2 should be averaged over 10 times for each of the methods `e7_2a` and `e7_2b`. The time it takes should be saved in 2 vectors, giving two vectors in total as a result. These vectors contain the time each algortihm takes to execute a particular matrix size. Plot both vectors, make the vectorized version plotted in blue, the loop version in red, and put a nice legend in the figure.

Hints, Task 3

There are alternative ways of using `tic` and `toc`. Get the time out in seconds from the toc function.