Scientific Computing Lecture 1: MATLAB fundamentals

InFoMM CDT
Mathematical Institute
University of Oxford

Andrew Thompson
(slides by Stuart Murray)

Lecture I: MATLAB fundamentals

What is MATLAB?

MATLAB combines many things:

- command line based interaction;
- suite of quality solvers/functions;
- programming language;
- interactive development environment (IDE);
- debugger and code analyzer.

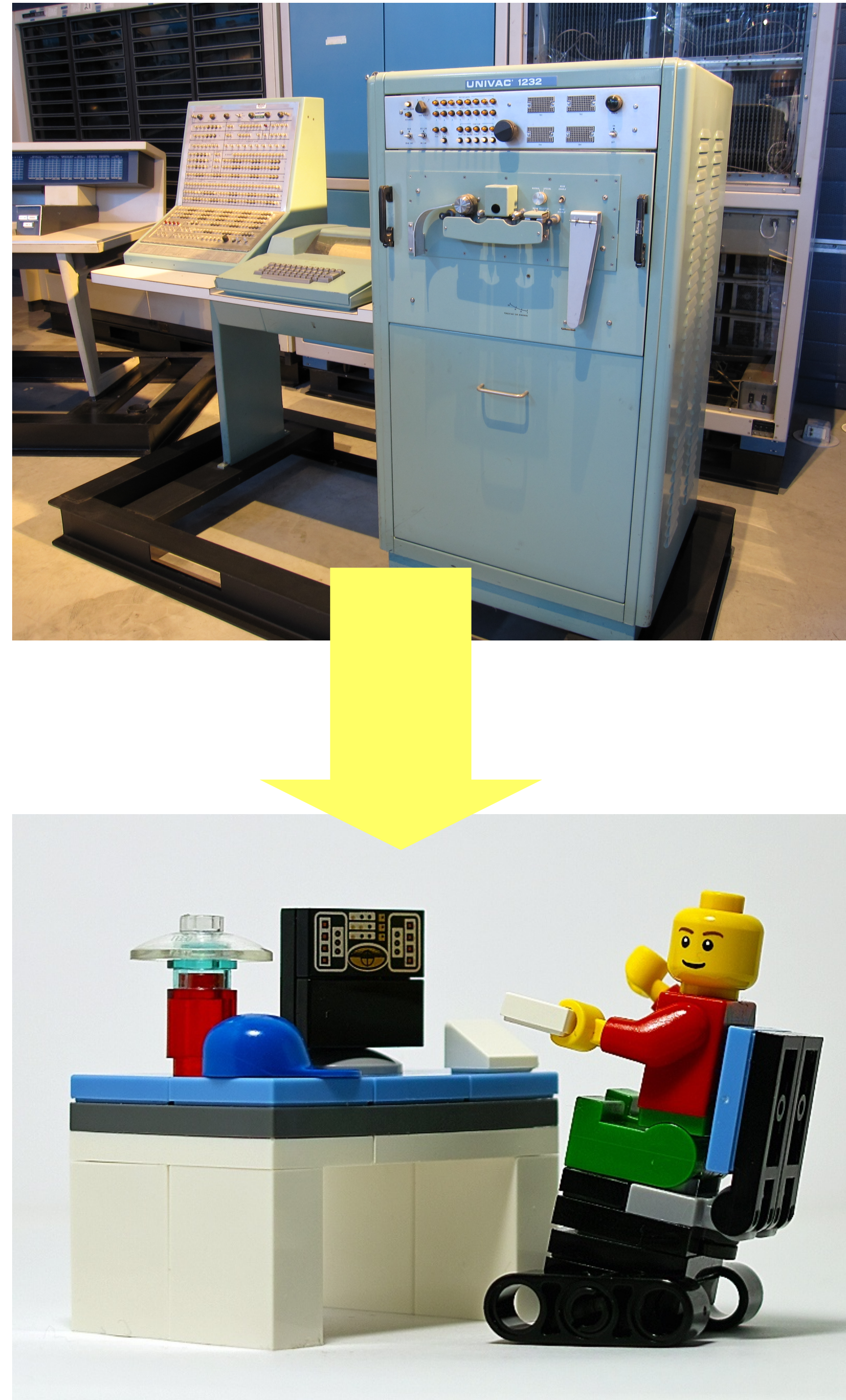
Designed for numerical problem solving

Differs from Maple and Mathematica in this respect

Some symbolic manipulation possible, but will not be covered

History and aims

- Invented by Cleve Moler in 1970s
- Interface to Fortran packages LINPACK and EISPACK
- Removes some of the pain from numerical programming
- Prevents reinvention of the wheel: routines for performing many tasks are included
- Now very evolved: range of toolboxes, support for parallel computing, object oriented
- Industry standard in many fields
- Fast if used correctly



MATLAB fundamentals

Getting started in the command window

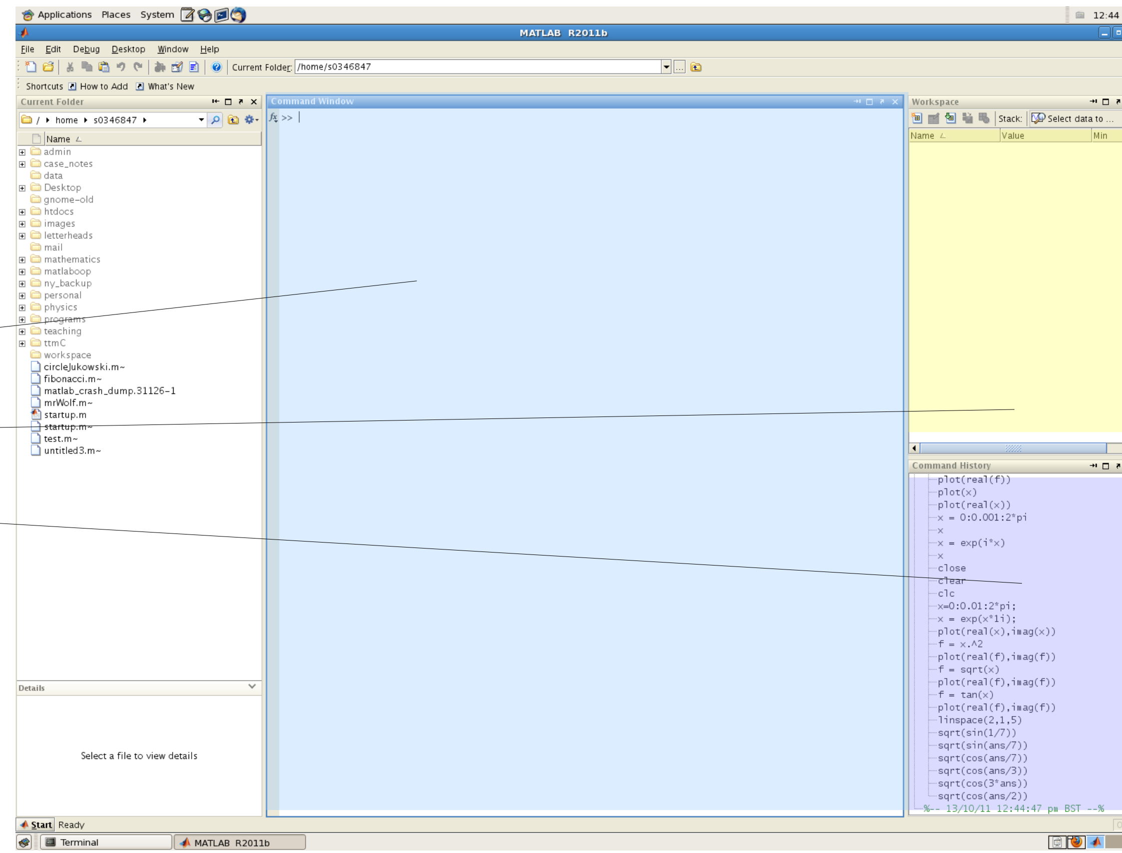
Open MATLAB

There are 3 main windows:

Command window

Workspace window

Command history



The command window is the means by which we will interact with MATLAB
Prompt looks like this:

\>>

Simple calculations

Arithemetic in the command window

```
>> 1 + 1
ans =
     2
>> 0.3 * 0.2
ans =
     0.06
>> (1/3 + 1/2)/(1/3 - 1/2)
ans =
     -5.0000
>> 2^64
ans =
     1.8447e+19
```

Number forms

integer	0	2	-3458	
double precision	0.3475	-0.1121		
scientific notation	1.2E-8	9E-10		
complex	i	j	2i	1-3j
other	pi	inf	NaN	nan

Setting variables

```
>> x = 2
x =
     2
>> y = 'hello'
y =
hello
```

N.B.
no variable declarations are required

Variables and workspace

Display a variable value by typing its name

All variables in use are stored in the *workspace*

Saving the workspace:

```
>> save mywork
```

Clear workspace:

```
>> clear
```

Clear command window:

```
>> clc
```

Reload data into the workspace:

```
>> load mywork
```

Mathematical functions

MATLAB contains hundreds of basic mathematical functions

We call a function using its name:

```
>> rand
ans =
      0.3458
```

Include any arguments between brackets:

```
>> sin(pi/2)
ans =
      1.0000
```

Some functions can return more than one result:

```
>> [theta,r] = cart2pol(0.5,0.5);
```

Notice the semicolon suppressed the output.

Mathematical functions

trigonometric:

```
sin,   cos,   tan,   cosec,    sec,   cot,
asin,  acos,  atan,  acosec,   asec,  acot
```

hyperbolic:

```
sinh,  cosh,  tanh,  cosech,   sech,  coth,
asinh, acosh, atanh, acosech,  asech, acoth
```

exponential:

```
sqrt,  realsqrt,  exp,  expm1m,  log,
log10,  log2,  log1p,  nthroot
```

complex:

```
real,  imag,  abs,  angle,  conj
```

integer:

```
mod(a,b), rem(a,b), round, fix, ceil, floor
```

discrete:

```
lcm(a,b),  gcd(a,b),  isprime,  primes,  factors,
factorial,  nchoosek(n,k)
```

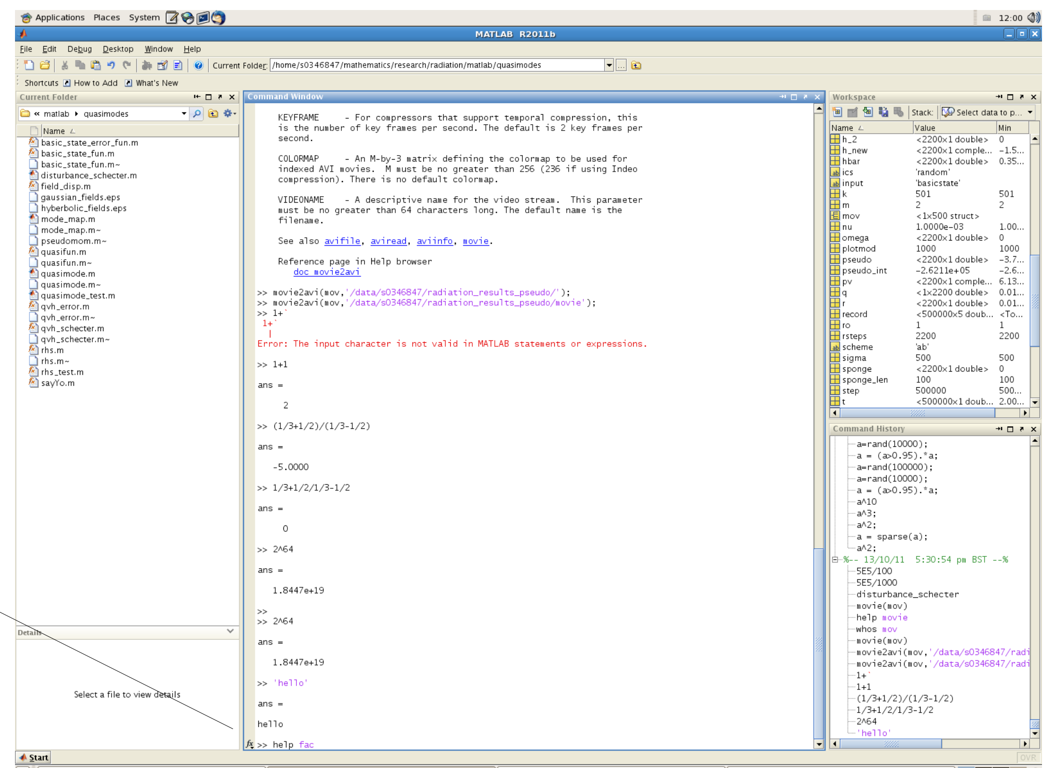
Find information on any function using `help`:

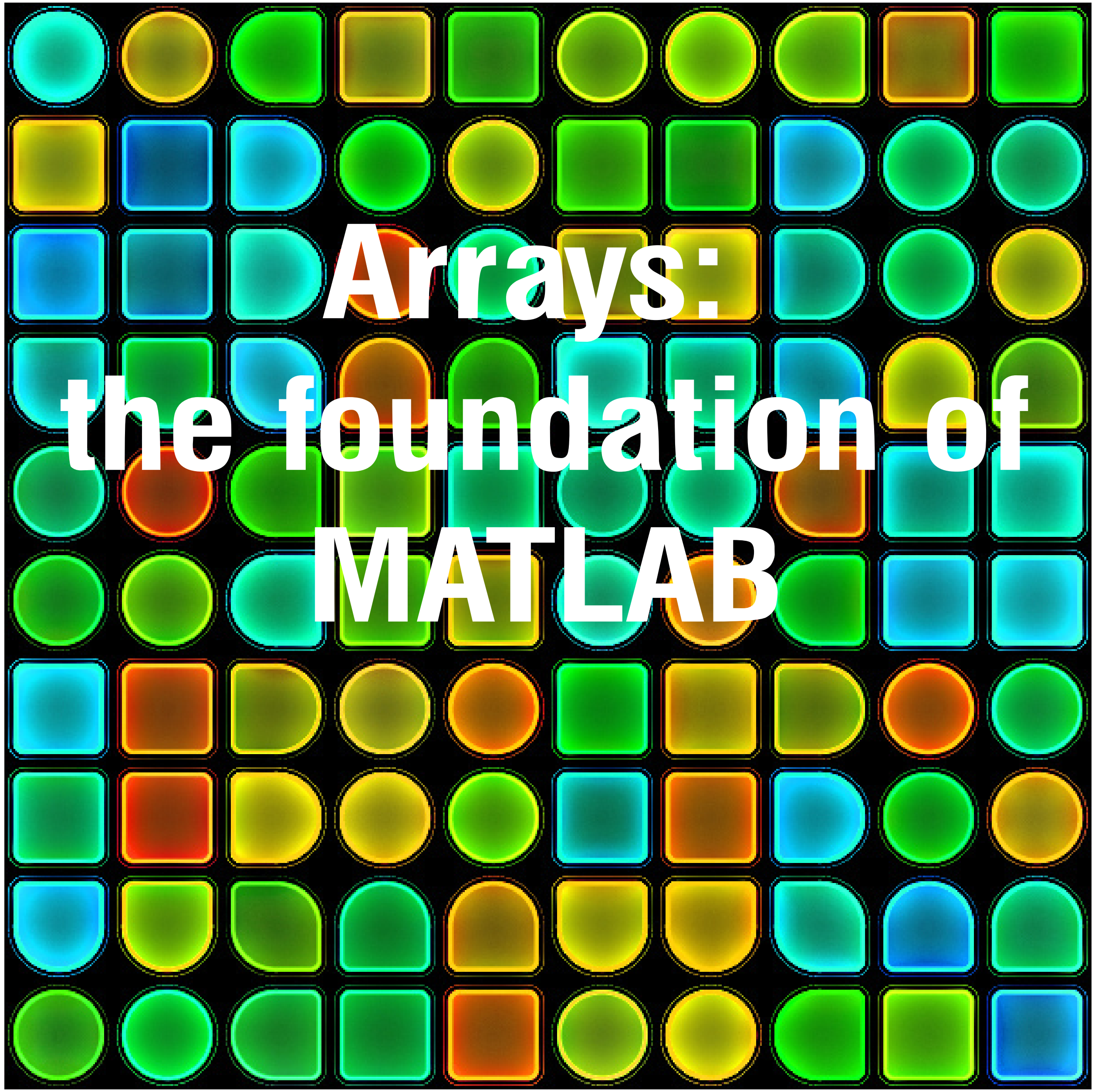
```
>> help factorial
```

or documentation using `doc`:

```
>> doc factorial
```

find a function using the f_x button




Arrays:
the foundation of
MATLAB

Basics

MATLAB is very good at dealing with arrays

A vector is a 1d array; a matrix a 2d array

Arrays with more dimensions are allowed, but uncommon

Construct a row vector like so:

```
>> a = [1 2 3 4]
a =
     1     2     3     4
```

Enter a 2-by-2 matrix like this

```
>> A = [1 2; 3 4]
A =
     1     2
     3     4
```

N.B. MATLAB is case sensitive, so a and A are different variables.

Concatenation

Note that the semicolon was used to separate two rows of the matrix

The semicolon works as a concatenation operator

It can be used to concatenate two arrays in the up-down direction:

```
>> a
a =
     1     2     3     4
>> [a;a]
ans =
     1     2     3     4
     1     2     3     4
```

The space concatenates in the left-right direction:

```
>> [A A]
ans =
     1     2     1     2
     3     4     3     4
```

Ranges

Often we require a vector of equally spaced numbers

MATLAB has *ranges* to deal with this

Declare a range with `startvalue:stopvalue`

```
>> r = 1:10
r =
     1     2     3     4     5     6     7     8     9    10
```

Ranges need not have integral spacing: use `startvalue:step:stopvalue`

```
>> r = 1:0.2:2
r =
    1.0000    1.2000    1.4000    1.6000    1.8000    2.0000
r = 2:-0.2:1
    2.0000    1.8000    1.6000    1.4000    1.2000    1.0000
```

Array manipulation

Matrix transpose:	`transpose(a)` or `a.'`
Complex conjugate:	`conj(a)`
Hermitian transpose:	`a'`
Inverse:	`inv(a)`
Left matrix division (solve **Ax=b**)	`A\b`
Right matrix division (solve **xA=b**)	`b/A`
Determinant:	`det(a)`

Left and right matrix division are much more efficient than using `inv`

Array arithmetic

For matrices * is interpreted as matrix multiplication

+ and - work for matrices

Addition of a matrix and a scalar is interpreted sensibly:

```
>> [1 2 3] + 1
ans =
     2     3     4
```

Elementwise operations

There are occasions when we wish operations to act on each element of a matrix, rather than the whole matrix.

Example: computing the square of every element of a matrix `squareMat`:

`squareMat^2` is not what is required.

To make an operator act *elementwise,* prefix it with a dot:

```
squareMat.^2
```

Another example: consider vectors `x` and `y`:

```
x./y + y.^2 -2*y.*x
```

Most of the mathematical functions covered work with arrays elementwise:

```
>> sin([0 pi/4 pi/3 pi/2 pi])
ans =
     0.0000    0.7071    0.8660    1.0000    0.0000
```

`exp` works elementwise: use `expm` for matrix exponentials

Array construction functions

MATLAB has many functions to construct common matrices:

`eye(n)`	n-by-n identity matrix
`zeros(m,n)`	m-by-n zero matrix
`ones(m,n)`	m-by-n matrix of ones
`rand(m,n)`	uniformly distributed m-by-n matrix
`randn(m,n)`	$N(0,1)$ distributed m-by-n matrix
`diag(x)`	diagonal matrix formed using vector x

and some that are less common:

`topeliz`	Topelitz matrix
`hadamard`	Hadamard matrix
`vander`	Vandermonde matrix
`hilb`	Hilbert matrix
`magic`	magic square

Array access

Vectors are accessed using a single subscript between brackets:

```
>> v =[1 3 5];
>> v(3)
ans =
     5
>>v = v.';
>> v(2)
ans =
     3
```

Matrix elements are accesed using the row and column number:

```
>> A = [1 2;3 4];
>> A(2,2)
ans =
4
```

Array access continued

The word end can be used to refer to the last element along a dimension:

```
>> x = 1:100;
>> x(end)
ans =
     100
```

Ranges can be used to access arrays:

```
>> x(1:5)
ans =
     1     2     3     4     5
```

A more complicated example:

```
>> A = [1 2 3;4 5 6;7 8 9];
>> A(2,1:end)
ans =
     4     5     6
```

Functions for array manipulation

`repmat(A,m,n)`	concatenate A m times vertically, n times horizontally
`reshape(A,m,n)`	reshape the elements of A into an m-by-n matrix
`sort(A,dim)`	sort A along the dimension `dim`
`flipud(A)`	flip A in the up-down direction
`fliplr(A)`	flip A left-to-right
`circshift(A,n)`	circularly shift elements of A down by an amount n

Functions that interrogate arrays

`sum(A,dim)`	sum elements of A along dimension dim
`prod(A,dim)`	form product of elements of A along dimension dim
`size`	return vector of dimensions of A
`length`	return length of vector
`numel`	return number of elements of an array
`nnz`	return number of elements not equal to
max	return maximum of each column

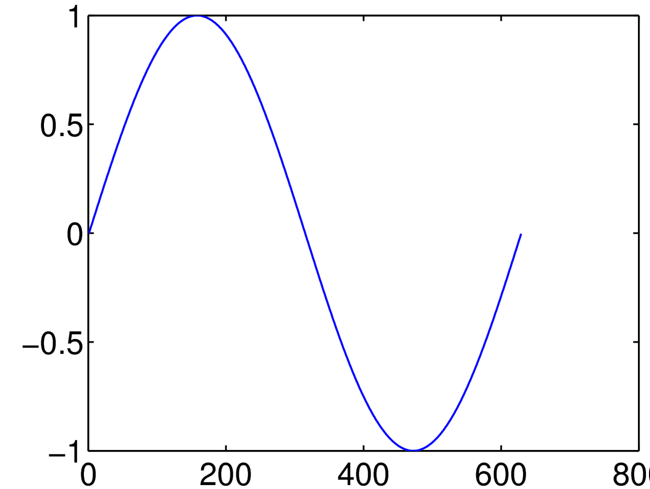
Simple plotting

The `plot` command

MATLAB has many features for producing high quality plots

Plot the values of a vector using plot:

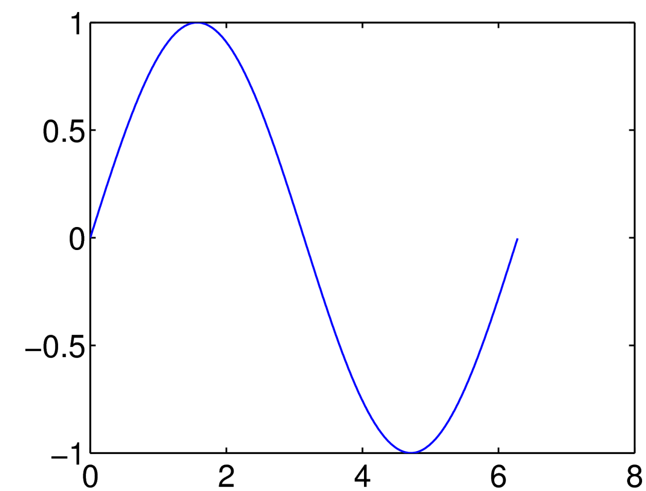
```
>> x = 0:0.01:2*pi;
>> y = sin(x);
>> plot(y);
```



By default, elements are plotted against their indices.

Plot one data set against another using plot(x,y):

```
>> plot(x,y)
```



Create a new figure window with `figure`

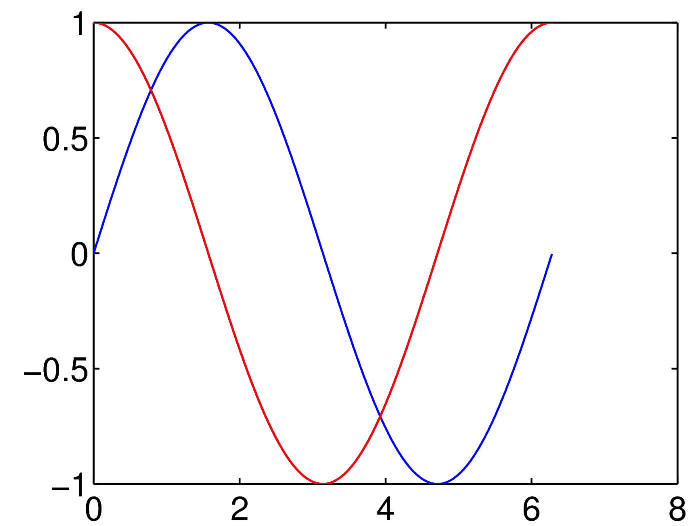
Close all figure windows with `close all`

More plot tools

The axis([xmin xmax ymin ymax]) sets the axis limits

Use `hold on` to plot multiple lines on the same figure:

```
>> hold on
>> plot(x,cos(x),'r');
```



Note we added an optional string for the line style

String controls colour, line type, and markers

Colours:

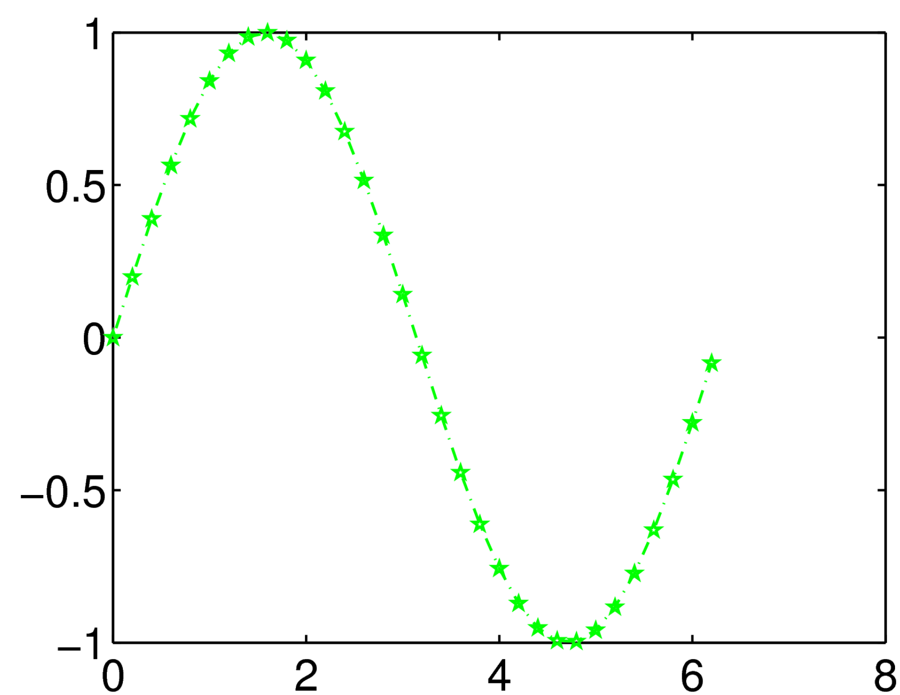
r	red
g	green
b	blue
c	cyan
m	magenta
y	yellow
b	black

Line types:

-	solid (default)
- -	dashed
:	dotted
- .	dash-dotted
none	no line; handy for markers

Markers:

+	cross
o	circle
*	star
.	dot
x	x
s	star
d	diamond
p	pentagram



Add a title using title(`'yourTitle'`)

Similarly with `xlabel` and `ylabel`

Saving plots

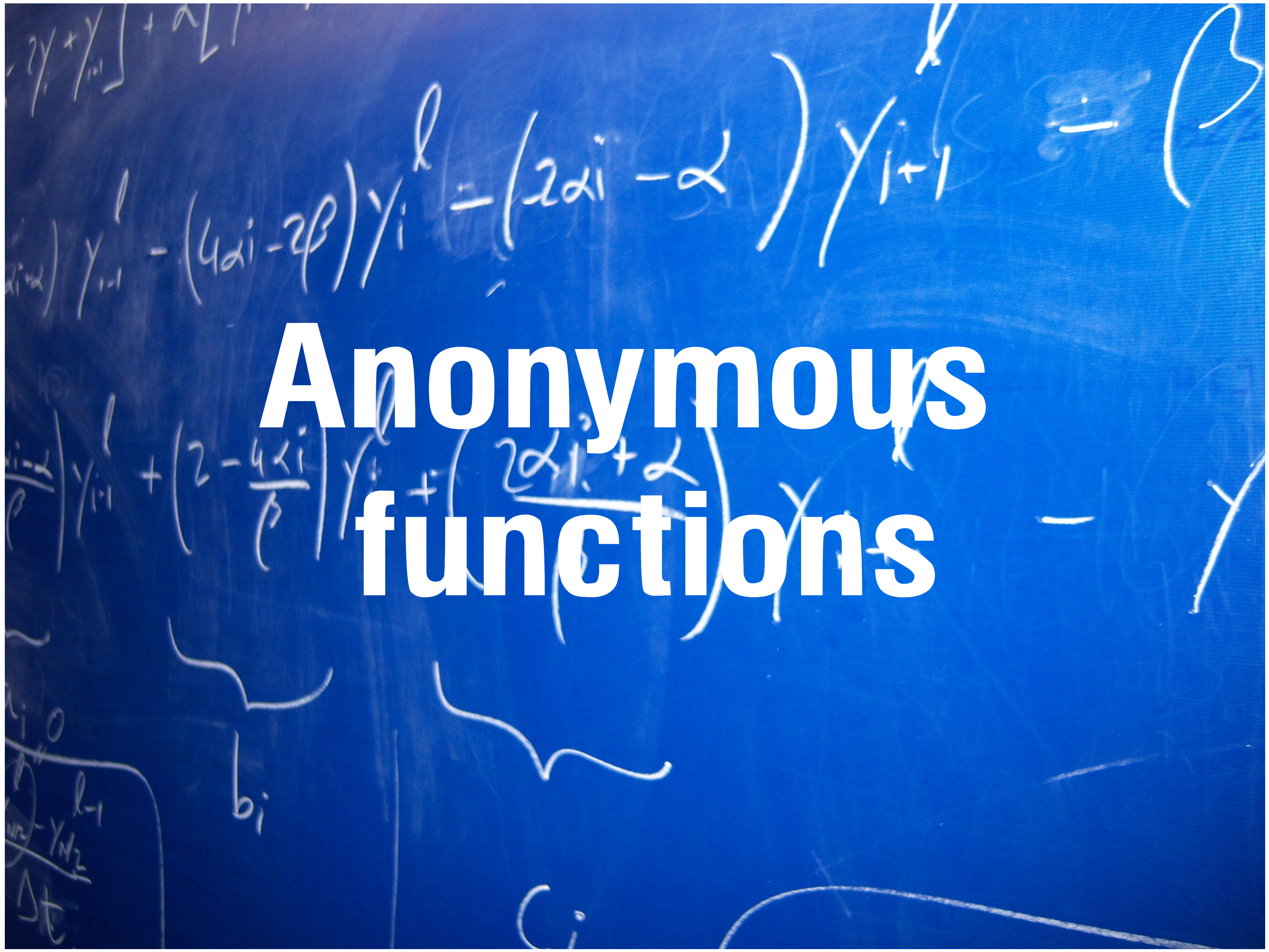
The default MATLAB figure format is a .fig file

Simply go to `Figure: File > Save As`

For LaTeX reports save as encapsulated postscript: `.eps`

Alternatively in the command window use

```
>> print -depsc myFigureName
```


Anonymous
functions

Writing your own function

Suppose we wish to study the function $f(x) = \dfrac{e^x}{1+e^{2x}}$.

MATLAB does not have a built-in function of this form.

We may write our own as an *anonymous function*

Anonymous functions have this structure:

```
myFunction = @(x,y,z,...) x+y-2*x+...
```

function name   at sign   arguments   function definition

call this function using

```
>> myFunction(1.2,4,3,...)
```

For our example:

```
>> f = @(x) exp(x)/(1+exp(2*x));
```

Anonymous functions in use: integration

Anonymous functions make integration simple.

Let us use our function f to integrate $f(x) = \frac{e^x}{1+e^{2x}}$.

The quad function is used for integration.

The integral in question is $\int_0^1 f(x)dx$

```
>> quad(f,0,1)
ans =
     0.4329
```

check using the exact result: $\int_0^1 f(x)dx = \tan^{-1}(e) - \tan^{-1}(1)$.

```
>> atan(exp(1))-atan(exp(0))
ans =
     0.4329
```

quad handles everything for us and is very accurate.

Optimization tools

Useful functions

We will consider three very useful optimization/root finding functions

`fminsearch`	find local minimum of nonlinear function
`fsolve`	solve system of nonlinear equations
`roots`	find roots of a polynomial

These functions are sufficiently quick and accurate for many problems

To use `fminsearch` and `fsolve` we need to remind ourselves about anonymous functions from lecture I

Anonymous functions and function handles

Recall how to define an anonymous function

Examples: define functions $f(x)=\sqrt{x+1}$ and $g(x)=\sin(x)+\cos(x)$:

```
f = @(x) sqrt(x+1);
g = @(x) sin(x) + cos(x);
```

We can now use f and g like normal functions

They are special variables that refer to functions called *function handles*

Another simple example: define $h(x)=\tan(x)$:

```
h = @(x) tan(x);
h = @tan;        % shorthand definition
```

The second line shows a quick way of creating a function handle to a built-in function

Using `fminsearch`

We now see why function handles are important

The function `fminsearch` takes two arguments: a function to minimise, and a starting value

Example: find a local minimum of the function $f(x)=\sin(\cos(x)-x^2)$ near $x=4$:

```
f = @(x) sin(cos(x)-x^2);     % create function handle
fminsearch(f,4);
```

Example: find a local minimum of $\cosh(x)$:

```
f = @cosh                       % create function handle
fminsearch(f,1);
```

Using `fsolve`

The function `fsolve` solves equations of the form $f(\mathbf{x}) = 0$

It is called just like `fminsearch`, using a function handle and a starting guess

Example: solve $x^3 + x^2 = \exp(-x)$:

First rewrite in the correct form

```
f = @(x) x^3 + x^2 -exp(-x)    % create function handle
fsolve(f,1)
```

Example: find a complex zero of the second Hankel function of the first kind:

Solve equation $H_2^{(1)}(x) = 0$:

```
f = @(x) besselh(2,1,x)    % create function handle
fsolve(f,1-1i);                 % use complex initial guess
```

Polynomials

Roots takes a vector of polynomial coefficients

Example: find the roots of $x^5 - 2x^2$:

```
>> roots([1 0 0 -2 0 0])  % remember constant coefficient
ans =
     0
     0
  -0.6300 + 1.0911i
  -0.6300 – 1.0911i
   1.2599
```