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

Profiling in MATLAB: Optimising your code

By Rowin Appanah and Gareth Price

(full proofs shown in presentation and example code)

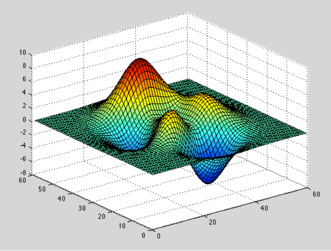
What is Profiling?

- *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.
  - … 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" one.

Simple Stop-watchers: tic/toc and cputime

- This dual function effectively measures the performance of programs or calculations by keeping track of the execution time.
- Its syntax is simply:

```
tic;
    for i = 50000,
            a(i) = sin(i); // your code of interest
    end
toc
OUTPUT: Elapsed time is 0.023058 seconds.
```

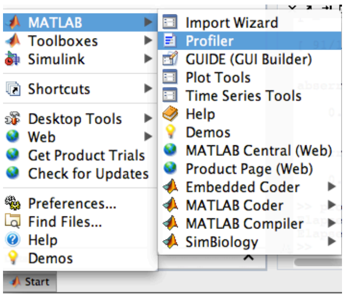


- The total CPU time used by the MATLAB script can be found by calling the `cputime` function:

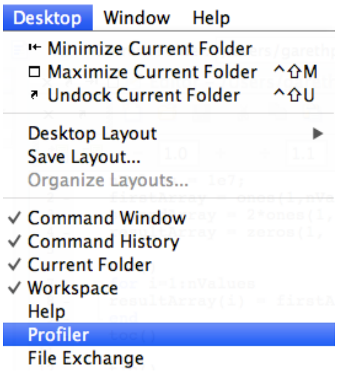
```
T = cputime;
    Surf(peaks(60));
R = cputime - T
OUTPUT: 0.1200 seconds.
```

The Profiler

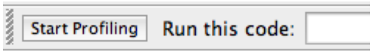
- Tic/toc is amenable to simple programs where the only output is limited to time of execution.
- The profiler is a much more comprehensive family of tools that give us considerably more information.
- It can be started using the GUI (graphical user interface) by:



or



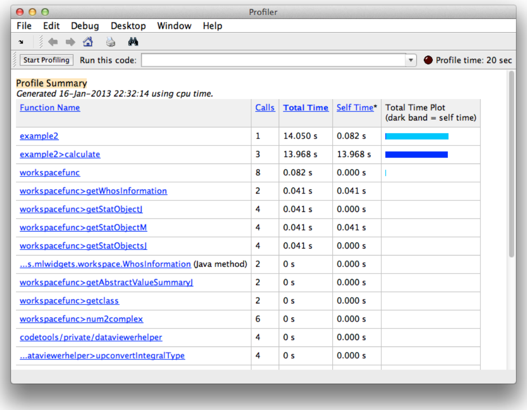
…and then Start Profiling



- It is simple to use in code, too:

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

- It outputs results, including:
  - 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)



Profile time: 20 sec

Profile Summary

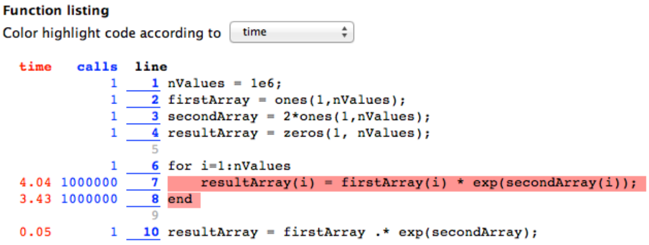
Generated 16-Jan-2013 22:32:14 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
example2	1	14.050 s	0.082 s	
example2>calculate	3	13.968 s	13.968 s	
workspacefunc	8	0.082 s	0.000 s	
workspacefunc>getWhosInformation	2	0.041 s	0.041 s	
workspacefunc>getStatObjectJ	4	0.041 s	0.000 s	
workspacefunc>getStatObjectM	4	0.041 s	0.041 s	
workspacefunc>getStatObjectsJ	4	0.041 s	0.000 s	
...s.mlwidgets.workspace.WhosInformation (Java method)	2	0 s	0.000 s	
workspacefunc>getAbstractValueSummaryJ	2	0 s	0.000 s	
workspacefunc>getclass	2	0 s	0.000 s	
workspacefunc>num2complex	6	0 s	0.000 s	
codetools/private/dataviewerhelper	4	0 s	0.000 s	
...ataviewerhelper>upconvertIntegralType	4	0 s	0.000 s	

- The light blue bar shows the time in another function, dark blue is time in itself. More interesting results are shown when the function name is clicked.

General Tips: from the Profiler

- **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
  - 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



Function listing

Color highlight code according to time

```
time    calls  line
            1     1 nValues = 1e6;
            1     2 firstArray = ones(1,nValues);
            1     3 secondArray = 2*ones(1,nValues);
            1     4 resultArray = zeros(1, nValues);
                  5
            1     6 for i=1:nValues
4.04  1000000     7     resultArray(i) = firstArray(i) * exp(secondArray(i));
3.43  1000000     8 end
                  9
0.05        1    10 resultArray = firstArray .* exp(secondArray);
```

- 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)`

- 2D arrays (i.e. matrices and vectors) are sequential 1D arrays
  - Thus it is quicker to access a **consecutive** sequence of elements (this differs with how you code is set up, but generally `array(:,1)` is quicker than `array(1,:)`

- 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

- Often a function can be implemented multiple ways, for example:
  - `random('gamma', 2, 2);` can also be implemented as: `gamrnd(2, 2);`
  - `gamrnd` is 4x quicker – `random('gamma',x,x)` is a wrapper for `gamrnd` and introduces overhead

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