‘Stop-watching’ in MATLAB

Rowin Appanah
Gareth Price

Presentation, cheat sheet and example code can be downloaded from:
http://www2.warwick.ac.uk/fac/sci/moac/people/students/2012/gareth_price/presentations
Overview

• A simple introduction on the need to measure performance and for optimization of scripts.

• What is profiling

• Why should we profile?

• Matlab-specific:
  (1.) tic/toc.
  (2.) cputime
  (3.) Profiler.
Program optimisation is (intuitively) desirable as it enables economy of space, resources and time.

Various levels can be targeted for optimization, including design level, source code level and run time.

Thus, we require 'stop-watches' which would enable us to measure the performance of whole programs and single lines of codes.

Herein, we introduce MATLAB-specific stop-watches.

---

**Measuring Performance**

Desirable

Not Desirable
Tic/toc functions: A simple way to measure execution time

• This dual function effectively measures the performance of programs or calculations.

Syntax:

```
tic;
    code of interest;
toc
```
format long;
REPS = 1000;    minTime = Inf;    nsum = 10;

tic;

for i=1:REPS
    tStart = tic;    total = 0;
    for j=1:nsum,
        total = total + besselj(j,REPS);
    end
    tElapsedTime(i) = toc(tStart);
end

minTime = min(tElapsedTime(i), minTime),

end
tElapsedTime

averageTime = toc/REPS
Tic/toc functions

```matlab
tic;
for i = 1:50000
    a(i) = sin(i);
end
toc
```

```matlab
tic;
i = [1, 50000];
a = sin(i);
toc
```

Command Window

```
>> ticocwithforloops
Elapsed time is 0.005508 seconds.
```

```
>> ticocwitharray
Elapsed time is 0.000021 seconds.
```
Tic and toc functions

• The manipulation of arrays is faster than regular manipulation with for-loops.
  (Why? Explained later)

• But, it may be impossible to forgo for-loops.
CPU Time.

- CPU time (syntax: `cputime`).

- Returns total CPU time (in seconds) used since current initialization of Matlab.

- `t=cputime; surf(peaks(60)); r= cputime-t;`

- `r= 0.1200 (seconds)`
Profiler

• Tic/toc is amenable to simple programs where the output is limited to time of execution.

• The profiler is a more comprehensive family of tools that give us considerably more information.
Profiling: what is it and why?

- Profiling measures where a program/script spends time – i.e. how long each line took to execute
- It is important because MATLAB is intrinsically slower than native C, C++ or even Fortran (an old language from the 1950s)
  - and when analysing large datasets this really makes a difference
- Profiling helps to uncover performance problems by:
  - Avoiding unnecessary (re-)computation
  - Identifying bottlenecks
  - Changing resource-costly functions for “cheaper” ones
Running Profiler using the GUI

Method 1. Click start, then Profiler

Method 2. Click Desktop, then Profiler

After Method 1 or 2 click Start Profiling
Running Profiler in code

profile clear  % clears the viewer contents
profile on     % turns profiler on
doFunction()   % or script
profile off    % turn it off
profile viewer % view results
The Results

• Which functions called that the function and how many times

• The individual lines where the most time was spent, including the number of times that line was executed and how much time was spent on that line

• What other profiled functions were called by that function

• A coverage summary showing the number of lines run vs. not run

• A color-coded version of your code, showing potential problem spots.

• However, not all inbuilt functions are profiled (but most are)
```matlab
function [x,y,z] = awhile
    for i=100:100:300
        x = rand(i);
        y = rand(i);
        z = calculate(x,y);
    end
end

function z = calculate(x,y)
    z = conv2(x,y);
    pause(2);
end
```

![Profiler output](image)
Example: Vectorising

• Vectorising your code is *far* faster than using a *for* loop.
  – For loops are not fully compiled into assembly as in C, C++, Fortran etc.
  – Matrix/vector operations *are*, and are thus much faster

• We can analyse the difference using the profiler – Example time!
Sample code (example3.m)

nValues = 1e6;
firstArray = ones(1,nValues);
secondArray = 2*ones(1,nValues);
resultArray = zeros(1, nValues);

for i=1:nValues
    resultArray(i) = firstArray(i) * exp(secondArray(i));
end

resultArray = firstArray .* exp(secondArray);
## Results

**Profile Summary**

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time*</th>
<th>Total Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>example3</td>
<td>1</td>
<td>7.465 s</td>
<td>7.465 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc</td>
<td>4</td>
<td>0.124 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;getStatObjects</td>
<td>2</td>
<td>0.075 s</td>
<td>0.050 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;getShortValueObject</td>
<td>6</td>
<td>0.050 s</td>
<td>0.050 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;getShortValueObjects</td>
<td>1</td>
<td>0.050 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;getStatObject</td>
<td>6</td>
<td>0.025 s</td>
<td>0.025 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;getWhosInformation</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>...s.mlwidgets.workspace.WhosInformation (Java method)</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;getAbstractValueSummary</td>
<td>4</td>
<td>0 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;getClass</td>
<td>4</td>
<td>0 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>workspacefunc&gt;num2complex</td>
<td>12</td>
<td>0 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>codetools/private/dataviewerhelper</td>
<td>8</td>
<td>0 s</td>
<td>0.000 s</td>
<td></td>
</tr>
<tr>
<td>...dataviewerhelper&gt;upconvertIntegralType</td>
<td>1</td>
<td>0 s</td>
<td>0.000 s</td>
<td></td>
</tr>
</tbody>
</table>
### Lines where the most time was spent

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Code</th>
<th>Calls</th>
<th>Total Time</th>
<th>% Time</th>
<th>Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>resultArray(i) = firstArray(i)...</td>
<td>1000000</td>
<td>4.035 s</td>
<td>53.5%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>end</td>
<td>1000000</td>
<td>3.432 s</td>
<td>45.5%</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>resultArray = firstArray .* ex...</td>
<td>1</td>
<td>0.046 s</td>
<td>0.6%</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>for i=1:nValues</td>
<td>1</td>
<td>0 s</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>resultArray = zeros(1, nValues...)</td>
<td>1</td>
<td>0 s</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>All other lines</td>
<td></td>
<td></td>
<td>0.023 s</td>
<td>0.3%</td>
<td></td>
</tr>
<tr>
<td>Totals</td>
<td></td>
<td></td>
<td>7.537 s</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>
- 7.47 seconds vs. 0.05 seconds
- Also look at the number of calls!
If a loop is necessary

- You can code loops in C in .mex files
- These are much quicker as the instructions within the loop do not have to be interpreted and compiled each time
- However, this does require C knowledge!
  - remember you can see how MATLAB implements functions by typing edit FUNCTION in the command window
Pre-allocate memory

• Although matrices, vectors and arrays can be made and extended dynamically in a for loop, it is much faster to pre-allocate memory before the loop by:
  – zeros(100,100)
  – ones(100,100)

• this is because arrays are stored in memory in a contiguous block
  – when they are increased sequentially, compiler has to look for a contiguous block of memory that is big enough for the whole array
  – it can also lead to fragmentation of memory
tic
x = 0;
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
toc

tic
x = zeros(1, 1000000);
for k = 2:1000000
    x(k) = x(k-1) + 5;
end
toc

Elapsed time without pre-allocation is **0.198532** seconds.  
Elapsed time with pre-allocation is **0.009586** seconds.

There’s nothing wrong with tic/toc in simple cases, remember! But the point could also be proven to be true through the profiler.
Array indices

```matlab
someArray = ones(1000, 1000);
someVector = ones(1000,1000);
nLoops = 1e6;

for i=1:nLoops
    someVector = someArray(1,:);
end

for i=1:nLoops
    someVector = someArray(:,1);
end
```

generateู example5.m
• 2D arrays (i.e. matrices and vectors) are sequential 1D arrays
• Thus it is quicker to access a consecutive sequence of elements

\[
\begin{align*}
\text{someArray}(::,1) & : 1 & 2 & 3 & 4 & 1 & 2 & 3 & 4 \\
\text{someArray}(1,:) & : 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4
\end{align*}
\]
Functions vs. Scripts

• Functions are much quicker than scripts as functions are loaded into memory in their entirety and compiled all at once
• Scripts are loaded into memory line-by-line and executed individually
Use the correct functions

• Often a function can be implemented multiple ways
• For example:
  – random(‘gamma’, 2, 2);
  – gamrnd(2, 2);
• gamrnd is 4x quicker – random(‘gamma’,x,x) is a wrapper for gamrnd and introduces overhead
Word of Warning

- Remember, often code readability is as important as execution speed
  - Especially if collaborating in a group!
References


Mathworks.co.uk.