Development, Debugging and Improving Performance

Overview

- matlab workbench
- help and matlab central
- working with vectors and matrices
- programming structures repetition and condition checking
- data types e.g. cell arrays, structures and handles
- functions and handles
- simple plotting

Outline

- Code Analyzer
- Debugging
- Profiling and Timing
- Vectorisation
- Memory Management and Preallocation
- Algorithm Devlopment

Provide the tools to develop and maintain high quality software for your research. Properly documented to enable researchers to repeat and reuse your research.

The Matlab workbench and editor already provide many features to make programming much easier, just check the code fragment below. Copy and paste this fragment as a script in the matlab editor.

Observe how the matlab editor highlights warnings and errors on the right hand bar clicking on the individual items shows the warning/error and provides suggestions.

```
%% Simple example to show features
%Sample fragment to high light features of the workbench

a=1  %unused variable
```

```
a = 1
```

```
b=[1 2; 3 4]
```

```
b =

     1     2
     3     4
```

-

```
c=[1  4]
```

```
c =

     1     4
```

-

```
%FIXME
x=b*c


%% Another code section
%TODO
%add more code and doo something useful
y=b*b;
```

Directory Reports.

Directory reports may be used as a first step to obtain an overview of particular aspects of files in a given directory (folder). Directory reports are available from a drop-down list in the Current Folder Browser menu. Note that these reports are specific to a given directory.

As a first step, we can produce a Code Analyzer Report listing all Code Analyzer warnings for each file in the current folder. This will help us to fix syntactical and other errors present in the current code.

Recommended activity: run a Code Analyzer Report to generate a summary of all Code Analyzer warnings and errors.

Analyzing Code in the Editor.

The MATLAB Editor has various integrated tools intended for code debugging. The status box in the upper right-hand corner provides an indication of the current status of the code. Green means that there are no detectable errors in the code; orange means that there is potential for unexpected results or poor performance, and red means that there are errors which currently will prevent the code from running.

A summary of the code issues can be obtained from the Code Analyzer Report as described above. This analysis can also be done programmatically using the CHECKCODE function:

```
addpath('../matlab_examples/casestudies/health/');
checkcode('findBestPredictors') %probably need to execute this from matlab workbench!
```

```
L 85 (C 33-40): The function 'getPairs' might be unused.
L 94 (C 9-11): FOR might not be aligned with its matching END (line 98).
L 95 (C 13-20): The variable 'varPairs' appears to change size on every loop iteration. Consider prealld
L 96 (C 13-20): The variable 'varPairs' appears to change size on every loop iteration. Consider prealld
```

```
L 97 (C 15): Terminate statement with semicolon to suppress output (in functions).
L 128 (C 60): Invalid syntax at ';'. Possibly, a ), }, or ] is missing.
```

Using the Debugger

In the examples directory under the ***numericalanalysis/root_bisection***folder you will find a MATLAB script named **plotroot**.

- Make sure to make directory named ***numericalanalysis/root_bisection*** your “Current Directory” via the “***Current Folder***” window or alternatively add this folder to the MATLAB Path via *the context-menu* of the folder named root in the *current-folder window*
- Execute the script root_bisection under the action of the debugger.
- To do this open root_bisection.m in the edito.
- We will inspect the value of x and fx at each step of the while loop to do this click on the dash to the right of the line number 14, a red mark will appear indicating a breakpoint has been set
- Now when you run the script execution will stop at the breakpoint a green arrow indicating the current line to be executed - the editor will open and in the bar at the top there are options to continue, step and quit debugging etc.
- Now go back to the workbench and observe the workspace values. Observe that the command prompt is a K>> indicating debugger mode has been set.
- You can step through each line using the controls in the editor. Breakpoints can be added or removed as you require.

Using the Profiler

•Type profile viewer to start the profiler up and

•In the run this code field enter the name of your script file, or keep that field clear and just click on Start profiling followed by running your task as usual from the Matlab Command Window and when finish click on the stop profiling icon.

•Clicking on the profiling summary icon will display a profile summary which can be expanded by clicking on the routine names on the list.

Measuring Performance: Stopwatch Timer

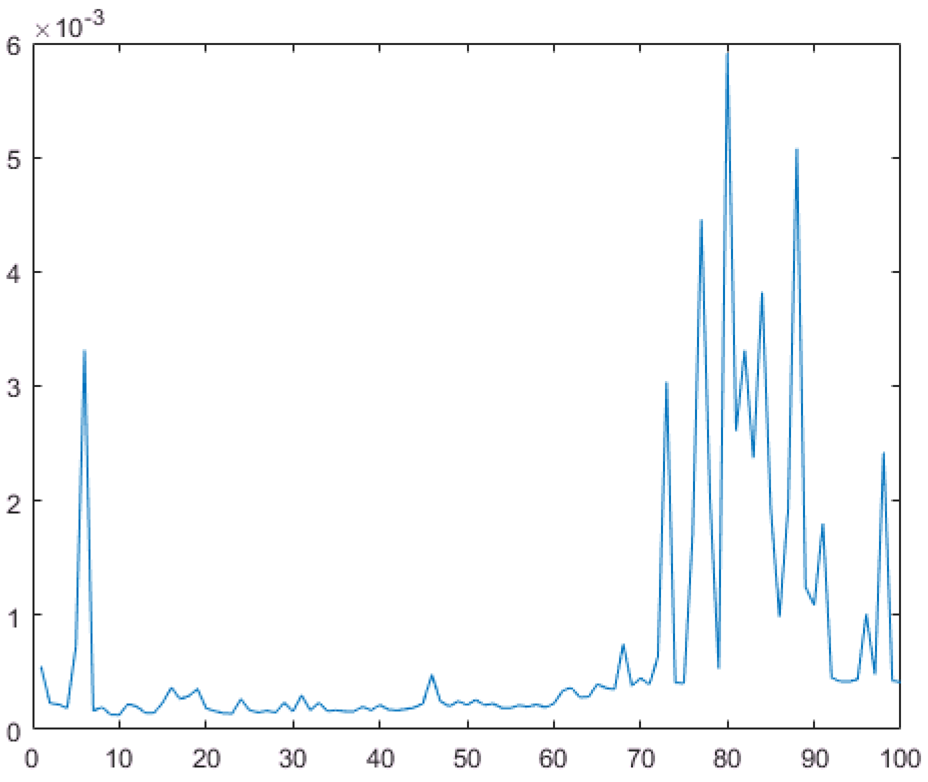
•Start the timer

–tic

•Stop the timer

–toc

```
t = zeros(1,100);
for n = 1:100
    A = rand(n,n);
    b = rand(n,1);
    tic;
    x = A\b;
    t(n) = toc;
end
plot(t)
```



Datatypes and Memory

In order to improve the memory usage and the performance of the MATLAB code, it is important to understand the memory requirements of the MATLAB datatypes. Judiciously choosing the datatype to store your data can increase the efficiency of your code.

You can determine the memory used by a variable by choosing to display the 'Bytes' column in MATLAB workspace.

- The amount of memory required by a datatype increases with the increase in the range or the precision of the values that the datatype can express.
- Container variables (tables, cell arrays, and structure arrays) require overhead in addition to the data they store. This additional memory is used to store information about the contents of the variable. The amount of the overhead depends on the size of the variable.

Special Variable Types

MATLAB has datatypes that are designed to be used with data having specific characteristics. These datatypes can significantly reduce the memory consumption.

Categorical arrays can be used to store a list of text labels or strings containing many repeated entries. Instead of storing each item in the list separately, a categorical array stores only the unique entries and creates references to these entries to represent the elements in the list. The memory savings are large when the list contains many repeated entries.

Memory Anatomy.

On a Windows system, the MEMORY command can be used to obtain available memory information. Calling the MEMORY function with two outputs produces two structures which contain user and system information, respectively.

```
[user, sys] = memory;
disp(user)
```

```
    MaxPossibleArrayBytes: 2.3328e+10
    MemAvailableAllArrays: 2.3328e+10
            MemUsedMATLAB: 1.7902e+09
```

```
disp(sys)
```

```
    VirtualAddressSpace: [1×1 struct]
           SystemMemory: [1×1 struct]
         PhysicalMemory: [1×1 struct]
```

Total System Memory (`sys.SystemMemory.Available`): Physical RAM available + Page File (or swap) available. It depends on the amount of RAM (fast, but expensive) and swap disk (cheap, but slow). To monitor system memory, you can also use the Windows Task Manager.

MATLAB Process Virtual Memory (`sys.VirtualAddressSpace`): Total and available memory associated with the whole MATLAB process. It is limited by processor architecture and operating system.

MATLAB Workspace (`user.MemAvailableAllArrays`) **and Largest block** (`user.MaxPossibleArrayBytes`): The effective workspace available for data is the MATLAB process virtual address space minus system DLLs, Java™ (JVM), MATLAB.exe and DLLs. The largest block (for numerical arrays) is affected by fragmentation, third-party DLLs , and other running processes. You can limit the size of the largest array that MATLAB can create. To set the limit, go to: MATLAB Preferences -> Workspace -> MATLAB array size limit.

Preallocation of Memory.

When your code runs slower than expected, one of the things you can look at is how the variables are created and modified as the code runs.

MATLAB allows variables to be resized dynamically, for example, during the individual iterations of a loop. This is convenient, but in some cases can lead to poor performance.

- Preallocating memory in advance is the recommended approach.
- For numeric arrays, preallocation can be done using the zeros/ones/NaN function.
- It is important to preallocate for the correct data type. All numeric data types are supported by zeros and ones; NaN may initialise double or single values.
- Cells and structures may be preallocated using the cell, struct and repmat functions.

```
clear all
x = zeros(1,50000);
whos
```

```
  Name      Size                Bytes  Class     Attributes

  x         1x50000            400000  double
```

```
%You can create an array having other datatypes by specifying an additional input.
```

```
x = zeros(1,50000,'single');
y = zeros(1,50000,'int16');
whos
```

```
  Name       Size                  Bytes  Class     Attributes

  x          1x50000              200000  single
  y          1x50000              100000  int16
```

```
%Another way to preallocate an array is to assign a value to the last element of the array.
x(8) = 3; %for this example check the workspace
whos
```

```
  Name       Size                  Bytes  Class     Attributes

  x          1x50000              200000  single
  y          1x50000              100000  int16
```

- To preallocate a cell array using the function **`cell`**.
- To preallocate a structure array, start by defining the last element of the array. MATLAB will automatically replicate the field names to all of the preceding elements in the array.

```
C = cell(1,4);
S(5) = struct('field1',6,'field2',7)
```

```
S = 1×5 struct array with fields:
    field1
    field2
```

```
whos
```

```
  Name       Size                  Bytes  Class     Attributes

  C          1x4                      32  cell
  S          1x5                     432  struct
  x          1x50000              200000  single
  y          1x50000              100000  int16
```

Study the function makeA.m in the folder course_examples/usingmatlab/vectorize

Run the function by enterting the following command in the command window. Observe the elapsed time.

`tic; A = makeA(5000,100); toc`

Write code to preallocate the array A inside the function and run the function again by enterting the above command in the command window.

Observe the effect of preallocation on the elapsed time.

```
addpath('../matlab_examples/usingmatlab/vectorize/');
edit('makeA.m');
tic; A = makeA(5000,100); toc
```

```
Elapsed time is 0.757988 seconds.
```

Vectorisation

MATLAB is an array-based language, and as such all vector and matrix operations are optimised for performance. Replacing sequences of scalar operations performed in loops with smaller numbers of vector and matrix operations is referred to as vectorisation. This process can lead to more readable and efficient code.

Think of the best way of completing each of the following tasks in MATLAB:

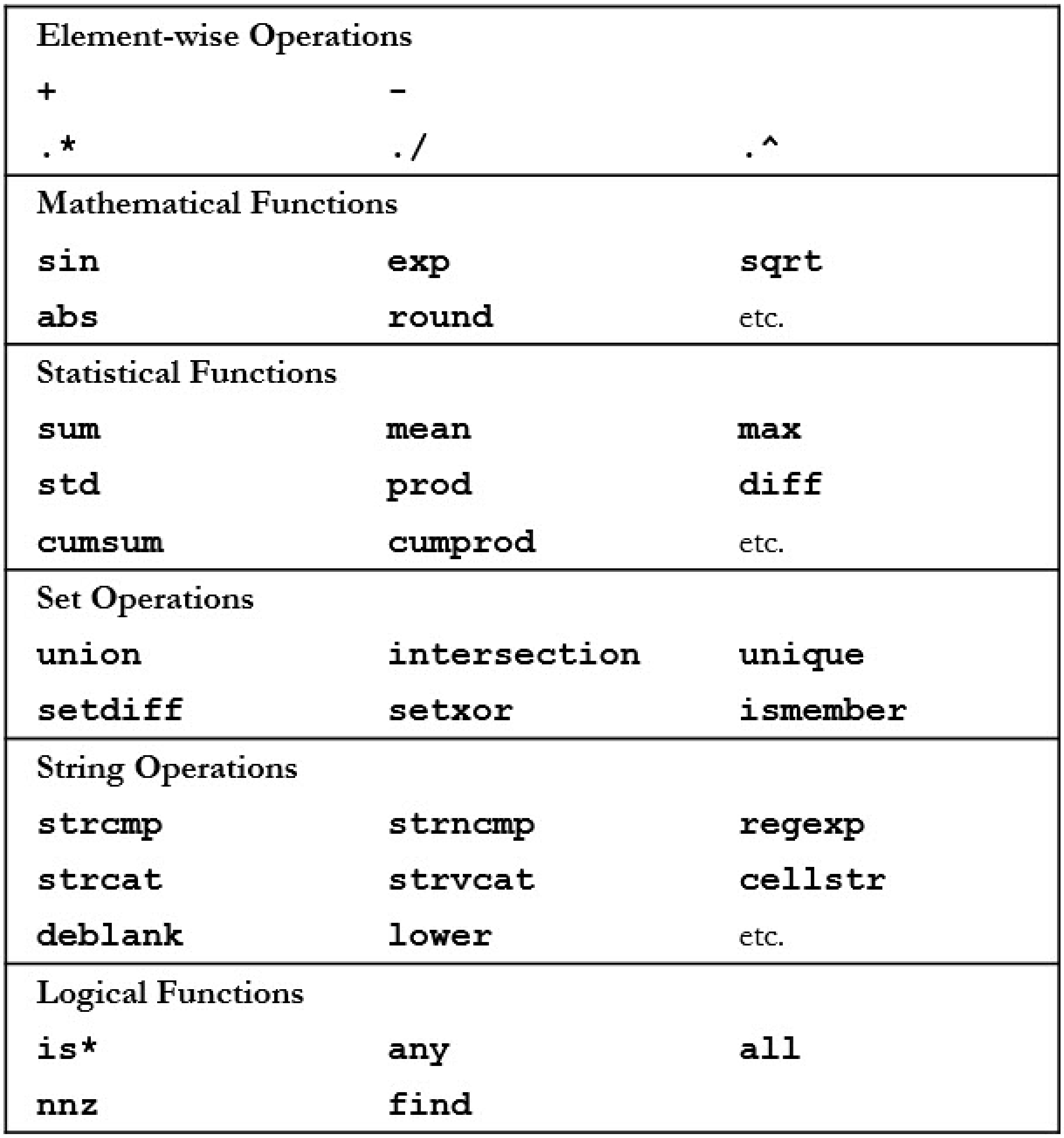
1. Add two vectors `x` and `y`.
2. Find the square root of each of the elements of the vector `n`.
3. Calculate the difference between the adjacent elements of the vector `x`.

Each of the above tasks can be performed using a `for` loop. However, MATLAB has many vectorized functions and operators which provide an alternative to the`for` loop and are often faster and more concise.

Other useful functions include `sum, diff, prod, gradient, del2`

•Some matlab routines are *.m files, others are compiled into matlab (built-in) and are much faster. Try to use built-ins where possible. Use type *functionname* to see if a function is a built-in.

A subset of vectorized functions and operations is provided below. You should look for opportunities to vectorize your code and consult MATLAB documentation for possible solutions.



Element-wise Operations		
`+`	`-`	
`.*`	`./`	`.^`
Mathematical Functions		
`sin`	`exp`	`sqrt`
`abs`	`round`	etc.
Statistical Functions		
`sum`	`mean`	`max`
`std`	`prod`	`diff`
`cumsum`	`cumprod`	etc.
Set Operations		
`union`	`intersection`	`unique`
`setdiff`	`setxor`	`ismember`
String Operations		
`strcmp`	`strncmp`	`regexp`
`strcat`	`strvcat`	`cellstr`
`deblank`	`lower`	etc.
Logical Functions		
`is*`	`any`	`all`
`nnz`	`find`	

•Matlab scripts can be written using fortran/C-style "for" loops, but to make the most of matlab's abilities you should try whenever possible to treat matrices as a single entity rather than a bundle of elements.

•Vectorisation functions

–`arrayfun, strutfun, cellfun`

–`repmat`

–`cumsum`

–`bsxfun` (Binary Singleton eXpansion FUNction)

```
tic
for i = 1:100
 for j = 1:100
```

```
    r(i,j) = sqrt(i^2+j^2);
  end
end
toc
```

```
Elapsed time is 1.872856 seconds.
```

vectorised this becomes

```
tic
[i,j]=meshgrid(1:100,1:100);
r = sqrt(i.^2+j.^2);
toc
```

```
Elapsed time is 0.043231 seconds.
```

```

```

Note for small loops there is no discernable performance gain, for what size of loop do we gain benefit? Investigate and find out!

Vectorisation: Example 2

```
tic
n=1000; x(1)=1;
for j=1:n-1,
 x(j+1) = x(j) + n - j;
end
toc
```

```
Elapsed time is 0.227259 seconds.
```

vectorised using the cumulative summation function `cumsum`

```
tic
n=1000; x(1)=1;
j=1:n-1;
x(j+1) = n - j;
cumsum(x);
toc
```

```
Elapsed time is 0.060030 seconds.
```

The cellfun, strutfun and arrayfun

Apply function to each element of array or cellaray using `arrayfun` or `celfun` or `strutfun`

`[B1,...,Bn] = arrayfun(func,A1,...,An)`

`[A1, ...,An ]= cellfun(func,C1,...,Cn)`

```
tic
x=-pi:pi/1000:pi;
```

```
y=arrayfun(@sin,x);
toc
```

```
Elapsed time is 0.028436 seconds.
```

compare with for loop

```
tic
x=-pi:pi/1000:pi;
for i=1:numel(x)
    y(i)=sin(x(i));
end
toc
```

```
Elapsed time is 0.423526 seconds.
```

For in-buillt and functions we can easily write!

```
tic
x=-pi:pi/1000:pi;
y=sin(x);
toc
```

```
Elapsed time is 0.032401 seconds.
```

Handling arrays of different sizes

```
clear all
[4;5;6] > [1 2 3; 4 5 6; 7 8 9] %note for later versions of matlab matrix will automatically b
```

```
ans = 3×3 logical array
   1   1   1
   1   0   0
   0   0   0
```

```
                                %do the computation

%the following solution increases memory usage

reshape(repmat([4;5;6],3,1),3,3) > [1 2 3; 4 5 6; 7 8 9]
```

```
ans = 3×3 logical array
   1   1   1
   1   0   0
   0   0   0
```

`bsxfun` to perform operations on arrays having different dimensions. This MATLAB function applies the element-wise binary operation specified by the function handle fun to arrays A and B.

```
x = [4;5;6]
```

```
x =

     4
     5
```

```
      6
```

```
y = [1 2 3; 4 5 6; 7 8 9];
bsxfun(@gt,x,y)
```

```
ans = 3×3 logical array
   1   1   1
   1   0   0
   0   0   0
```

Note above the handle to the 'greater than' function `gt`. Note that all MATLAB operators have equivalent function forms. For example, the > and + operators are equivalent to the `gt` and `plus` functions, respectively.

bsxfun does the following

1. Replicate input arrays along singleton dimension until resulting dimensions match.
2. Perform binary function on each element to produce output.

Copy on write behaviour

When assigning one variable to another, MATLAB does not create a copy of that variable until it is necessary. Instead, it creates a reference.

When you assign a copy of a large variable to another variable, how is the 'available memory' affected? Such situations often arise when passing inputs to a function.

MATLAB does maintain the local copy of the variable inside the function. However, the copy is simply a reference to the variable which was passed as a function input. MATLAB breaks the reference, and creates an actual copy of that variable, only when code modifies one or more of the values.

This behavior, known as copy-on-write, or lazy-copying, ensures that no additional memory is assigned to an input variable until it is modified within the function.

```
clear all
A = rand(6e3);
m = memory;
disp('Memory available (GB):')
```

```
Memory available (GB):
```

```
disp(m.MemAvailableAllArrays/1073741824)
```

```
   21.4133
```

```
B = A;
% This is only a reference at the moment, so available memory should
% remain constant.
m = memory;
disp('Memory available (GB):')
```

```
Memory available (GB):
```

```
disp(m.MemAvailableAllArrays/1073741824)
```

```
21.4025
```

```
B(1, 1) = 0;
% Now that we have made a change, copy-on-write should kick-in.
m = memory;
disp('Memory available (GB):')
```

```
Memory available (GB):
```

```
disp(m.MemAvailableAllArrays/1073741824)
```

```
21.1320
```

In-Place Optimisation.

An in-place optimisation conserves memory by using the same variable for an output argument as for an input argument. This is valid only when the input and output have the same size and type, and only within functions. When working on large data and memory is a concern, in-place optimisations could be used to attempt to conserve memory use. However, in some situations, it is not possible to have a true in-place optimisation, because some temporary storage is necessary to perform the operation.

When performing calculations on a variable having large size, you can improve memory usage if you avoid using temporary variables.

Consider the following code snippet.

```
y = x.^2 + 1;
```

If the variable x is not required in the program after the calculation is performed, you can avoid creating an additional variable y by assigning the result back to x. This approach saves memory as well as execution time for allocating new memory.

```
x = x.^2 + 1;
```

You can also benefit from the in-place optimization if you declare and call a function using the same variable to define the input and output arguments.

```
clear all
edit('inplace1.m')
edit('inplace2.m')
x1=rand(5000);
x1=inplace1(x1);
m = memory;
disp('Memory available (GB):')
```

```
Memory available (GB):
```

```
disp(m.MemAvailableAllArrays/1073741824)
```

```
21.4637
```

```

clear all
x2=rand(5000);
x2=inplace2(x2);
m = memory;
```

```
disp('Memory available (GB):')
```

```
Memory available (GB):
```

```
disp(m.MemAvailableAllArrays/1073741824)
```

```
    21.4639
```

If `inplace1` or `inplace2` modifies the input variable and returns it in an appropriate way, the operation will be carried out without an additional copy in the MATLAB workspace.

This strategic technique is useful for large data sets, when a duplicate of the variable would immediately stop the execution by throwing an out-of-memory error.

Nested Loops

MATLAB functions are typically used for organizing the code, avoiding replication, and hiding functionality.

- You can also use the functions for managing memory and interfaces.
- write nested functions to effectively share large data between functions and conforming to a predetermined function signature while writing certain applications.
- Like a local function, a nested function is also used to organize several functions in a single function file and it often acts as a helper function to the primary function.

However, nested functions exihibit one additional feature: They allow functions in the file to share workspace variables.

In the grossIncome example, below, the functions `convertToDollar` and convertToStirling share workspace variables with the `grossIncome` function and with each other.

Note that even while the shared data is accessible from anywhere within the file, it is hidden from the outside world.

How to create a nested function?

To nest a function inside another function, the extents of the functions must be marked explicitly by using the keywords `function` and `end`.

```
edit('grossIncome.m')
grossIncome(50,100, 80)
```

```
ans = 380
```

In the example above if `dollarIncome,euroIncome, stirlingIncome` are large, using a nested function is more efficient because it operates directly on the data of the parent function.

More Performance More Data!

Distributed Computing

•Coarse grained

–Multiple tasks

–High throughput computing

•Fine grained

–Distributed computing

–MPI programming distributed matrices

–GPU Arrays

Acces to More Resources

•HPC

–Sheffield Advanced Research Computer

•Benefits

–Access to data

–More compute resources

–Share applications with collaboration partners

Make Matlab Use more Resources

•Compute Cluster e.g. Iceberg

•Uses a scheduler e.g. Sun Grid Engine

•Break a matlab job in to many independent tasks

–E.g. cycle of a for loop might run independently

•Array of matlab tasks running on compute cluster

•Can run 1000's of tasks

•Good for

–parametric sampling

–Genetic algorithms

–Model ensembles

Submitting a matlab job to Sun Grid Engine ShARC

•qsub mymatlabjob.sh

•Where mymatlabjob.sh is:

```
#!/bin/sh

#$ -cwd

#$ -l h_rt=hh:mm:ss

#$ -l rmem=12G

module load apps/matlab/2017a
```

matlab -nojvm -nosplash -nodisplay < matlabscriptfile > outputfile

```
addpath('../matlab_examples/casestudies/beats_taskarray/')
edit('beats.m')
edit('beats.sh')
```

Final Remarks

•Task arrays are a powerful method utilising large pools of compute resources e.g. clusters

•Mainly use batch submission mode not interactive use of matlab

•Parallel computing toolbox enables researchers to develop interactive matlab applications able to exploit a pool of compute resources

Algorithm Development - A Case Study

We are now happy writing scripts to organise and collect sequences of MATLAB commands, but these become increasingly difficult to manage as the complexity of our algorithms grows. Moreover, a common requirement is to run scripts repeatedly using different values for certain parameters within the script. Functions are a more effective programming construct for managing these issues. Moving forward an important concept here is how to modularise code so that it becomes reusable, maintainable, flexible and robust.

For our algorithm development case study we present the development of a three body gravity simulation (earth, moon and rocket). The objective is to alter the force on the rocket so that eventually moves into orbit around the moon. An interactive graphical display is used to present the evolution of the orbital state of the system. Each developmental stage of the application is presented in the follwoing sub folders of /matlab_examples/casestudies/mission-moon

- code-001 - Initial prototype
- code-002-newfunctions - Replacing parts of the script with gravaccel and distance functions
- code-003-newdatastructures - Using data structures to store the simulation state, simulation controls and simulation constants
- code-004-functioncalls - Replacing scripts for initialisation, graphics start up and to start the simulation with function calls
- code-005-tests - Implementing unit tests
- code-006-gui -