Further MATLAB Programming – Make Your Code Efficient and Robust

Louise Brown, Yijie (Amy) Zheng

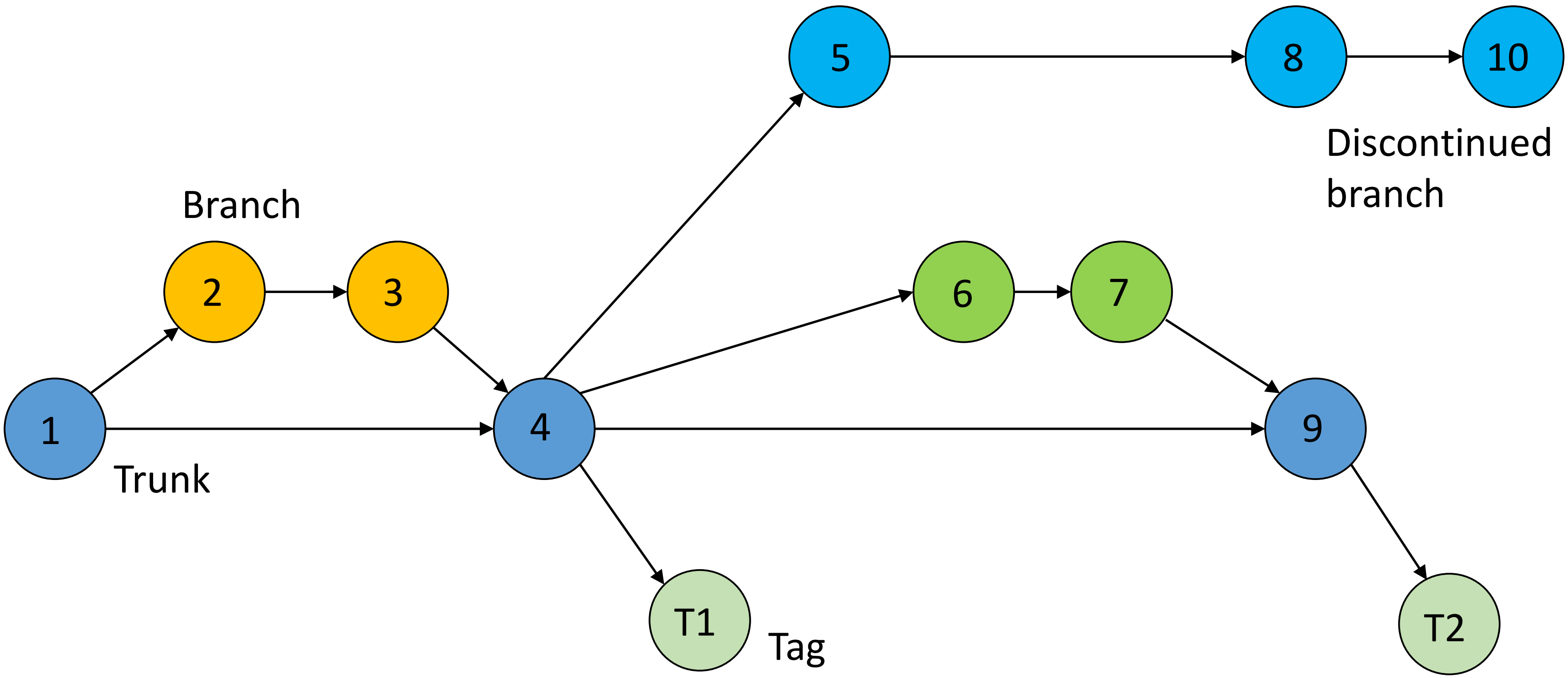
Version Control

Does this look familiar?

TestCode.m
TestCode_data_set1.m
TestCode_data_set1_v2.m
TestCode_data_set1_v2_with_output.m

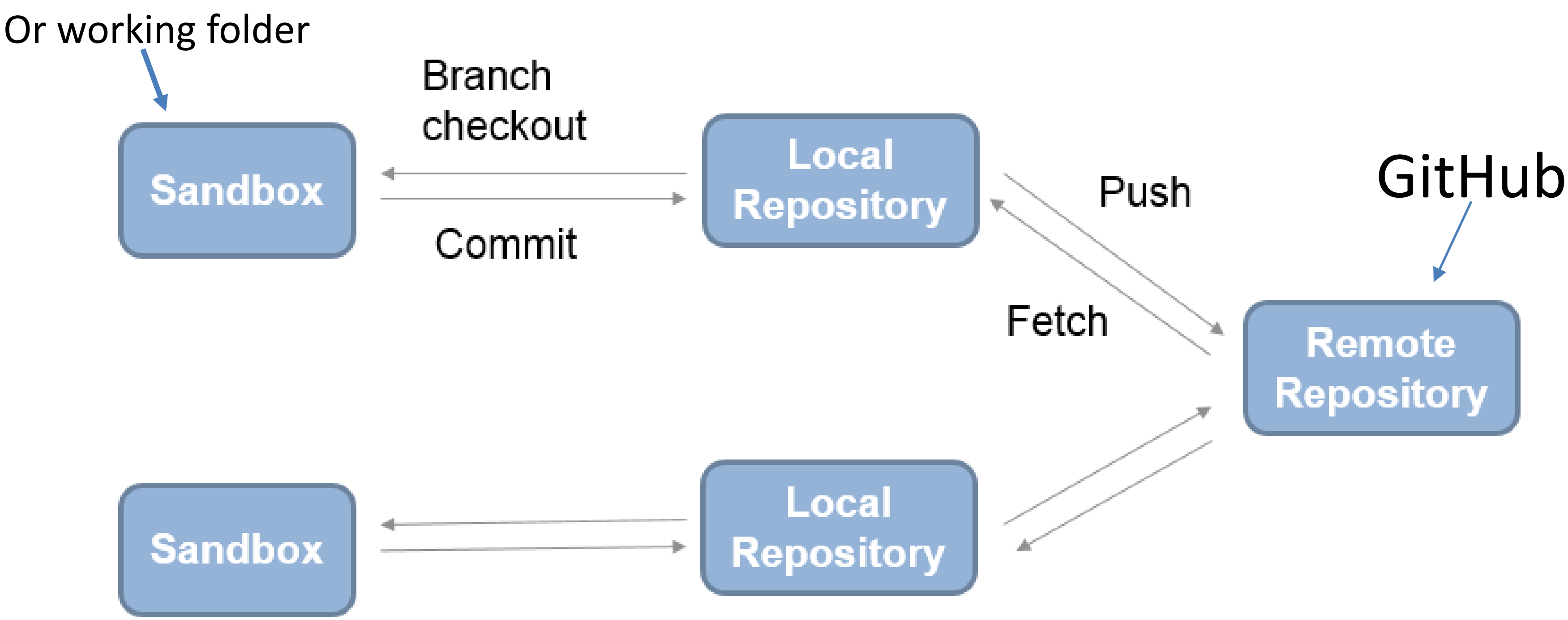
Use version control, eg Git, Subversion or Mercurial

Git Workflow



Tags can correspond to code producing results for research papers

Git Workflow

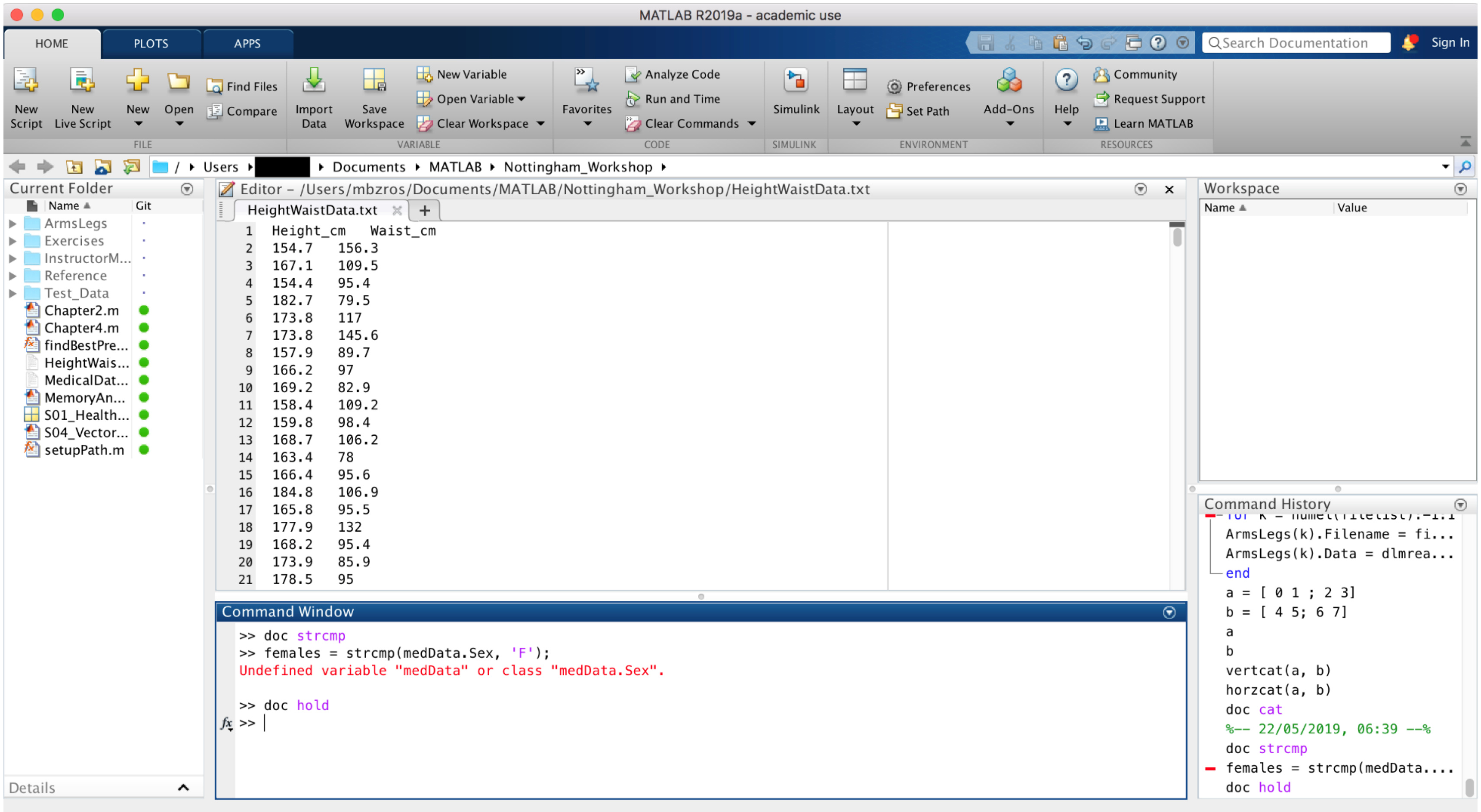


Use git in MATLAB:
https://uk.mathworks.com/help/matlab/matlab_prog/use-git-in-matlab.html

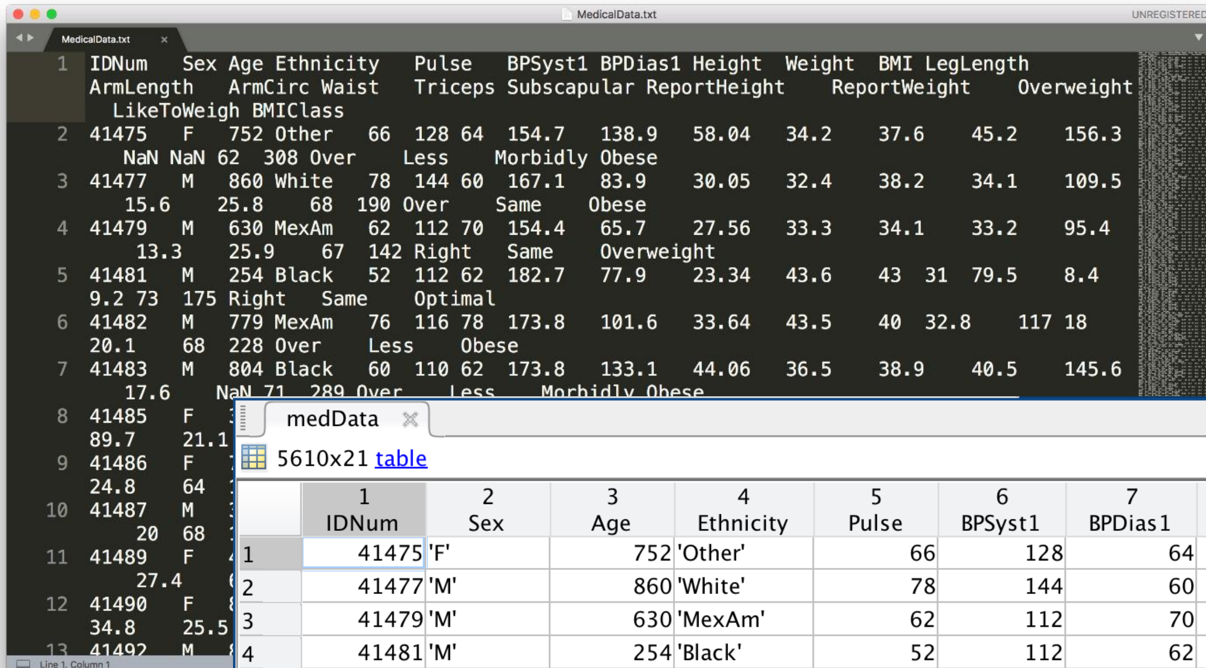
Fork and Clone Course Materials

- louisepb/Further-MATLAB-Student-Materials on GitHub
- Fork the repo
- Clone the repo into BootCampFiles folder
- Run setupPath.m

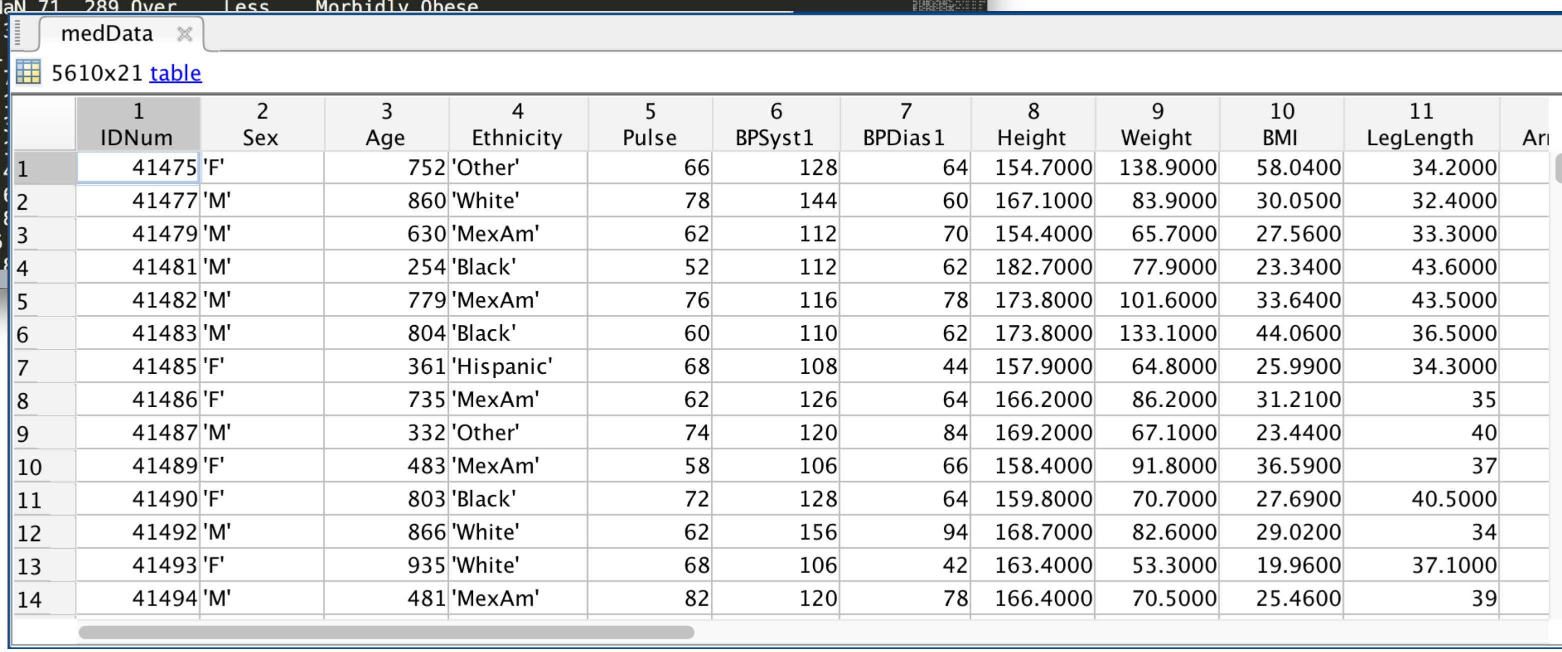
The MATLAB Desktop



Heterogeneous data




```
IDNum	Sex Age Ethnicity	Pulse	BPSyst1 BPDias1 Height	Weight	BMI LegLength
ArmLength	ArmCirc Waist	Triceps Subscapular ReportHeight	ReportWeight	Overweight
 LikeToWeigh BMIClass
41475	F	752 Other	66	128 64	154.7	138.9	58.04	34.2	37.6	45.2	156.3
   NaN NaN 62	308 Over	Less	Morbidly Obese
41477	M	860 White	78	144 60	167.1	83.9	30.05	32.4	38.2	34.1	109.5
   15.6	25.8	68	190 Over	Same	Obese
41479	M	630 MexAm	62	112 70	154.4	65.7	27.56	33.3	34.1	33.2	95.4
    13.3	25.9	67	142 Right	Same	Overweight
41481	M	254 Black	52	112 62	182.7	77.9	23.34	43.6	43	31	79.5	8.4
9.2 73	175 Right	Same	Optimal
41482	M	779 MexAm	76	116 78	173.8	101.6	33.64	43.5	40	32.8	117 18
20.1	68	228 Over	Less	Obese
41483	M	804 Black	60	110 62	173.8	133.1	44.06	36.5	38.9	40.5	145.6
```



medData

5610x21 table

	1 IDNum	2 Sex	3 Age	4 Ethnicity	5 Pulse	6 BPSyst1	7 BPDias1	8 Height	9 Weight	10 BMI	11 LegLength
1	41475	'F'	752	'Other'	66	128	64	154.7000	138.9000	58.0400	34.2000
2	41477	'M'	860	'White'	78	144	60	167.1000	83.9000	30.0500	32.4000
3	41479	'M'	630	'MexAm'	62	112	70	154.4000	65.7000	27.5600	33.3000
4	41481	'M'	254	'Black'	52	112	62	182.7000	77.9000	23.3400	43.6000
5	41482	'M'	779	'MexAm'	76	116	78	173.8000	101.6000	33.6400	43.5000
6	41483	'M'	804	'Black'	60	110	62	173.8000	133.1000	44.0600	36.5000
7	41485	'F'	361	'Hispanic'	68	108	44	157.9000	64.8000	25.9900	34.3000
8	41486	'F'	735	'MexAm'	62	126	64	166.2000	86.2000	31.2100	35
9	41487	'M'	332	'Other'	74	120	84	169.2000	67.1000	23.4400	40
10	41489	'F'	483	'MexAm'	58	106	66	158.4000	91.8000	36.5900	37
11	41490	'F'	803	'Black'	72	128	64	159.8000	70.7000	27.6900	40.5000
12	41492	'M'	866	'White'	62	156	94	168.7000	82.6000	29.0200	34
13	41493	'F'	935	'White'	68	106	42	163.4000	53.3000	19.9600	37.1000
14	41494	'M'	481	'MexAm'	82	120	78	166.4000	70.5000	25.4600	39

doc readtable

Tables

- Useful for heterogenous, column oriented or tabular data
- Variables can have different data types
- All columns must have the same number of rows
- Not restricted to column vectors (eg could have matrix but number of rows condition still applies

Create table from workspace data using the `table` function:

```
TableName = table(var1, var2, var3,… );
```

or by using `readtable` to load a data set:

```
TableName = readtable('data.dat');
```

Indexing into Tables

Address column data using dot notation:

```
PatientData.Gender  % Accesses the whole column
```

Then normal indexing for the data type:

```
PatientData.Gender(10) % Accesses data in column
```

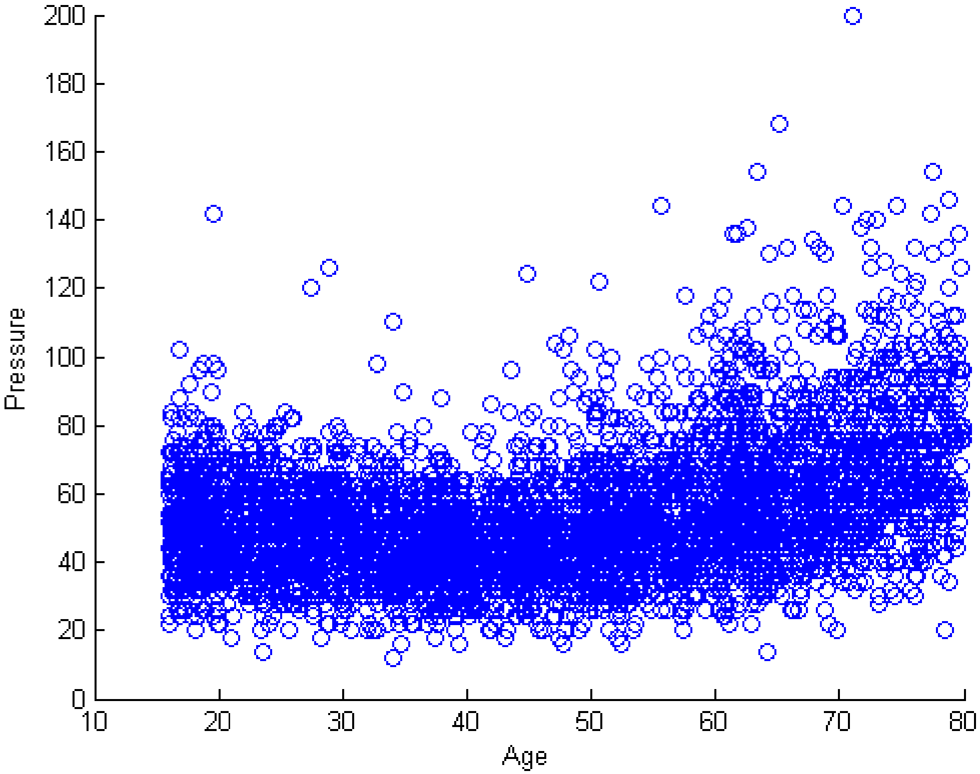
Access particular rows and columns using subscript notation:

```
PatientData(1:3, :);
ans =

     Gender      Age    Height    Weight
    ________     ___    ______    ______

    'Male'       38     71        176
    'Male'       43     69        163
    'Female'     38     64        131
```

Plotting vectors




```
figure % new figure

% basic plot
scatter(medData.Age, medData.BPDiff)
xlabel(‘Age’)
ylabel(‘Pressure’)
```

Low-Level file I/O

HeightWaistData.txt — Edited

```
Height_cm	Waist_cm
154.7	156.3
167.1	109.5
154.4	95.4
182.7	79.5
173.8	117
173.8	145.6
157.9	89.7
166.2	97
169.2	82.9
158.4	109.2
159.8	98.4
168.7	106.2
163.4	78
166.4	95.6
184.8	106.9
```

```
fileID = fopen(‘HeightWaistData.txt’);

dataFormat = ‘%f %f’;

heightWaistData = textscan(fileID, dataFormat, ...
‘HeaderLines’, 1, ‘Delimiter’, ‘\t’);

fclose(fileID);
```

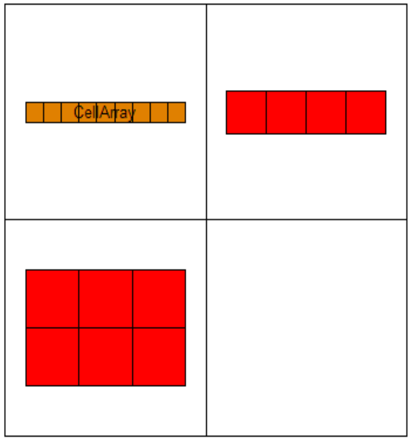
Cell Arrays

- Store different types of data
- Accessed by index
- Particularly useful for storing different length strings
- Can be used for importing spreadsheet data with different data types in the cells

```
>> A = ‘CellArray‘
>> B = 1:4;
>> C = [1:3,2:4];
>> CellArray = {A B; C []}
```

Curly brackets to specify cell array

Must specify empty elements or MATLAB will generate error



Visualise array using `cellplot` command
`celldisp` displays cell array contents

Accessing Cell Array Data

Note difference between use of () and {}:

Use CellArray(m,n) to display data structure

```
>> CellArray(1,2)
ans =
    [1x4 double]
```

Use CellArray{m,n} to display contents of the cell

```
>> CellArray{1,2}
ans =
     1     2     3     4
```

To access individual elements within a cell use ():

```
>> % second element of array at cell position (1,2)
>> CellArray{1,2}(2)
ans =
      2
```

Extending Cell Arrays

Add more cells using the {} notation:

Note that array is padded to keep rows and columns consistent

```
>> CellArray{3,3} = 1:4
CellArray =
    'CellArray'     [1x4 double]               []
    [2x3 double]              []               []
              []              []     [1x4 double]
```

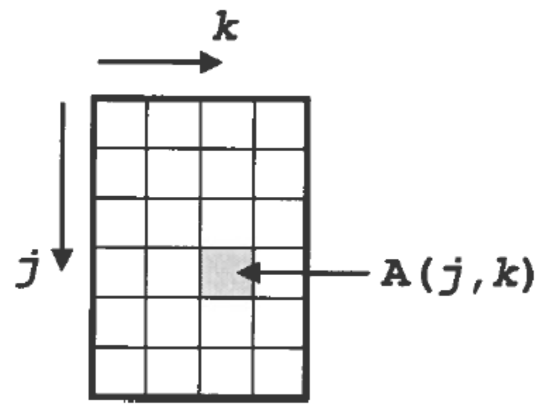
Preallocate cell arrays using the `cell` function:

`cell(10,10)` creates an empty 10x10 cell array

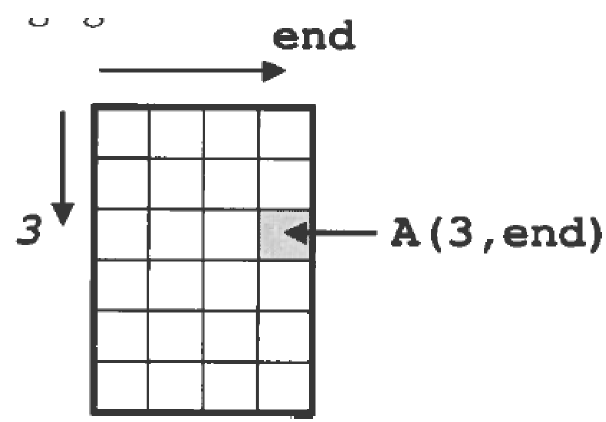
Accessing Data in Arrays

```
>> x = A(j,k)
```

Row Column



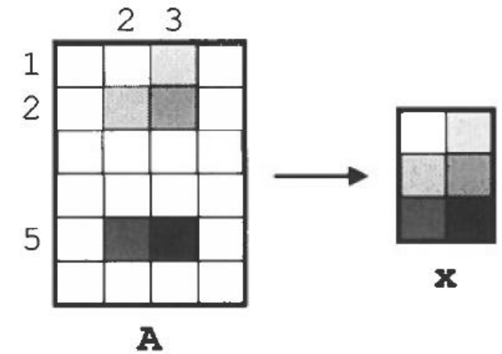

```
>> x = A(3,end)
```



Accessing Multiple Elements

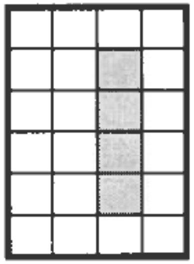
Use a vector of indices to reference multiple elements:

```
>> x = A([1,2,5],[2,3]);
```

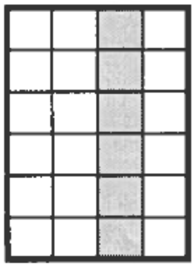


Use colon notation to indicate a range of rows and/or columns:

```
>> x = A(2:5, 3);
```



```
>> x = A(1:6, 3);
>> x = A(1:end, 3);
>> x = A(:, 3);
```



all extract the entire third column.

Dealing with Missing Data

Bad or missing data will be imported as NaN

This can be removed using logical indexing

```
badData = isnan(Height);
CleanHeight = Age(~badData);
```

Gives logical vector with value of 1 where data is NaN

Use ~ (not) so that good data has value of 1 in logical vector

Structures

Related data grouped together in fields which are referred to by name

alternate field names and contents

Create by:

```
>> cars = struct( 'year', 2010,
'colour','red','mpg', 35)
cars =
       year: 2010
     colour: 'red'
        mpg: 35
```

Or:

```
>> cars.year = 2010;
>> cars.colour = ‘red’;
>> cars.mpg = 35;
```

Arrays of Structures

Create arrays of structures either by adding another structure:

```
cars(2) = newcar;
```

Or by assigning new elements directly:

```
>> cars(3).year = 2012;
>> cars(3).colour = 'blue';
>> cars(3)
    ans =
          year: 2012
        colour: 'blue
           mpg: []
```

Any unassigned fields will be present in the structure but will be empty

	year	colour	mpg
cars(1)	2010	red	35
cars(2)	2008	black	42
cars(3)	2012	blue	

Linear Regression Models

- Regression is the process of using measured data to fit a model of a relationship between variables
- In this case between pulse pressure and age
- Propose a parametric model: $Y = f(X; \beta)$

  X is a vector of independent , or predictor, variables (age)

  Y is a vector of the dependent, or response, variables (pulse pressure)

  β is a vector of parameters

Linear Regression Models

- Models where f is linear in the parameters β are called *linear regression models*
- $P = F(A; c) = c_0 + c_1 A + c_2 A^2$
- The model can be written as

$$\underbrace{y}_{\text{Response}} = \sum \underbrace{\beta_k}_{\text{Model parameters}} \underbrace{f_k}_{\text{Design functions}}(\underbrace{x}_{\text{Predictor}})$$

for some set of *basis* or *design* functions f_k

Matrix Equations

Fitting *n* measured data points gives a system of *n* linear equations with *m* unknown model parameters (c_j)

$$
\begin{aligned}
y_1 &= c_1 f_1(x_1) + c_2 f_2(x_1) + \ldots + c_m f_m(x_1) \\
y_2 &= c_1 f_1(x_2) + c_2 f_2(x_2) + \ldots + c_m f_m(x_2) \\
&\vdots \\
y_n &= c_1 f_1(x_n) + c_2 f_2(x_n) + \ldots + c_m f_m(x_n)
\end{aligned}
$$

Using matrix algebra, this system can be written in matrix-vector form as:

$$
\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}
\begin{matrix} = \\ = \\ \\ = \end{matrix}
\begin{bmatrix} f_1(x_1) & f_2(x_1) & \ldots & f_m(x_1) \\ f_1(x_2) & f_2(x_2) & \ldots & f_m(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ f_1(x_n) & f_2(x_n) & \ldots & f_m(x_n) \end{bmatrix}
\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}
$$

Slash and Backslash

- Matrix multiplication is not commutative (generally $AB \neq BA$)
- $AX = B$ and $XA = B$ represent different systems of equations for X
- MATLAB uses / and \ to distinguish between these:

`>> X = B/A` represents "solve the system $XA = B$", while

`>> X = B\A` represents "solve the system $AX = B$"

Anonymous Functions

Anonymous functions are local functions only available until the workspace is cleared. They are called anonymous because they do not refer to a named function.

```
Fnhandle = @ (input arguments) functioncode;
```

Shows function handle being created

```
f = @(x,y) sin(x) + y
```

Can be saved in .mat file using load and save commands

```
>> f(pi/6, 0.5)

ans =

    1
```

Anonymous Functions

Anonymous function creating function handle

```
modelFun = @(c, x1, x2) c(1)+c(2)*x1+c(3)*x2c(4)*x1.^2 + ...
    c(5)*x2.^2 + c(6)*x1.*x2;
```

Mathematical notation

```
f(x1, x2) = c(1) +c(2)*x1 +c(3)*x2 + c(4)*x1.^2 + ...
   c(5)*x2.^2 + c(6)*x1*x2
```

$$f(x_1,x_2) = c_1 + c_2x_1 + c_3x_2 + c_4x_1^2 + c_5x_2^2 + c_6x_1x_2$$

Making Grids

The `meshgrid` function converts vectors of points into matrices that can represent a grid of points in the x-y plane

```
>> x = -2:4; % 1 x 7 vector
>> y = 0:5;  % 1 x 6 vector
>> [X,Y] = meshgrid(x,y)
```

42 points in grid of points in the x,y plan

```
X =
    -2    -1     0     1     2     3     4
    -2    -1     0     1     2     3     4
    -2    -1     0     1     2     3     4
    -2    -1     0     1     2     3     4
    -2    -1     0     1     2     3     4
    -2    -1     0     1     2     3     4
```

x coordinate at each grid point

```
Y =
     0     0     0     0     0     0     0
     1     1     1     1     1     1     1
     2     2     2     2     2     2     2
     3     3     3     3     3     3     3
     4     4     4     4     4     4     4
     5     5     5     5     5     5     5
```

y coordinate at each grid point

Creating a Function

Must have function declaration on line 1 of file

Output arguments

Must be the same as the filename – fitQuadModel.m

Input arguments

```
function [modelCoeffs, fh] = fitQuadModel(X, y, showplot)
% FITQUADMODEL Fits a quadratic model to the response data
% using the columns of X as regressors. X has one or two
% columns; y is a vector with the same number of rows as X.
```

Software Testing

Please take a look at this blog before the session tomorrow:

[A Brief Introduction to Software Testing]

Main test function

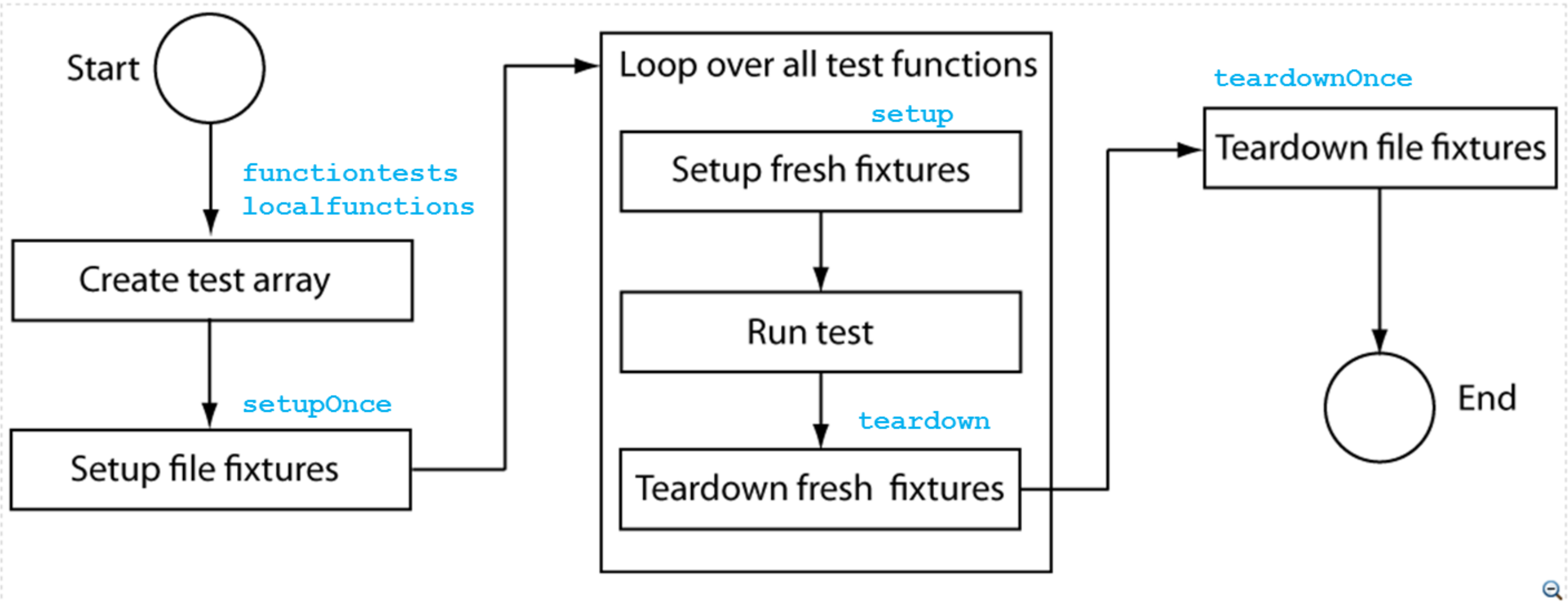
```
Function tests = testFunction
      tests = functiontests( localfunctions() );
End

function test_func1(testcase)
      …
end

function test_func2(testcase)
      …
end
```

Assembles cell array of function handles for local functions in current file

Task Execution



Debugging and Improving Performance

- Diagnosing problems
- Identifying common errors
- Evaluation of code performance
- Vectorisation techniques
- Managing memory effectively

Preallocation of Memory

Efficient matrix and array operations rely on data being located together in a contiguous block of memory addresses

Create a dummy version of a variable of appropriate size

```
>> A = zeros(m,n)
```

Subsequent operations then overwrite the zeros with the required values

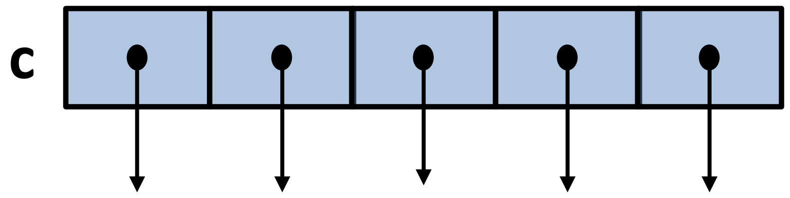
Assigning the last element of an array first creates an array of the appropriate size

```
>> x(8) = 3
x =
     0     0     0     0     0     0     0     3
```

Preallocating Cells and Structures

Cell arrays and structure arrays act as containers for various types of data

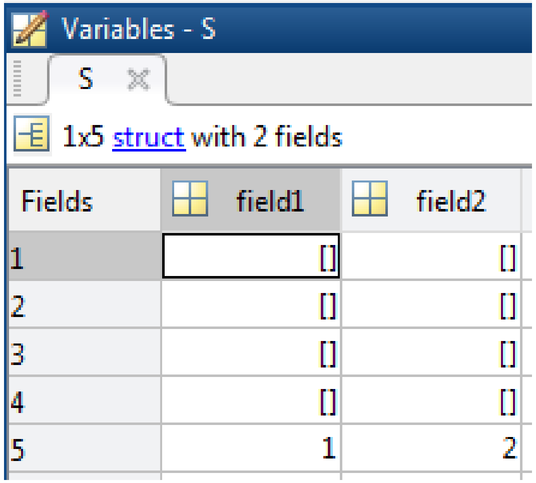
```
>> c = cell(1,5)
```



The `cell` command preallocates the container

Define the last element in a structure array

```
>> S(5).field1 = 1;
>> S(5).field2 = 2;
```



In-Place Optimisation

In-place optimisation saves on memory by reusing an input variable for output

```
y = 2*x + 3;
x = 2*x + 3;
```

Saves memory by assigning back into variable `x` and execution time for allocating memory

Functions

```
....
x = myfun(x)
....
```

Regular Function

```
Function y = myfun(x)
y = 2*x + 3;
```

In-Place Optimisation

```
Function x = myfun(x)
x = 2*x + 3;
```