MATLAB Programming

Systems and Control Engineering Department
College of Electronics Engineering
Ninevah University

Omar Y. Ismael

2018

Level: Undergraduate/Second	**Ninevah University College of Electronics Engineering Academic Year: 2018-2019**	**Lecturer: Omar Y. Ismael**
Department: Systems & Control Engineering		**Email: omar_mechatronics@yahoo.com**
Subject: MATLAB Programming (SCE1313)		**No. of Hrs/Week: 2 Th + 2 Pr**

Prerequisites

The subject of the MATLAB Programming for the second stage.

Course description and objectives:

The course provides a gentle introduction to the MATLAB computing environment, and is intended for beginning users and those looking for a review. It is designed to give students a basic understanding of MATLAB. The course consists of interactive lectures and sample MATLAB problems discussed in class. No prior programming experience or knowledge of MATLAB is assumed. Concepts covered include basic use, graphical representations and tips for designing and implementing MATLAB code.

Upon successful completion of this course, the student should be able to:

- Understand the main features of the MATLAB development environment
- Use the MATLAB GUI effectively
- Design simple algorithms to solve problems
- Write simple programs in MATLAB to solve scientific and mathematical problems
- Model practical systems by using the Simulink
- Building basic GUI
- Know where to find help

References and Software:

1. "Introduction to MATLAB®" By Ho, P., Dey, S., Šćepanović, D., & Pate, 2010.
2. "What Every Engineer Should Know About MATLAB® and Simulink®" By Adrian Biran, 2010.
3. "Getting Started with MATLAB - MathWorks" By MathWorks, 2018.
4. "Essential MATLAB for Engineers and Scientists" By Brian H. Hahn, Daniel T. Valentine, 2017.

The software that is used in this course is MATLAB®.

Evaluation Profile:

Quizzes/ Homework/Attendance/Class and Lab participation/Assignments/Projects	10 %
Midterm Exam	25 %
Lab reports/ Lab examination	15 %
Final Semester Exam	50 %

Attendance:

As per University's rules and regulation.

Semester II

Week	Date	Topic
1	Feb. 19-25	Getting Started, MATLAB Basics, Help/Docs, Scripts.
2	Feb. 26-Mar. 4	Making Variables (Scalars, Vectors & Matrices), save/clear/load, Built-in Functions, Manipulating Variables, Random Numbers.
3	Mar. 5-11	Basic Plotting, Line Plots, Surface plots.
4	Mar. 12-18	User defined Functions.
5	Mar. 19-25	Flow Control, Relational Operators, if/else/elseif, for, while, find.
6	Mar. 26-Apr. 1	Vectorization, Avoiding Loops, Efficient Code. Performance Measures.
7	Apr. 2-8	Linear Algebra, Polynomials, Optimization, Symbolic Math.
8	Apr. 9-15	Differentiation/Integration, Differential Equations.
9	Apr. 16-22	Probability and Statistics, Data Structures, Cells organization, Struct Arrays.
10	Apr.23-29	Images, Figure Handles, Reading/Writing Images, display.
11	Apr. 30-May 6	Debugging, File I/O, Reading and Writing Excel Files.
12	May 7-13	Introduction to the Simulink

13	May 4-20	Simulink examples.
14	May 21-27	Graphical User Interfaces (GUI).
15	May 28-Jun. 1	Introduction to the commenly used toolboxes
16	Jun. 4-10 2nd Semester Final Year Exam	**Final Semester Examinations**
17	Jun. 11-17	
18	Jun. 18-24	
19	Jun. 25-30	**Marking and Declaration of Final Results**

MATLAB Programming

Section 1: Variables, Scripts, operations, and Plots

Outline

(1) Getting Started
(2) Scripts
(3) Making Variables
(4) Manipulating Variables
(5) Basic Plotting
(6) Line Plots
(7) Image/Surface Plots

Matlab (=Matrix laboratory) is an interactive matrix-based software package intended for complex scientific computations. It has several advantages over high-level programming languages, such as C/C++, Fortran etc.

Advantages:

- A large set of toolboxes is available. A toolbox is a collection of matlab functions specific for a subject.
- At any time, variables (results of simulations) are stored in the workspace for debugging and inspection.
- Excellent visualisation (plots) capabilities,
- Easy programming, e.g. it is not required to define variable types (unless needed). All variables are usually of type double (64 bit representation).
- Quick code development.

Disadvantages:

- Very expensive for non students, although there are some free clones, such as Octave or Sscilab that are matlab compatible (but not 100%).
- Code execution can be slow if programmed carelessly without vectorisation.
- Algorithms developed in Matlab will need to be translated in C or assembly if to be used in some hardwares.

Some of MATLAB Applications:

1- Math, Statistics and Optimization
2- Control System Design and Analysis
3- Signal Processing and Communications
4- Image Processing and Computer Vision
5- Test and Measurement
6- Computational Finance
7- Computational Biology
8- Simulation Graphics and Reporting
9- Robotics
10- System modelling and Analysis

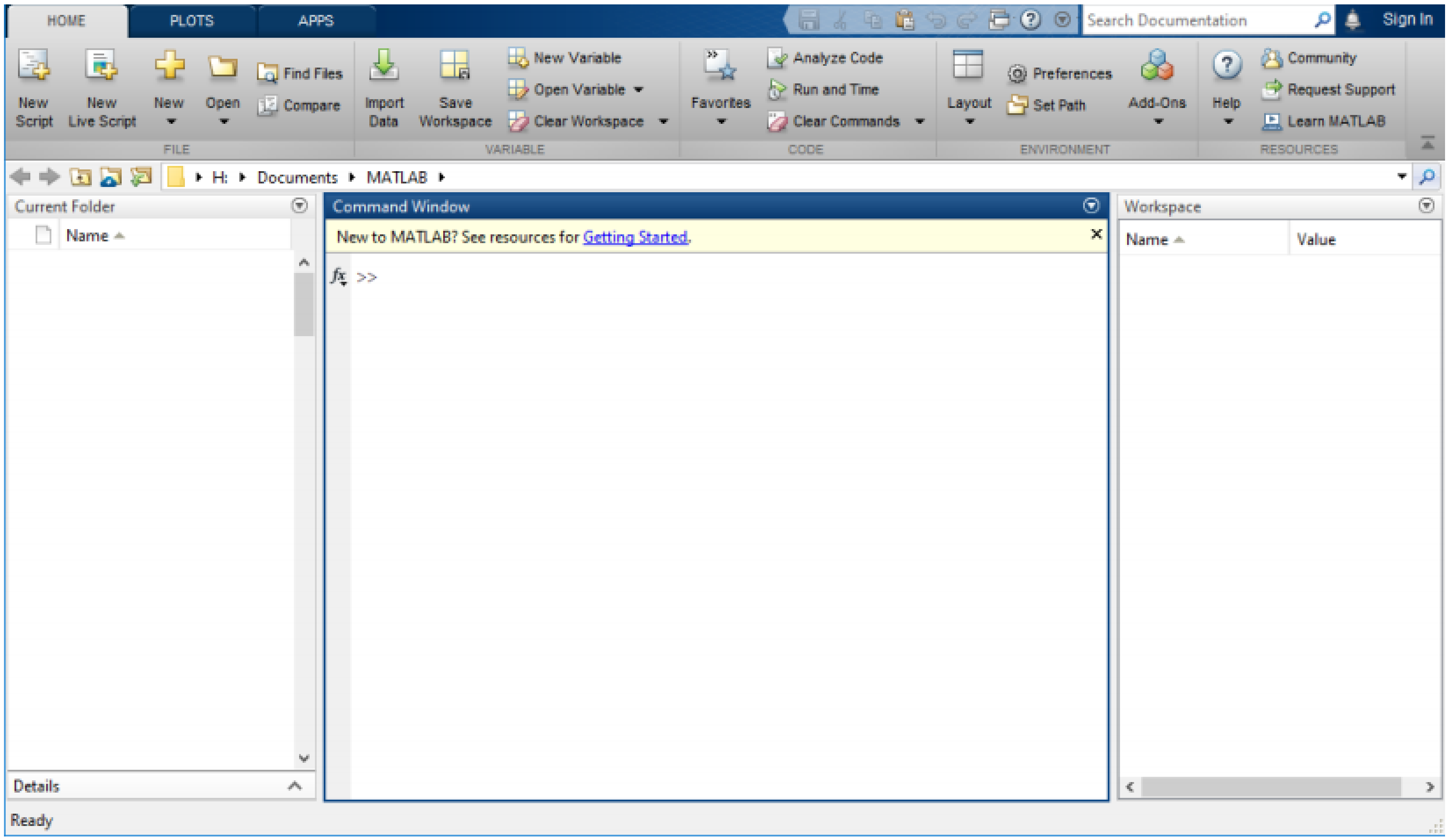
Getting Started

- Currently, Matlab is available in the PC cluster

- Open up MATLAB for Windows
  - ➢ Through the START Menu or the desktop

Desktop Basics

When you start MATLAB, the desktop appears in its default layout.



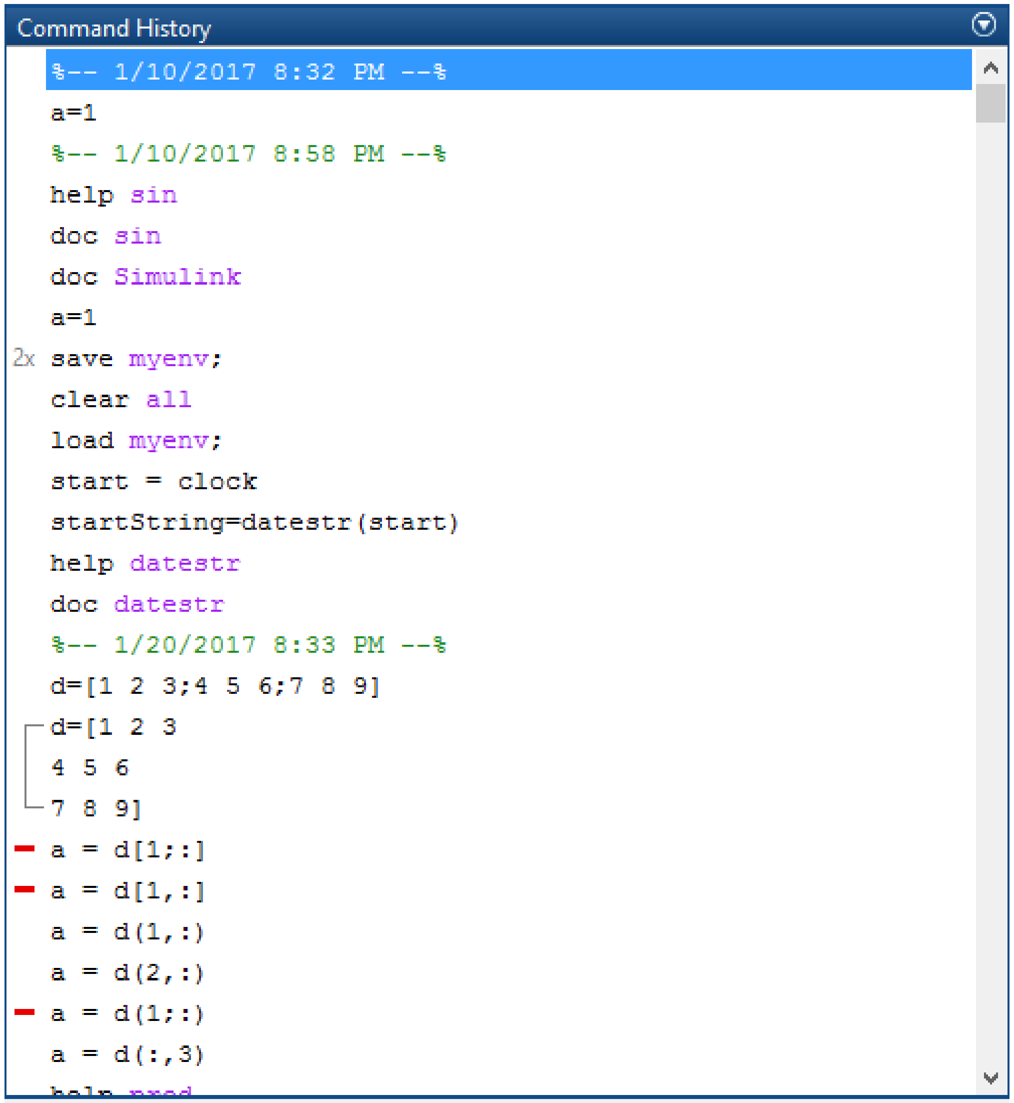
New to MATLAB? See resources for Getting Started.

```
>>
```

The desktop includes these panels:

- **Current Folder** — Access your files.
- **Command Window** — Enter commands at the command line, indicated by the prompt (>>).
- **Workspace** — Explore data that you create or import from files.

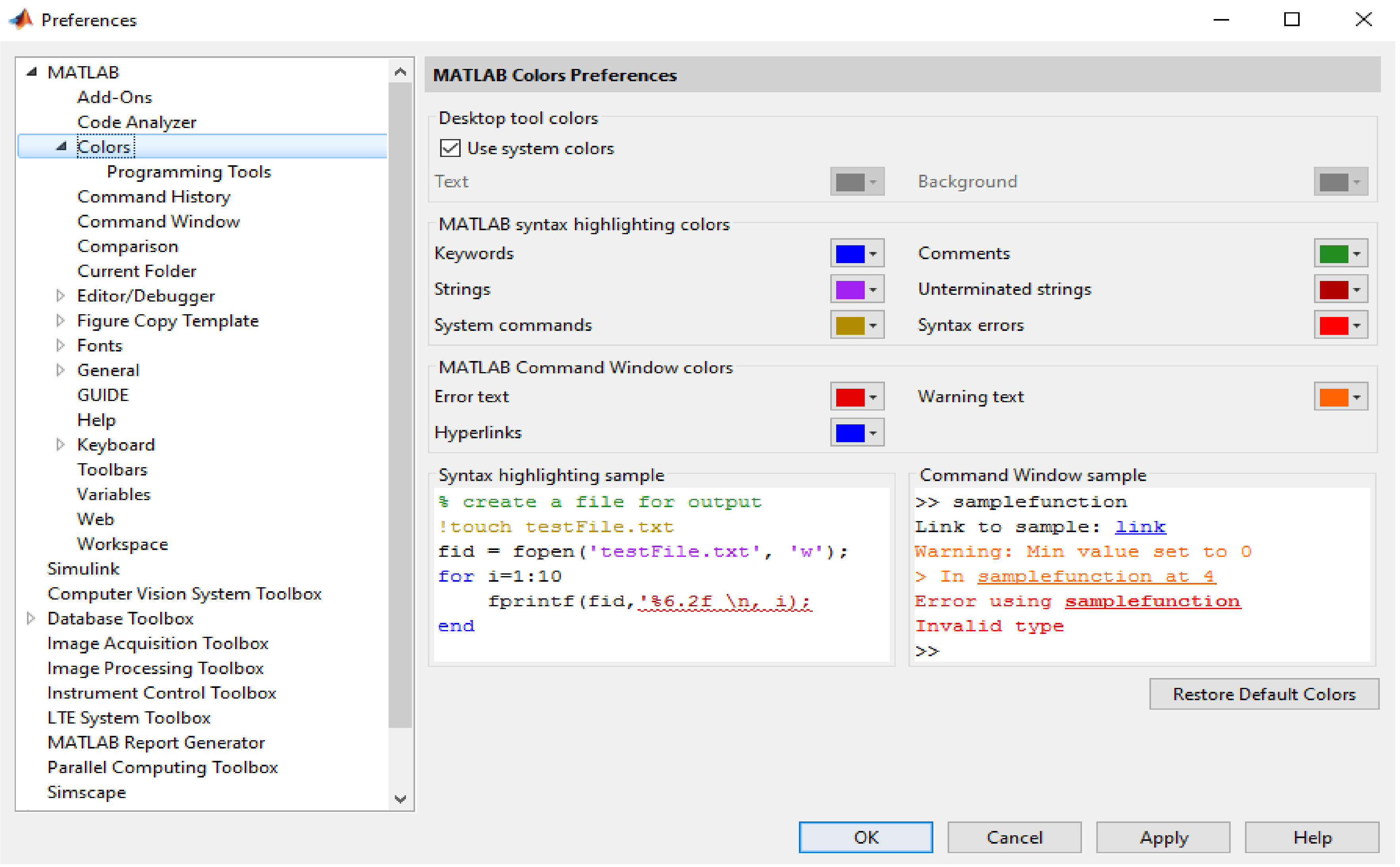
Command History
To dock or detach the Command History window, click , and then select an option. To view the Command History if it is closed: on the Home tab, in the Environment section, click Layout. Then, under Show, click Command History and select either Docked or Popup.


Command History

```
  %-- 1/10/2017 8:32 PM --%
  a=1
  %-- 1/10/2017 8:58 PM --%
  help sin
  doc sin
  doc Simulink
  a=1
2x save myenv;
  clear all
  load myenv;
  start = clock
  startString=datestr(start)
  help datestr
  doc datestr
  %-- 1/20/2017 8:33 PM --%
  d=[1 2 3;4 5 6;7 8 9]
  d=[1 2 3
  4 5 6
  7 8 9]
  a = d[1;:]
  a = d[1,:]
  a = d(1,:)
  a = d(2,:)
  a = d(1;:)
  a = d(:,3)
```

Customization

Home ------- Preferences



MATLAB Colors Preferences

Desktop tool colors

☑ Use system colors

Text	▨	Background	▨

MATLAB syntax highlighting colors

Keywords	🟦	Comments	🟩
Strings	🟪	Unterminated strings	🟥
System commands	🟨	Syntax errors	🟥

MATLAB Command Window colors

Error text	🟥	Warning text	🟧
Hyperlinks	🟦		

Syntax highlighting sample

```
% create a file for output
!touch testFile.txt
fid = fopen('testFile.txt', 'w');
for i=1:10
    fprintf(fid,'%6.2f \n, i);
end
```

Command Window sample

```
>> samplefunction
Link to sample: link
Warning: Min value set to 0
> In samplefunction at 4
Error using samplefunction
Invalid type
>>
```

MATLAB Basics

- MATLAB can be thought of as a super-powerful graphing calculator
  - ➢ Remember the TI-Nspire CX CAS Graphing Calculator?
  - ➢ With many more buttons (built-in functions)

- In addition it is a programming language
  - ➢ MATLAB is an interpreted language, like Java
  - ➢ Commands normally executed line by line
  - ➢ Parallel Pool (parallel language features such as "parfor")

Help/Docs

- `help`
  - ➢ **The most** important function for learning MATLAB on your own
- To get info on how to use a function:
  - » `help sin`
    - ➢ Help lists related functions at the bottom and links to the doc
- To get a nicer version of help with examples and easy-to-read descriptions:
  - » `doc sin`
- To search for a function by specifying keywords:
  - » `doc` + Search tab

Outline

(1) Getting Started

(2) Scripts

(3) Making Variables

(4) Manipulating Variables

(5) Basic Plotting

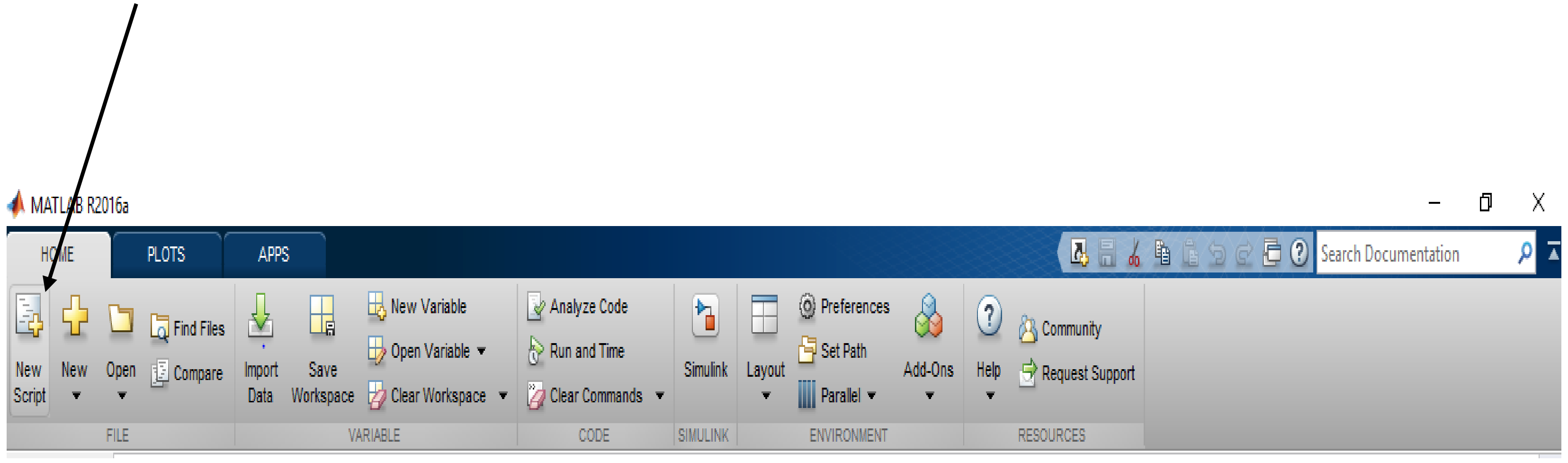
Scripts: Overview

- Scripts are
  - collection of commands executed in sequence
  - written in the MATLAB editor
  - saved as MATLAB files (.m extension)

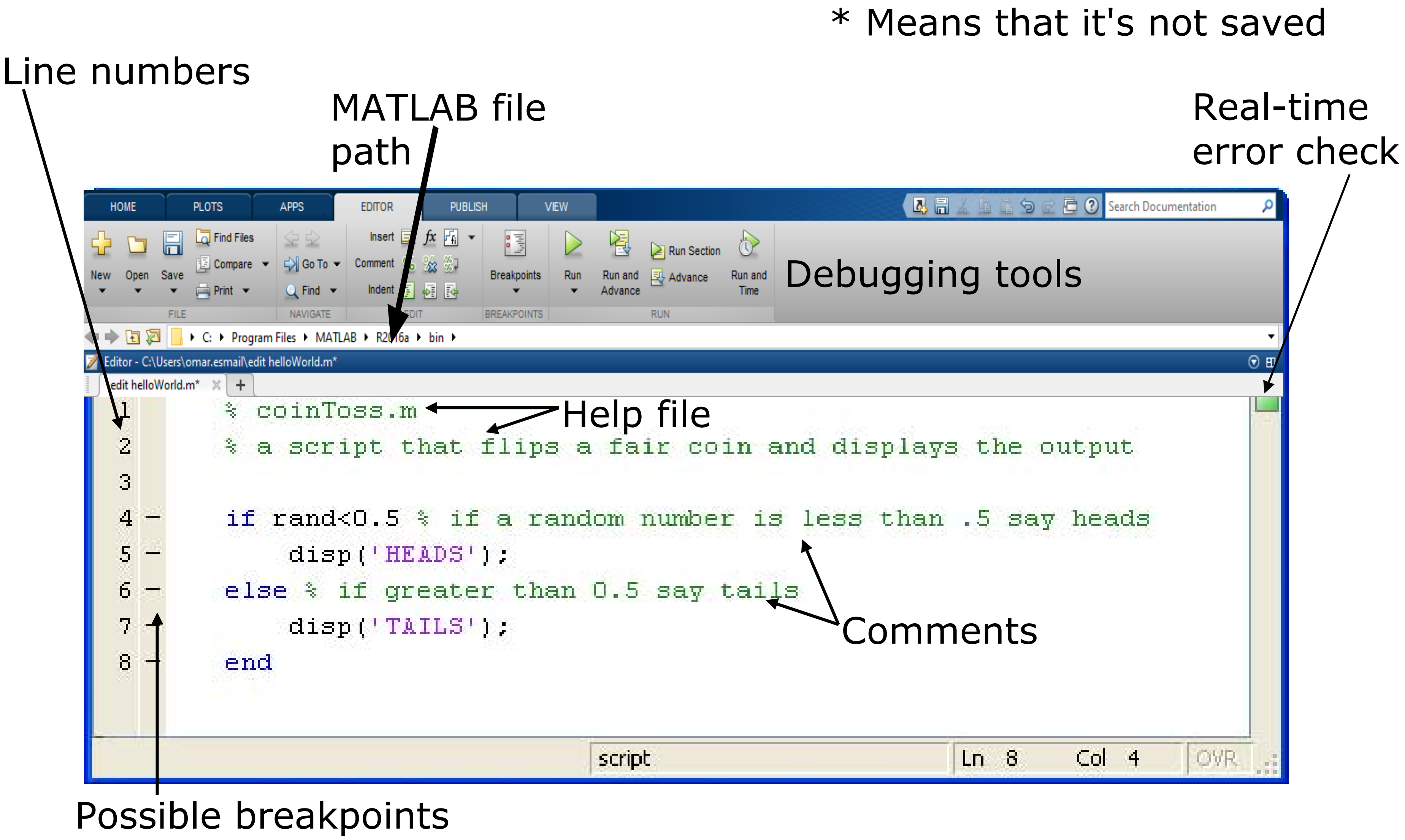
- To create an MATLAB file from command-line

```
» edit helloWorld.m
```

- or click



Scripts: the Editor



```
% coinToss.m
% a script that flips a fair coin and displays the output

if rand<0.5 % if a random number is less than .5 say heads
    disp('HEADS');
else % if greater than 0.5 say tails
    disp('TAILS');
end
```

Scripts: Some Notes

- **COMMENT!**
  - Anything following a **%** is seen as a comment
  - The first contiguous comment becomes the script's help file
  - Comment thoroughly to avoid wasting time later

- Note that scripts are somewhat static, since there is no input and no explicit output

- All variables created and modified in a script exist in the workspace even after it has stopped running

Exercise: Scripts

Make a `helloWorld` script

- When run, the script should display the following text:

```
Hello World!
I am going to learn MATLAB!
```

- **Hint:** use `disp` to display strings. Strings are written between single quotes, like `'This is a string'`

Exercise: Scripts

Make a `helloWorld` script

- When run, the script should display the following text:

```
Hello World!
I am going to learn MATLAB!
```

- **Hint:** use `disp` to display strings. Strings are written between single quotes, like `'This is a string'`

- Open the editor and save a script as helloWorld.m. This is an easy script, containing two lines of code:

```
» % helloWorld.m
» % my first hello world program in MATLAB

» disp('Hello World!');
» disp('I am going to learn MATLAB!');
```

Outline

(1) Getting Started

(2) Scripts

(3) Making Variables

(4) Manipulating Variables

(5) Basic Plotting

Variable Types

- MATLAB is a weakly typed language
  - ➢No need to initialize variables!
- MATLAB supports various types, the most often used are
  - » `3.84`
    - ➢64-bit double (default)
  - » `‘a’`
    - ➢16-bit char
- Most variables you’ll deal with will be vectors or matrices of doubles or chars
- Other types are also supported: complex, symbolic, 16-bit and 8 bit integers, etc.

Naming variables

- To create a variable, simply assign a value to a name:
  - » `var1=3.14`
  - » `myString='hello world'`

- Variable names
  - ➢ first character must be a LETTER
  - ➢ after that, any combination of letters, numbers and _
  - ➢ CASE SENSITIVE! (`var1` is different from `Var1`)

- Built-in variables. Don't use these names!
  - ➢ `i` and `j` can be used to indicate complex numbers
  - ➢ `pi` has the value 3.1415926…
  - ➢ `ans` stores the last unassigned value (like on a calculator)
  - ➢ `Inf` and `-Inf` are positive and negative infinity
  - ➢ `NaN` represents 'Not a Number'

Scalars

- A variable can be given a value explicitly

```
» a = 10
```

  - ➢ shows up in workspace!

- Or as a function of explicit values and existing variables

```
» c = 1.3*45-2*a
```

- To suppress output, end the line with a semicolon

```
» cooldude = 13/3;
```

EXERCISES

Give values to variables `a` and `b` on the command line, e.g., `a = 3` and `b = 5`. Write some statements to find the sum, difference, product and quotient of `a` and `b`.

Assign a value to the variable `x` on the command line, e.g., `x = 4 * pi^2`. What is the square root of `x`? What is the cosine of the square root of `x`? Assign a value to the variable `y` on the command line as follows: `y = -1`. What is the square root of `y`? Show that the answer is

```
ans =
         0 + 1.0000i
```

Yes, MATLAB deals with complex numbers (not just real numbers). Hence the symbol `i` should not be used as an index or as a variable name. By default, it is equal to the square root of -1. (Also, when necessary, `j` is used in MATLAB as a symbol for $\sqrt{-1}$. Hence, it also should not be used as an index or as a variable name.) Give an example of how you have used complex numbers in your studies of mathematics and the sciences up to this point in your education.

Arrays

- Like other programming languages, arrays are an important part of MATLAB
- Two types of arrays

(1) matrix of numbers (either double or complex)

(2) cell array of objects (more advanced data structure)

MATLAB makes vectors easy!
That's its power!

Row Vectors

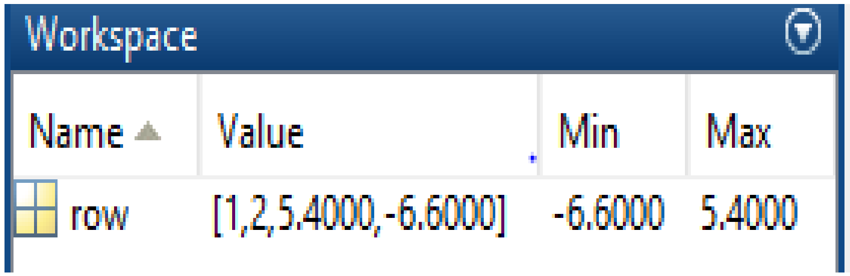
- Row vector: comma or space separated values between brackets

```
» row = [1 2 5.4 -6.6]
» row = [1, 2, 5.4, -6.6];
```

- Command window:

```
>> row=[1 2 5.4 -6.6]

row =

    1.0000    2.0000    5.4000   -6.6000
```

- Workspace:



Workspace

Name	Value	Min	Max
row	[1,2,5.4000,-6.6000]	-6.6000	5.4000

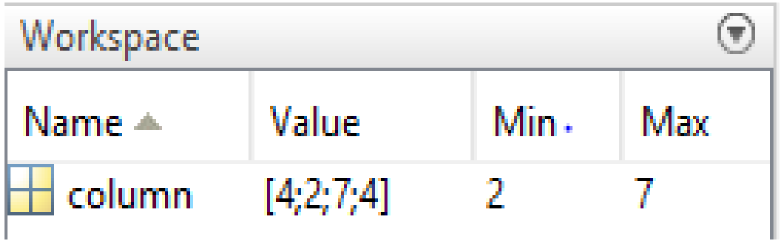
Column Vectors

- Column vector: semicolon separated values between brackets
  - `» column = [4;2;7;4]`

- Command window:

```
>> column=[4;2;7;4]

column =

     4
     2
     7
     4
```

- Workspace:



Workspace

Name	Value	Min	Max
column	[4;2;7;4]	2	7

size & length

- You can tell the difference between a row and a column vector by:
  - Looking in the workspace
  - Displaying the variable in the command window
  - Using the size function

```
>> size(row)

ans =

     1     4
```

```
>> size(column)

ans =

     4     1
```

- To get a vector's length, use the length function

```
>> length(row)

ans =

     4
```

```
>> length(column)

ans =

     4
```

Matrices

- Make matrices like vectors

- Element by element

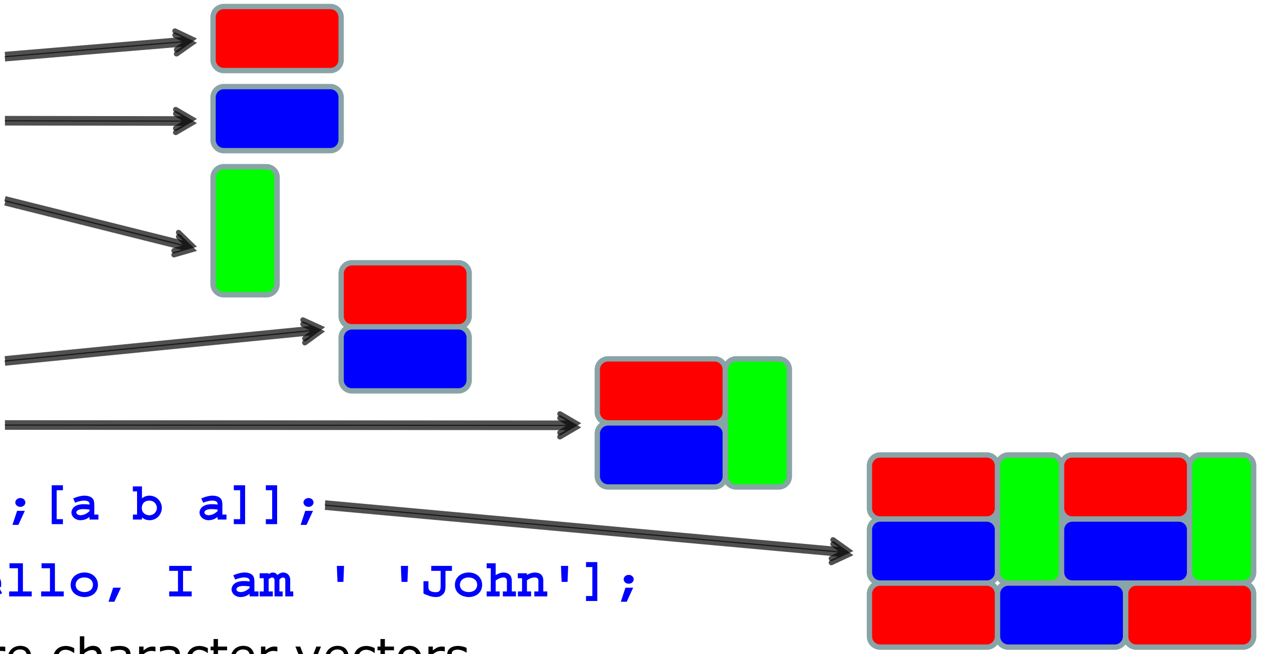
```
» a= [1 2;3 4];
```

$$a = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- By concatenating vectors or matrices (dimension matters)

```
» a = [1 2];
» b = [3 4];
» c = [5;6];

» d = [a;b];
» e = [d c];
» f = [[e e];[a b a]];
» str = ['Hello, I am ' 'John'];
```

  - Strings are character vectors



Outline

(1) Getting Started
(2) Scripts
(3) Making Variables
(4) Manipulating Variables
(5) Basic Plotting

Basic Scalar Operations

- Arithmetic operations (**+**,**-**,*****,**/**)

```
» 7/45
» (1+i)*(2+i)
» 1 / 0
» 0 / 0
```

- Exponentiation (**^**)

```
» 4^2
» (3+4*j)^2
```

- Complicated expressions, use parentheses

```
» ((2+3)*3)^0.1
```

- Multiplication is NOT implicit given parentheses

```
» 3(1+0.7) gives an error
```

- To clear command window

```
» clc
```

Table : Arithmetic operations

Operation	**Operator**	**Example**
Addition	+	2 + 2
Subtraction	-	4 -2
Multiplication	*	2*2
Division	/	4/2
Square root	sqrt	`sqrt(4)`
Power	^	`2^3`

To perform arithmetical operations in MATLAB we use the symbols listed in Table . When the same number must be used several times, it is recommended to store it as a constant. Thus, we spare work and avoid input errors. We can retrieve the names of the stored constants by typing `who`, and obtain more details with `whos`. All this information can be also found in the *Workspace* window

Table Arithmetic Operations Between Two Scalars

Operation	Algebraic form	MATLAB
Addition	$a + b$	`a + b`
Subtraction	$a - b$	`a - b`
Multiplication	$a \times b$	`a * b`
Right division	a/b	`a / b`
Left division	b/a	`a \ b`
Power	a^b	`a ^ b`

Table Precedence of Arithmetic Operations

Precedence	Operator
1	Parentheses (round brackets)
2	Power, left to right
3	Multiplication and division, left to right
4	Addition and subtraction, left to right

Built-in Functions

- MATLAB has an **enormous** library of built-in functions

- Call using parentheses – passing parameter to function

```
» sqrt(2)
» log(2), log10(0.23)
» cos(1.2), atan(-.8)
» exp(2+4*i)
» round(1.4), floor(3.3), ceil(4.23)
» angle(i); abs(1+i);
```

Table : Trigonometric and inverse trigonometric functions

Function	**Trigonometric**		**Inverse**	
Argument	Degrees	Radians	Degrees	Radians
sine (sin)	`sind`	`sin`	`asind`	`asin`
cosine (cos)	`cosd`	`cos`	`acosd`	`acos`
tangent (tan)	`tand`	`tan`	`atand`	`atan` `atan2`
cotangent (cot)	`cotd`	`cot`	`acotd`	`acot`
secant (sec)	`secd`	`sec`	`asecd`	`asec`
cosecant (csc)	`cscd`	`csc`	`acscd`	`acsc`

The set of trigonometric and inverse trigonometric functions provided by MATLAB is listed in table . For each function there is a variant that accepts arguments in radians, and one that accepts arguments in degrees.

Exponential and logarithmic functions can be calculated with the functions listed in Table

Table : Exponentials and logarithms

Operation	**Command**	**Example**	**Meaning**
Exponential function	`exp`	`exp(7)`	e^7
Natural logarithm	`log`	`log(2)`	$\log_e 7$
Common logarithm	`log10`	`log10(10)`	$\log_{10} 10$
Base 2 logarithm	`log2`	`log2(8)`	$\log2(8)$

As an example,
consider the parallelepiped shown in the Figure.
Its breadth is 40 mm, depth, 30 mm, and height, 60 mm.
We use the initials of these dimensions as names of constants and allocate them the above-mentioned values.
Next, we calculate the area of the base, the volume of the parallelepiped and the area of the developed surface as follows:

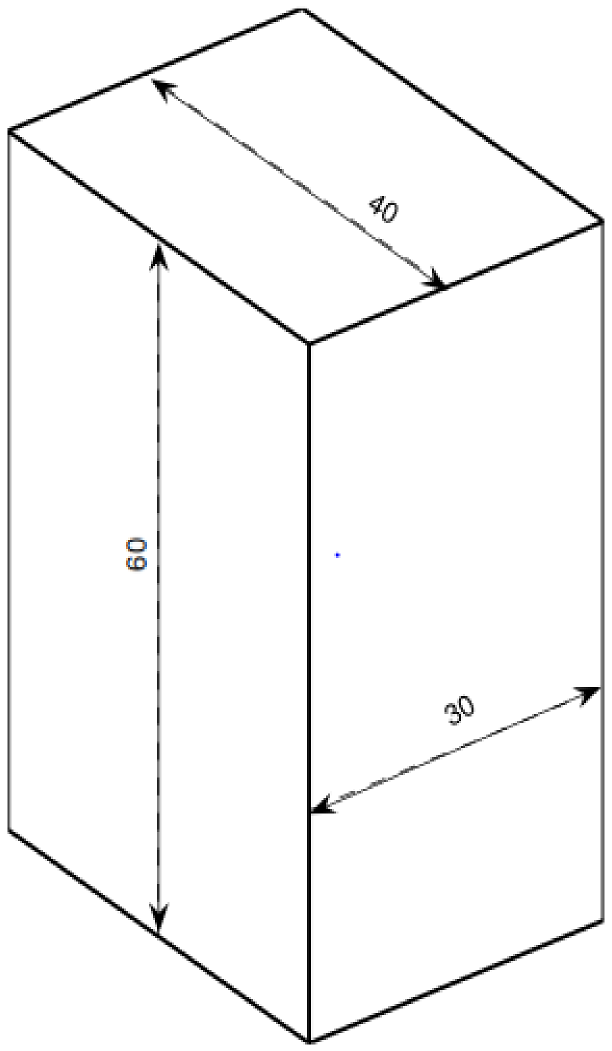


FIGURE: A parallelepiped

Solution:

```
≫ B = 40;
≫ D = 30;
≫ H = 60;
≫ BaseArea = B*D
BaseArea =
 1200
≫ Volume = BaseArea*H
Volume =
 72000
≫ Developed = 2*BaseArea + 2*B*H + 2*D*H
Developed =
 10800
```

Transpose

- The transpose operators turns a column vector into a row vector and vice versa

```
» a = [1 2 3 4+i]
» transpose(a)
» a'
» a.'
```

- The `'` gives the Hermitian-transpose, i.e. transposes and conjugates all complex numbers

- For vectors of real numbers `.'` and `'` give same result

Addition and Subtraction

- Addition and subtraction are element-wise; sizes must match (unless one is a scalar):

$$\begin{array}{r} \begin{bmatrix}12 & 3 & 32 & -11\end{bmatrix} \\ +\begin{bmatrix}2 & 11 & -30 & 32\end{bmatrix} \\ \hline =\begin{bmatrix}14 & 14 & 2 & 21\end{bmatrix} \end{array} \qquad \begin{bmatrix}12\\1\\-10\\0\end{bmatrix} - \begin{bmatrix}3\\-1\\13\\33\end{bmatrix} = \begin{bmatrix}9\\2\\-23\\-33\end{bmatrix}$$

- The following would give an error

```
» c = row + column
```

- Use the transpose to make sizes compatible

```
» c = row’ + column
» c = row + column’
```

- Can sum up or multiply elements of vector

```
» s=sum(row);
» p=prod(row);
```

Element-Wise Functions

- All the functions that work on scalars also work on vectors

```
» t = [1 2 3];
» f = exp(t);
```

  - is the same as

```
» f = [exp(1) exp(2) exp(3)];
```

- If in doubt, check a function’s help file to see if it handles vectors elementwise

- Operators (`* / ^`) have two modes of operation
  - element-wise
  - standard

Operators: element-wise

- To do element-wise operations, use the dot: `.` (`.*`, `./`, `.^`). BOTH dimensions must match (unless one is scalar)!
  - » `a=[1 2 3];b=[4;2;1];`
  - » `a.*b, a./b, a.^b` → `all errors`
  - » `a.*b', a./b', a.^(b')` → `all valid`

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} .* \begin{bmatrix}4\\2\\1\end{bmatrix} = ERROR$$

$$\begin{bmatrix}1\\2\\3\end{bmatrix} .* \begin{bmatrix}4\\2\\1\end{bmatrix} = \begin{bmatrix}4\\4\\3\end{bmatrix}$$

$$3\times1 .* 3\times1 = 3\times1$$

$$\begin{bmatrix}1 & 1 & 1\\2 & 2 & 2\\3 & 3 & 3\end{bmatrix} .* \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix} = \begin{bmatrix}1 & 2 & 3\\2 & 4 & 6\\3 & 6 & 9\end{bmatrix}$$

$$3\times3 .* 3\times3 = 3\times3$$

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} .\wedge 2 = \begin{bmatrix}1^2 & 2^2\\3^2 & 4^2\end{bmatrix}$$

Can be any dimension

Operators: standard

- Multiplication can be done in a standard way or element-wise
- Standard multiplication (*****) is either a dot-product or an outer-product
  - Remember from linear algebra: inner dimensions must MATCH!!
- Standard exponentiation (**^**) can only be done on square matrices or scalars
- Left and right division (**/ \**) is same as multiplying by inverse
  - Our recommendation: just multiply by inverse (more on this later)

$$\begin{bmatrix}1 & 2 & 3\end{bmatrix} * \begin{bmatrix}4\\2\\1\end{bmatrix} = 11$$

$$1\times 3 * 3\times 1 = 1\times 1$$

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} \wedge 2 = \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} * \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}$$

Must be square to do powers

$$\begin{bmatrix}1 & 1 & 1\\2 & 2 & 2\\3 & 3 & 3\end{bmatrix} * \begin{bmatrix}1 & 2 & 3\\1 & 2 & 3\\1 & 2 & 3\end{bmatrix} = \begin{bmatrix}3 & 6 & 9\\6 & 12 & 18\\9 & 18 & 27\end{bmatrix}$$

$$3\times 3 * 3\times 3 = 3\times 3$$

Table : Array operations

Operation	**Operator**	**Example**
Addition	+	A + B
Subtraction	-	A - B
Multiplication	.*	A.*B
Division	./	A./B
Power	.^	A.^2

Exercises

Use MATLAB array operations to do the following:

Add 1 to each element of the vector `[2 3 -1]`.

Multiply each element of the vector `[1 4 8]` by 3.

Find the array product of the two vectors `[1 2 3]` and `[0 -1 1]`. (Answer: `[0 -2 3]`)

Square each element of the vector `[2 3 1]`.

Automatic Initialization

- Initialize a vector of **ones**, **zeros**, or **rand**om numbers
  - » **`o=ones(1,10)`**
    - ➢ row vector with 10 elements, all 1
  - » **`z=zeros(23,1)`**
    - ➢ column vector with 23 elements, all 0
  - » **`r=rand(1,45)`**
    - ➢ row vector with 45 elements (uniform [0,1])
  - » **`n=nan(1,69)`**
    - ➢ row vector of NaNs (useful for representing uninitialized variables)

The general function call is:

```
    var=zeros(M,N);
```

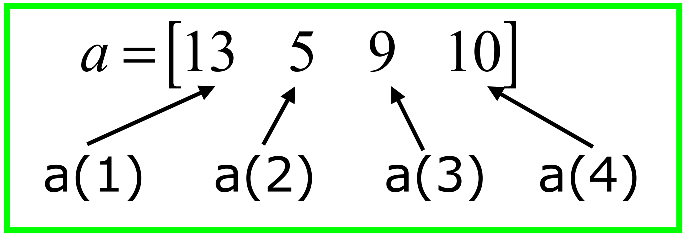
Number of rows   Number of columns

Automatic Initialization

- To initialize a linear vector of values use **linspace**
  - » `a=linspace(0,10,5)`
    - ➢ starts at 0, ends at 10 (inclusive), 5 values
- Can also use colon operator (**:**)
  - » `b=0:2:10`
    - ➢ starts at 0, increments by 2, and ends at or before 10
    - ➢ increment can be decimal or negative
  - » `c=1:5`
    - ➢ if increment isn't specified, default is 1
- To initialize logarithmically spaced values use **logspace**
    - ➢ similar to **linspace**, but see **help**

Vector Indexing

- MATLAB indexing starts with **1**, not **0**
- a(n) returns the nth element



- The index argument can be a vector. In this case, each element is looked up individually, and returned as a vector of the same size as the index vector.

```
» x=[12 13 5 8];
» a=x(2:3);      →  a=[13 5];
» b=x(1:end-1);  →  b=[12 13 5];
```

Matrix Indexing

- Matrices can be indexed in two ways
  - using **subscripts** (row and column)
  - using linear **indices** (Recommended if matrix is a vector)
- Matrix indexing: subscripts or linear indices

$$\begin{array}{r} b(1,1) \longrightarrow \\ b(2,1) \longrightarrow \end{array} \begin{bmatrix} 14 & 33 \\ 9 & 8 \end{bmatrix} \begin{array}{l} \longleftarrow b(1,2) \\ \longleftarrow b(2,2) \end{array}$$

$$\begin{array}{r} b(1) \longrightarrow \\ b(2) \longrightarrow \end{array} \begin{bmatrix} 14 & 33 \\ 9 & 8 \end{bmatrix} \begin{array}{l} \longleftarrow b(3) \\ \longleftarrow b(4) \end{array}$$

- Picking submatrices

```
» A = rand(5) % shorthand for 5x5 matrix
» A(1:3,1:2) % specify contiguous submatrix
» A([1 5 3], [1 4]) % specify rows and columns
```

Advanced Indexing 1

- To select rows or columns of a matrix, use the **:**

$$c = \begin{bmatrix} 12 & 5 \\ -2 & 13 \end{bmatrix}$$

```
» d=c(1,:);  ──────────►  d=[12 5];
» e=c(:,2);  ──────────►  e=[5;13];
» c(2,:)=[3 6];   %replaces second row of c
```

Advanced Indexing 2

- MATLAB contains functions to help you find desired values within a vector or matrix

```
» vec = [5 3 1 9 7]
```

- To get the minimum value and its index:

```
» [minVal,minInd] = min(vec);
```

  - ➢ **max** works the same way

- To find any the indices of specific values or ranges

```
» ind = find(vec == 9);
» ind = find(vec > 2 & vec < 6);
```

  - ➢ **find** expressions can be very complex, more on this later

- To convert between subscripts and indices, use **ind2sub**, and **sub2ind**. Look up **help** to see how to use them.

EXAMPLE: Spring in series

1. the force acting on each spring is F;
2. the total compression, x, is the sum of the individual compressions of the three springs:

$$x = \frac{F}{k_1} + \frac{F}{k_2} + \frac{F}{k_3}$$

Assume the following data for the Figure:

$$F = 0.5\text{N},\ \ k_1 = 0.05\ \ \text{N mm}^{-1},\ \ k_2 = 0.08\ \ \text{N mm}^{-1},\ \ k_3 = 0.04\ \ \text{N mm}^{-1}$$

In MATLAB, calculate the equivalent spring constant and the individual compressions of the three springs with the commands

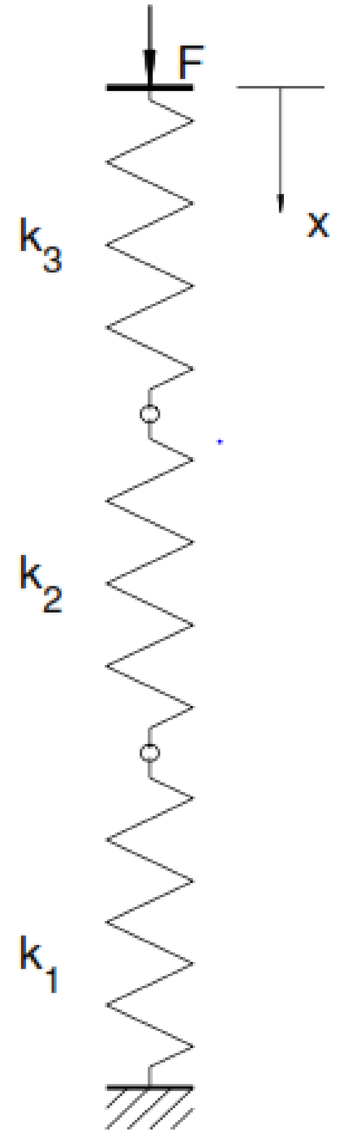


FIGURE: Three helicoidal springs in series

Solution:

```
≫ F = 0.05;
≫ k = [ 0.05 0.08 0.04 ];
≫ Keq = 1/sum(1./k)
Keq =
 0.0174
x = F/Keq
 2.8750
```

```
≫ xi = F./k
xi =
 1.0000 0.6250 1.2500
```

```
≫ sum(xi) - x
ans =
 0
```

Example:

What happens if the resistors are connected in parallel, as in Figure In this case the voltage across any resistor is equal to E, while the current intensities in the three resistors are equal to

$$E/R_1,\ E/R_2, E/R_3$$

According to Kirchoff's current law, the total current, i, through E equals the sum of the three currents and we can write

$$i = E/R_1 + E/R_2 + E/R3$$

If we note by R_{eq} the equivalent resistance that can replace the three given resistors, we can write

$$i = \frac{E}{R_{eq}} = E/R_1 + E/R_2 + E/R3$$

The obvious conclusion is

$$\frac{1}{R_{eq}} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}$$

It is easy to generalize the above result and state that *the equivalent resistance of a number of resistors connected in parallel is equal the the harmonic mean of the individual resistances.*

For a numerical example we use the same values as for the series connection of resistors. Then, we calculate the equivalent resistance and the total current with the MATLAB commands

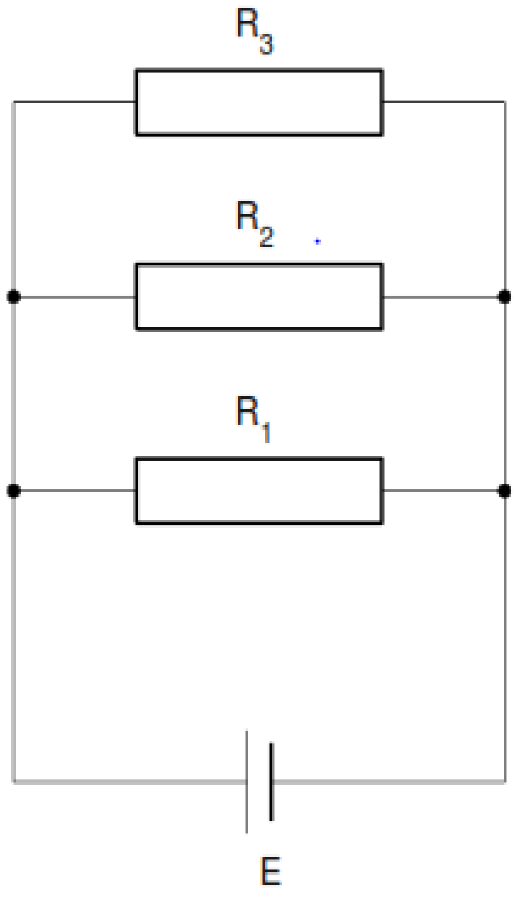


FIGURE: Three resistors connected in parallel

Solution:

```
≫ R = [ 100 200 300 ];
≫ E = 12;
≫ Req = 1/sum(1./R)
Req =
 54.5455
≫ it = E/Req
it =
 0.2200
```

We can calculate the currents through the three resistors

```
≫ i = E./R
i =
 0.1200 0.0600 0.0400
```

and check that their sum equals the total current

```
≫ sum(i)
ans =
 0.2200
```

Random Numbers

- MATLAB contains the common distributions built in
  - » **`rand`**
    - ➢ draws from the uniform distribution from 0 to 1
  - » **`randn`**
    - ➢ draws from the standard normal distribution (Gaussian)
  - » **`random`**
    - ➢ can give random numbers from many more distributions
    - ➢ see **doc random** for help
    - ➢ the docs also list other specific functions
- You can also seed the random number generators
  - »
    ```
    rand('state',0); rand(1); rand(1);
    rand('state',0); rand(1);
    ```

save/clear/load

- Use **save** to save variables to a file
  - » `save myFile a b`
    - ➢ saves variables a and b to the file myfile.mat
    - ➢ myfile.mat file is saved in the current directory
    - ➢ Default working directory is
  - » `\MATLAB`
    - ➢ Make sure you're in the desired folder when saving files. Right now, we should be in:
  - » `MATLAB\`

- Use **clear** to remove variables from environment
  - » `clear a b`
    - ➢ look at workspace, the variables a and b are gone

- Use **load** to load variable bindings into the environment
  - » `load myFile`
    - ➢ look at workspace, the variables a and b are back

- Can do the same for entire environment
  - » `save myenv; clear all; load myenv;`

Outline

(1) Getting Started

(2) Scripts

(3) Making Variables

(4) Manipulating Variables

(5) Basic Plotting

Plotting

- Example

```
» x=linspace(0,4*pi,10);
» y=sin(x);
```

- Plot values against their index

```
» plot(y);
```

- Usually we want to plot y versus x

```
» plot(x,y);
```

MATLAB makes visualizing data fun and easy!

What does plot do?

- **plot** generates dots at each (x,y) pair and then connects the dots with a line
- To make plot of a function look smoother, evaluate at more points

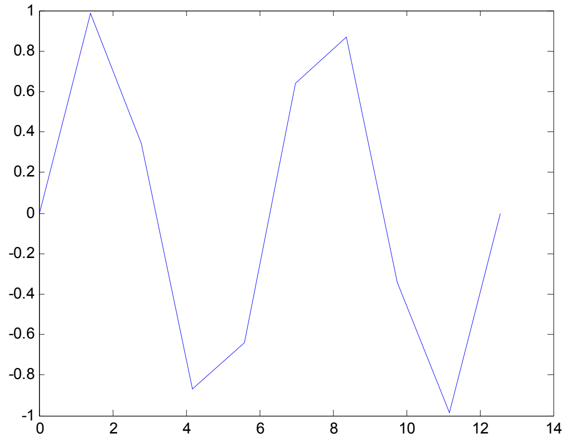
```
» x=linspace(0,4*pi,1000);
» plot(x,sin(x));
```

- x and y vectors must be same size or else you'll get an error

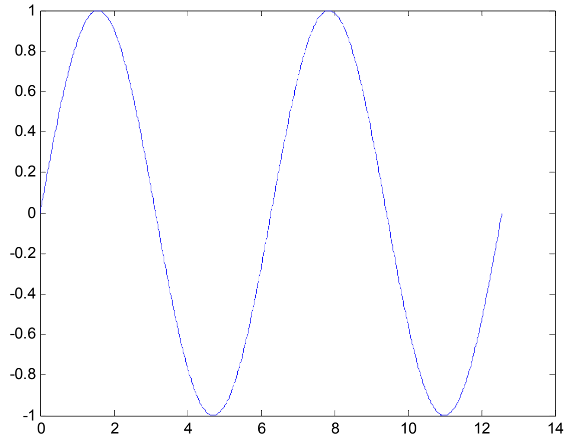
```
» plot([1 2], [1 2 3])
```

  - ➢ error!!

10 x values:



1000 x values:



Let us **plot** the function $y = \sin x$ in the interval 0 to 2π radians. The sequence of commands is

```
≫ x = 0:  pi/60:  2*pi;
≫ y = sin(x);
≫ plot(x, y), grid
≫ title('Sine function')
≫ xlabel('x, radians')
≫ ylabel('sin(x)')
```

The command in the first line generates the sequence

$$0,\ \pi/60,\ 2\pi/60,\ \ldots,\ 2\pi$$

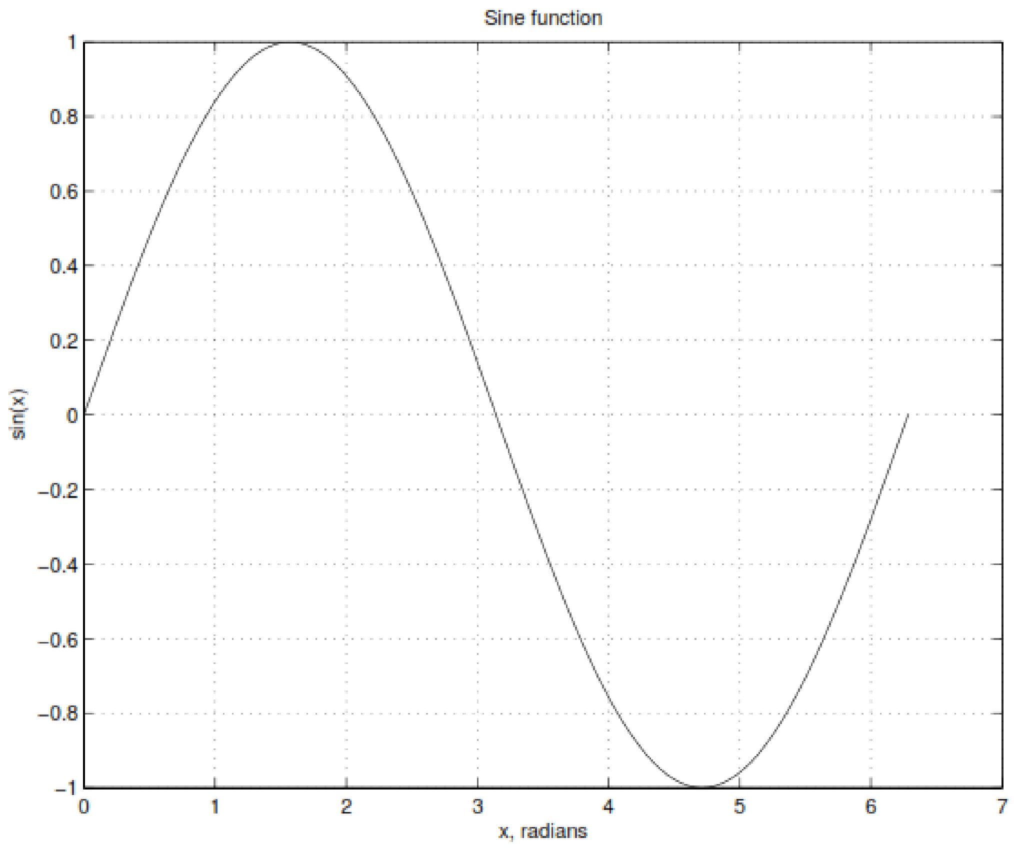


FIGURE : A plot of the sine function

Outline

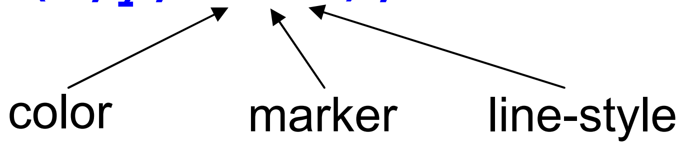
(6) Line Plots

(7) Image/Surface Plots

Plot Options

- Can change the line color, marker style, and line style by adding a string argument

```
» plot(x,y,’k.-’);
```



- Can plot without connecting the dots by omitting line style argument

```
» plot(x,y,’.’)
```

- Look at **help plot** for a full list of colors, markers, and linestyles

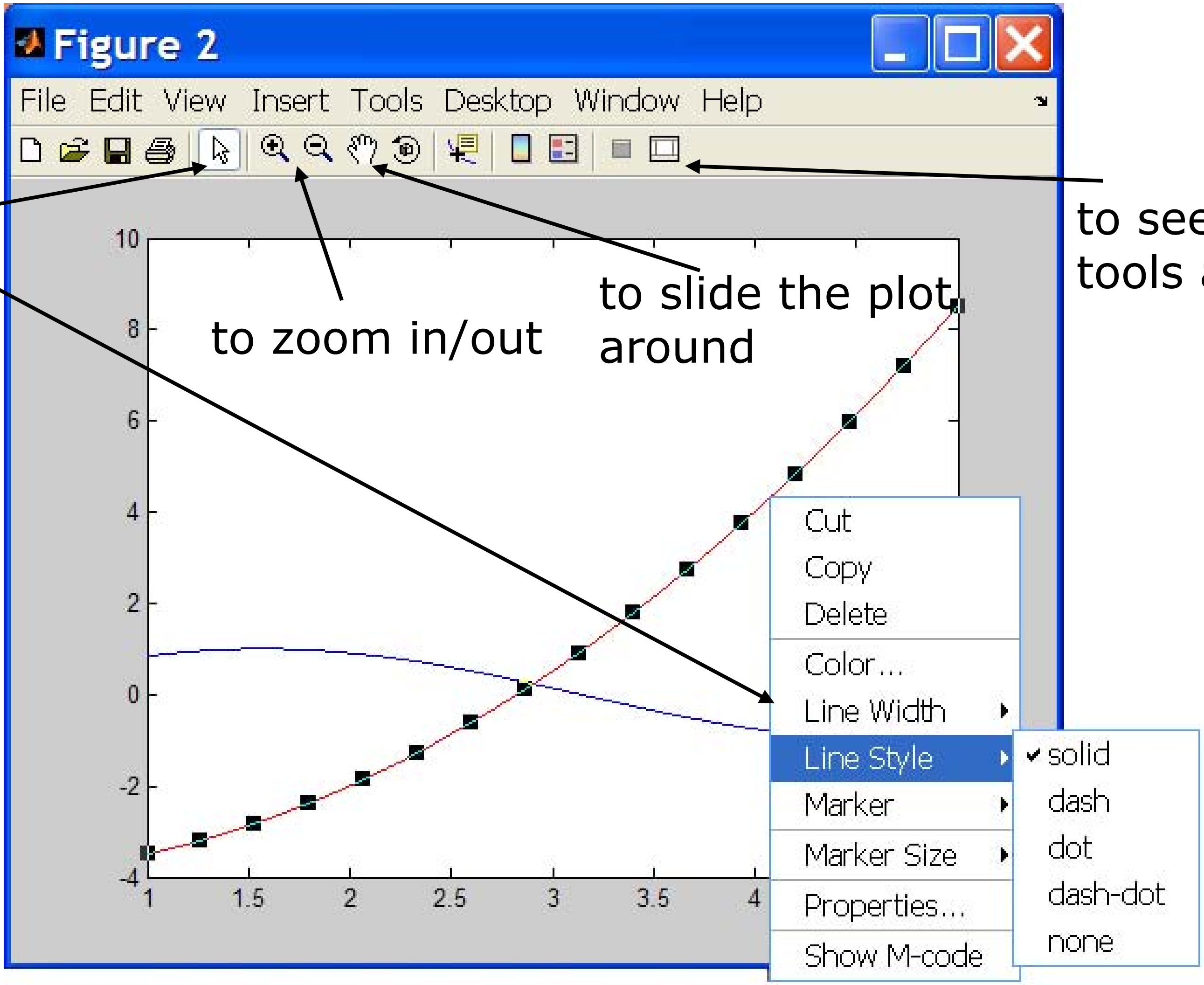
Playing with the Plot

to select lines and delete or change properties

to zoom in/out

to slide the plot around

to see all plot tools at once

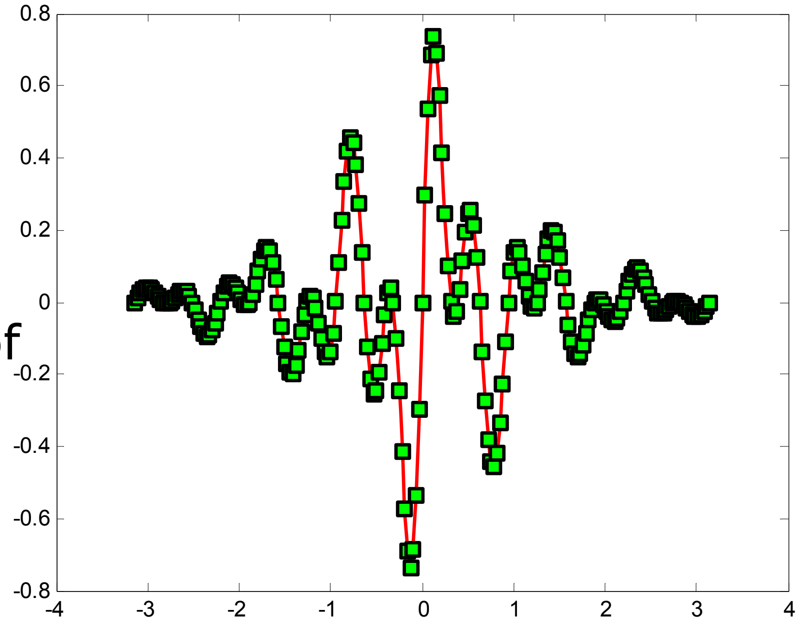


Line and Marker Options

- Everything on a line can be customized

```
» plot(x,y,'--s','LineWidth',2,...
              'Color', [1 0 0], ...
              'MarkerEdgeColor','k',...
              'MarkerFaceColor','g',...
              'MarkerSize',10)
```

You can set colors by using a vector of [R G B] values or a predefined color character like 'g', 'k', etc.



- See **doc line_props** for a full list of properties that can be specified

Cartesian Plots

- We have already seen the plot function

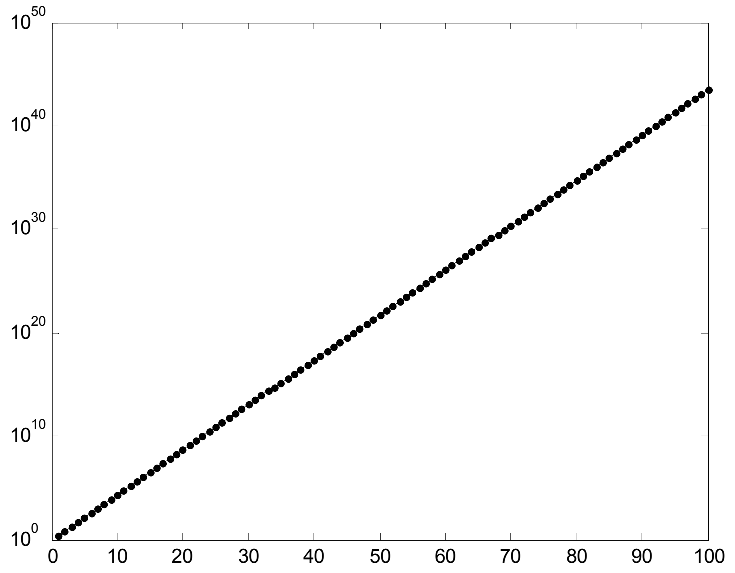
```
» x=-pi:pi/100:pi;
» y=cos(4*x).*sin(10*x).*exp(-abs(x));
» plot(x,y,'k-');
```

- The same syntax applies for semilog and loglog plots

```
» semilogx(x,y,'k');
» semilogy(y,'r.-');
» loglog(x,y);
```

- For example:

```
» x=0:100;
» semilogy(x,exp(x),'k.-');
```



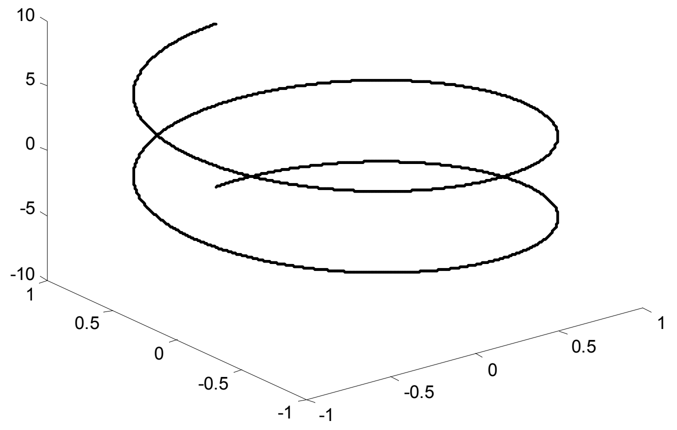
3D Line Plots

- We can plot in 3 dimensions just as easily as in 2

```
» time=0:0.001:4*pi;
» x=sin(time);
» y=cos(time);
» z=time;
» plot3(x,y,z,'k','LineWidth',2);
» zlabel('Time');
```

- Use tools on figure to rotate it
- Can set limits on all 3 axes

```
» xlim, ylim, zlim
```



Axis Modes

- Built-in axis modes
  - » `axis square`
    - ➢ makes the current axis look like a box
  - » `axis tight`
    - ➢ fits axes to data
  - » `axis equal`
    - ➢ makes x and y scales the same
  - » `axis xy`
    - ➢ puts the origin in the bottom left corner (default for plots)
  - » `axis ij`
    - ➢ puts the origin in the top left corner (default for matrices/images)

Multiple Plots in one Figure

- To have multiple axes in one figure
  - » `subplot(2,3,1)`
    - ➢ makes a figure with 2 rows and three columns of axes, and activates the first axis for plotting
    - ➢ each axis can have labels, a legend, and a title
  - » `subplot(2,3,4:6)`
    - ➢ activating a range of axes fuses them into one
- To close existing figures
  - » `close([1 3])`
    - ➢ closes figures 1 and 3
  - » `close all`
    - ➢ closes all figures (useful in scripts/functions)

Copy/Paste Figures

- Figures can be pasted into other apps (word, ppt, etc)
- *Edit→ copy options→ figure copy template*
    - ➢ Change font sizes, line properties; presets for word and ppt
- *Edit→ copy figure* to copy figure
- Paste into document of interest

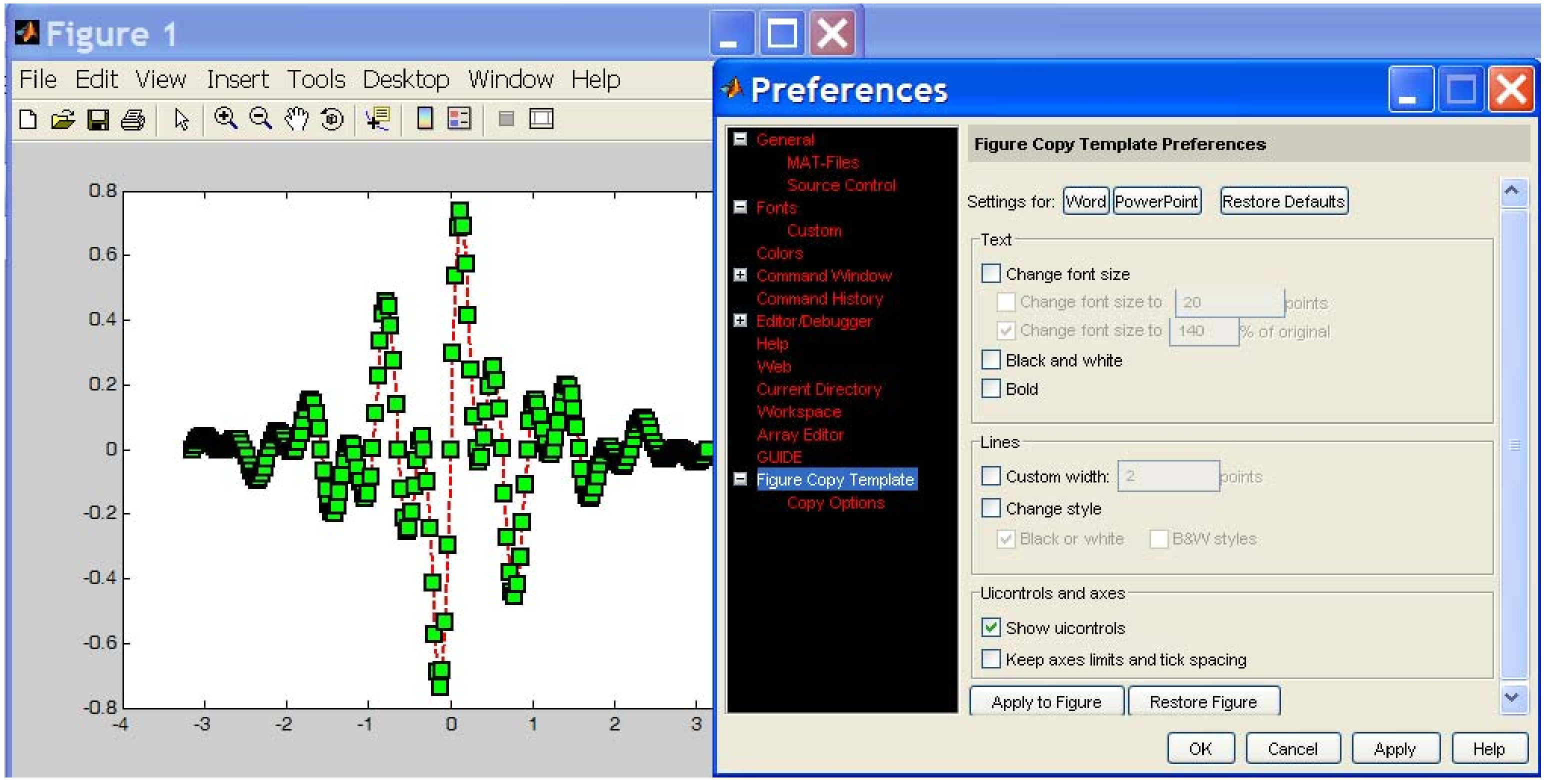


Figure Copy Template Preferences

Settings for: Word PowerPoint Restore Defaults

Text
- [ ] Change font size
  - [ ] Change font size to 20 points
  - [✓] Change font size to 140 % of original
- [ ] Black and white
- [ ] Bold

Lines
- [ ] Custom width: 2 points
- [ ] Change style
  - [✓] Black or white
  - [ ] B&W styles

Uicontrols and axes
- [✓] Show uicontrols
- [ ] Keep axes limits and tick spacing

Courtesy of The MathWorks, Inc. Used with permission.

Saving Figures

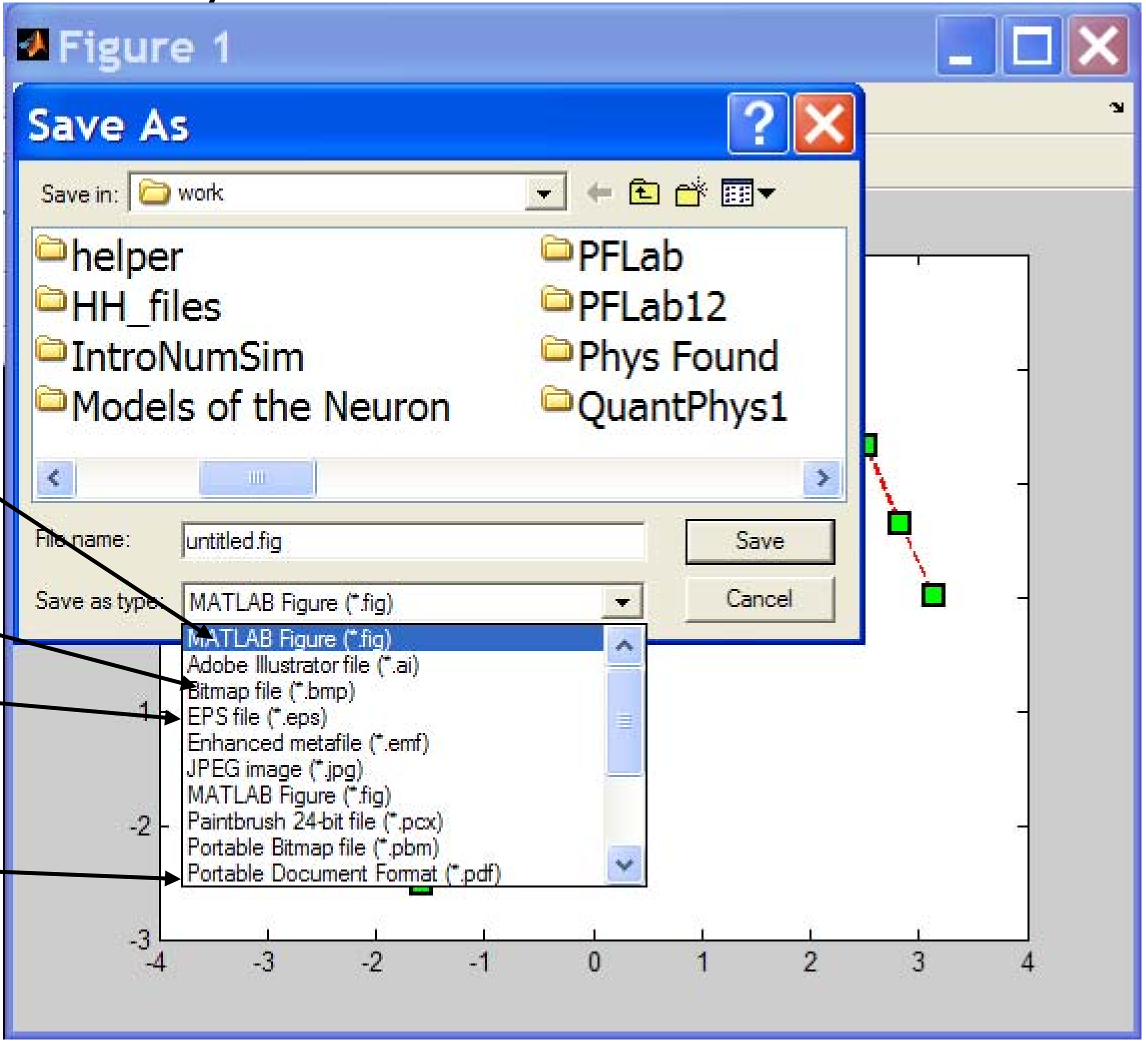
- Figures can be saved in many formats. The common ones are:

.fig preserves all information

.bmp uncompressed image

.eps high-quality scaleable format

.pdf compressed image

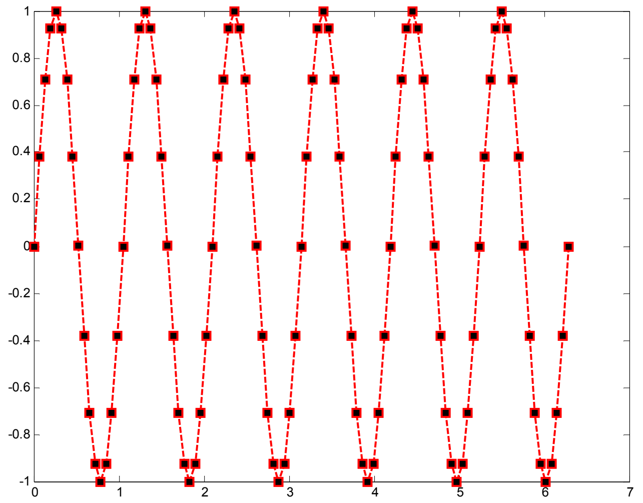


Courtesy of The MathWorks, Inc. Used with permission.

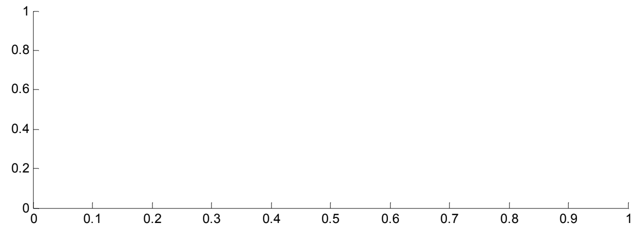
Advanced Plotting: Exercise

- Modify the plot command in your plotSin function to use **squares** as markers and a **dashed red** line of **thickness** 2 as the line. Set the marker face color to be **black** (properties are `LineWidth`, `MarkerFaceColor`)
- If there are 2 inputs, open a new figure with 2 axes, one on top of the other (not side by side), and activate the top one (`subplot`)

`plotSin(6)`



`plotSin(1,2)`



Advanced Plotting: Exercise

- Modify the plot command in your plotSin function to use **squares** as markers and a **dashed red** line of **thickness** 2 as the line. Set the marker face color to be **black** (properties are `LineWidth`, `MarkerFaceColor`)
- If there are 2 inputs, open a new figure with 2 axes, one on top of the other (not side by side), and activate the top one (`subplot`)

```
» if nargin == 1
      plot(x,sin(f1*x),'rs--',...
       'LineWidth',2,'MarkerFaceColor','k');
  elseif nargin == 2
      subplot(2,1,1);
  end
```

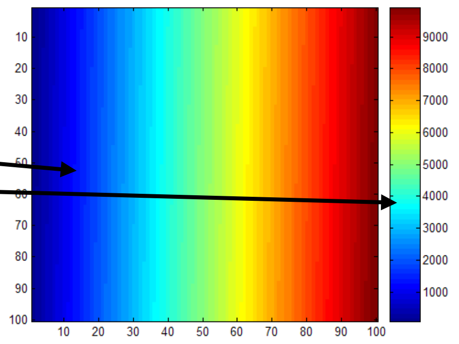
Outline

(6) Line Plots

(7) Image/Surface Plots

Visualizing matrices

- Any matrix can be visualized as an image
  - » `mat=reshape(1:10000,100,100);`
  - » `imagesc(mat);`
  - » `colorbar`

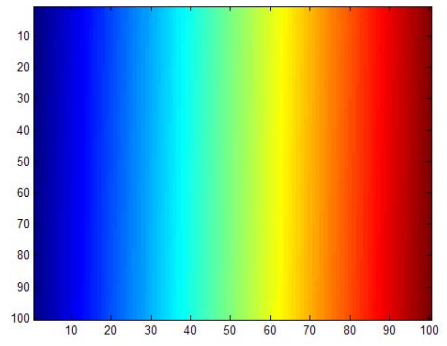


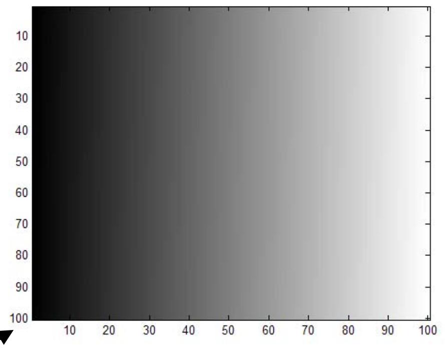
- **imagesc** automatically scales the values to span the entire colormap

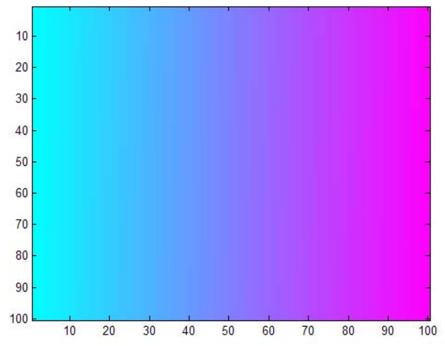
- Can set limits for the color axis (analogous to `xlim`, `ylim`)
  - » `caxis([3000 7000])`

Colormaps

- You can change the colormap:
  - » `imagesc(mat)`
    - ➢ default map is jet
  - » `colormap(gray)`
  - » `colormap(cool)`
  - » `colormap(hot(256))`

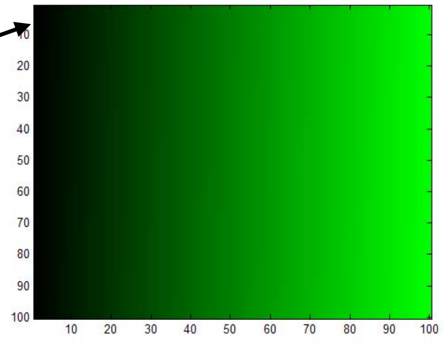


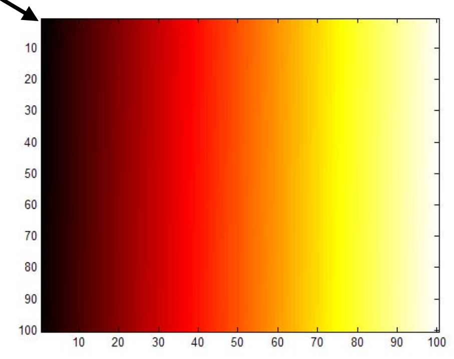




- See `help hot` for a list

- Can define custom colormap
  - » `map=zeros(256,3);`
  - » `map(:,2)=(0:255)/255;`
  - » `colormap(map);`





Surface Plots

- It is more common to visualize *surfaces* in 3D

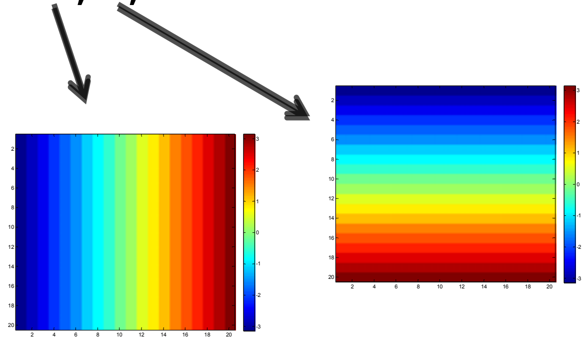
- Example:

$$\boxed{\begin{array}{l} f(x,y) = sin(x)cos(y) \\ x \in [-\pi,\pi]; y \in [-\pi,\pi] \end{array}}$$

- **surf** puts vertices at specified points in space x,y,z, and connects all the vertices to make a surface

- The vertices can be denoted by matrices X,Y,Z

- How can we make these matrices
  - ➢ loop (DUMB)
  - ➢ built-in function: **meshgrid**



surf

- Make the x and y vectors

```
» x=-pi:0.1:pi;
» y=-pi:0.1:pi;
```

- Use meshgrid to make matrices (this is the same as loop)

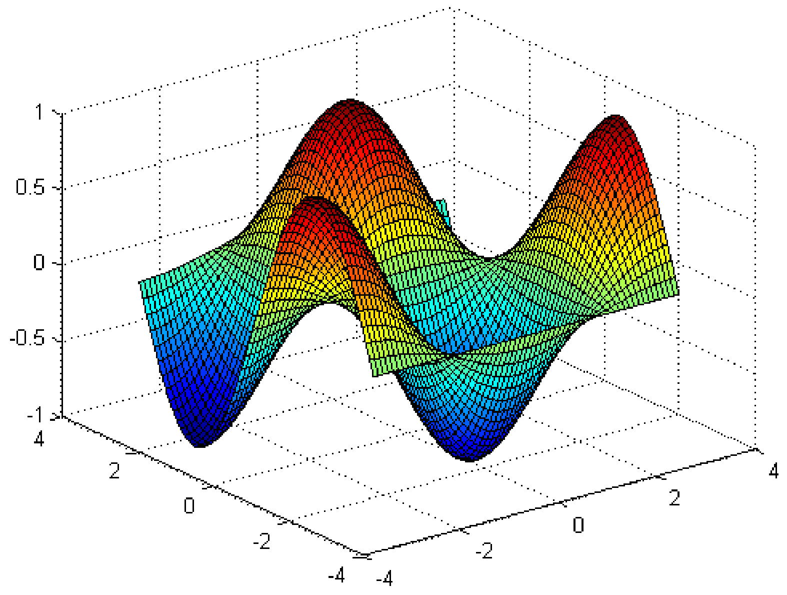
```
» [X,Y]=meshgrid(x,y);
```

- To get function values, evaluate the matrices

```
» Z =sin(X).*cos(Y);
```

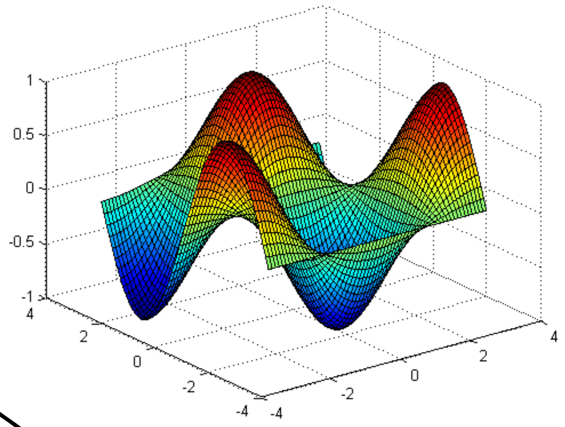
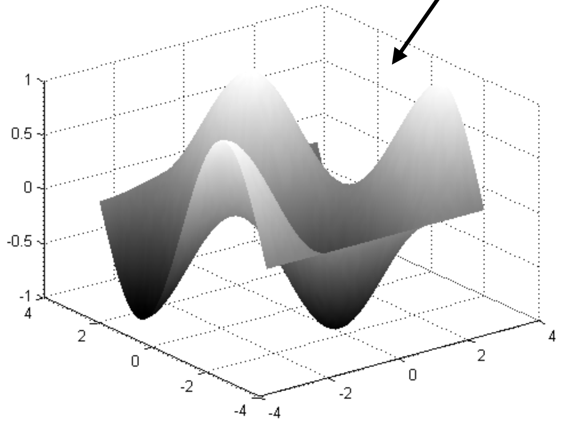
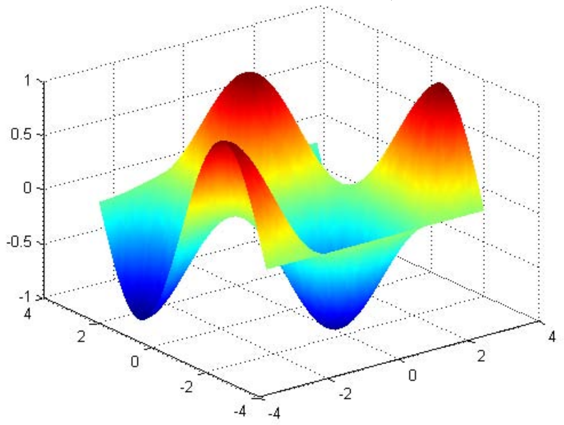
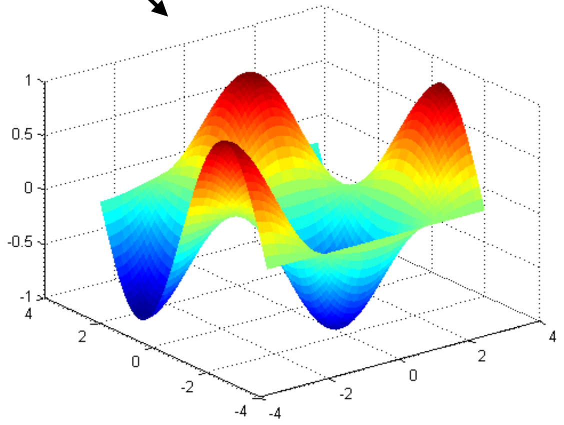
- Plot the surface

```
» surf(X,Y,Z)
» surf(x,y,Z);
```



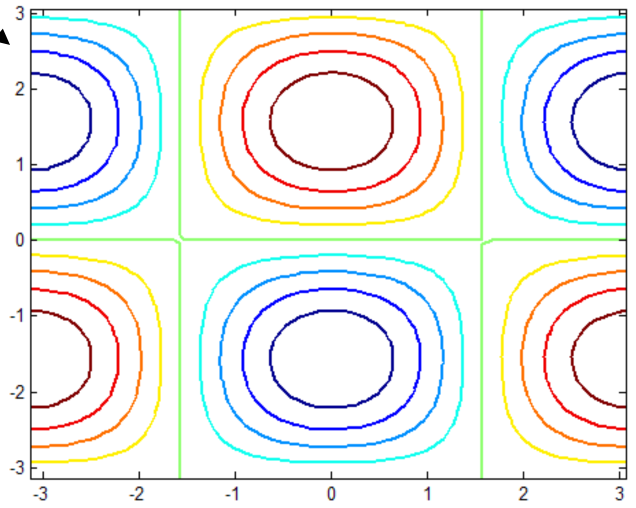
surf Options

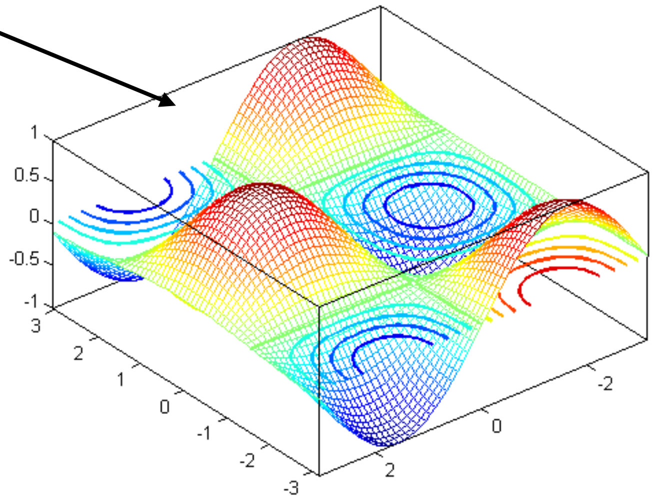
- See **help surf** for more options
- There are three types of surface shading
  - » `shading faceted`
  - » `shading flat`
  - » `shading interp`
- You can change colormaps
  - » `colormap(gray)`

contour

- You can make surfaces two-dimensional by using contour
  - » `contour(X,Y,Z,'LineWidth',2)`
    - ➢ takes same arguments as surf
    - ➢ color indicates height
    - ➢ can modify linestyle properties
    - ➢ can set colormap
  - » `hold on`
  - » `mesh(X,Y,Z)`





Exercise: 3-D Plots

- Modify `plotSin` to do the following:
- If two inputs are given, evaluate the following function:

$$Z = \sin(f_1 x) + \sin(f_2 y)$$

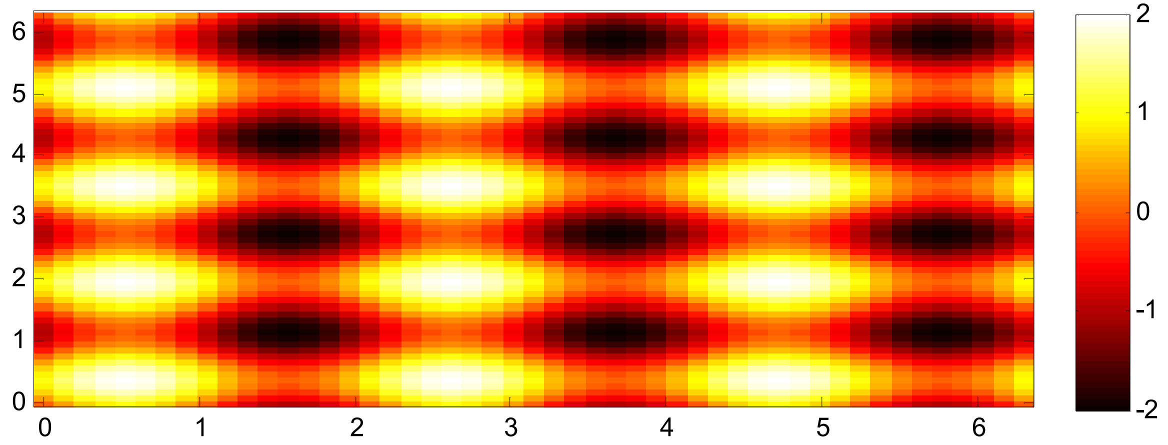
- y should be just like x, but using f2. (use `meshgrid` to get the X and Y matrices)
- In the top axis of your subplot, display an image of the Z matrix. Display the colorbar and use a `hot` colormap. Set the axis to xy (`imagesc`, `colormap`, `colorbar`, `axis`)
- In the bottom axis of the subplot, plot the 3-D surface of Z (`surf`)

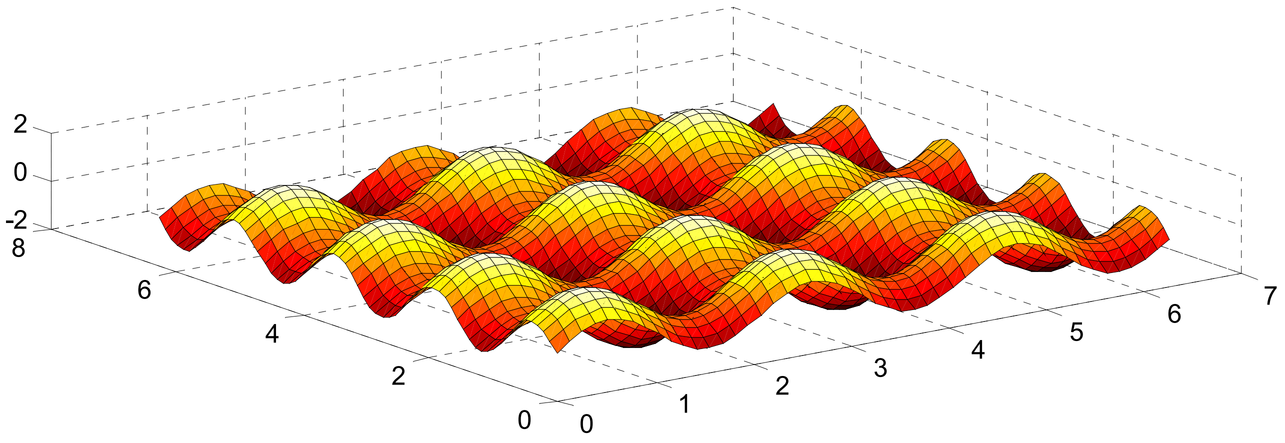
Exercise: 3-D Plots

```
» function plotSin(f1,f2)

  x=linspace(0,2*pi,round(16*f1)+1);
  figure

  if nargin == 1
      plot(x,sin(f1*x),'rs--',...
       'LineWidth',2,'MarkerFaceColor','k');
  elseif nargin == 2
      y=linspace(0,2*pi,round(16*f2)+1);
      [X,Y]=meshgrid(x,y);
      Z=sin(f1*X)+sin(f2*Y);
      subplot(2,1,1); imagesc(x,y,Z); colorbar;
      axis xy; colormap hot
      subplot(2,1,2); surf(X,Y,Z);
  end
```

Exercise: 3-D Plots

`plotSin(3,4)` generates this figure





Specialized Plotting Functions

- MATLAB has a lot of specialized plotting functions
- **polar**-to make polar plots
  - » `polar(0:0.01:2*pi,cos((0:0.01:2*pi)*2))`
- **bar**-to make bar graphs
  - » `bar(1:10,rand(1,10));`
- **quiver**-to add velocity vectors to a plot
  - » `[X,Y]=meshgrid(1:10,1:10);`
  - » `quiver(X,Y,rand(10),rand(10));`
- **stairs**-plot piecewise constant functions
  - » `stairs(1:10,rand(1,10));`
- **fill**-draws and fills a polygon with specified vertices
  - » `fill([0 1 0.5],[0 0 1],'r');`
- see help on these functions for syntax
- **doc specgraph** – for a complete list

End of Section 1

(1) Getting Started

(2) Scripts

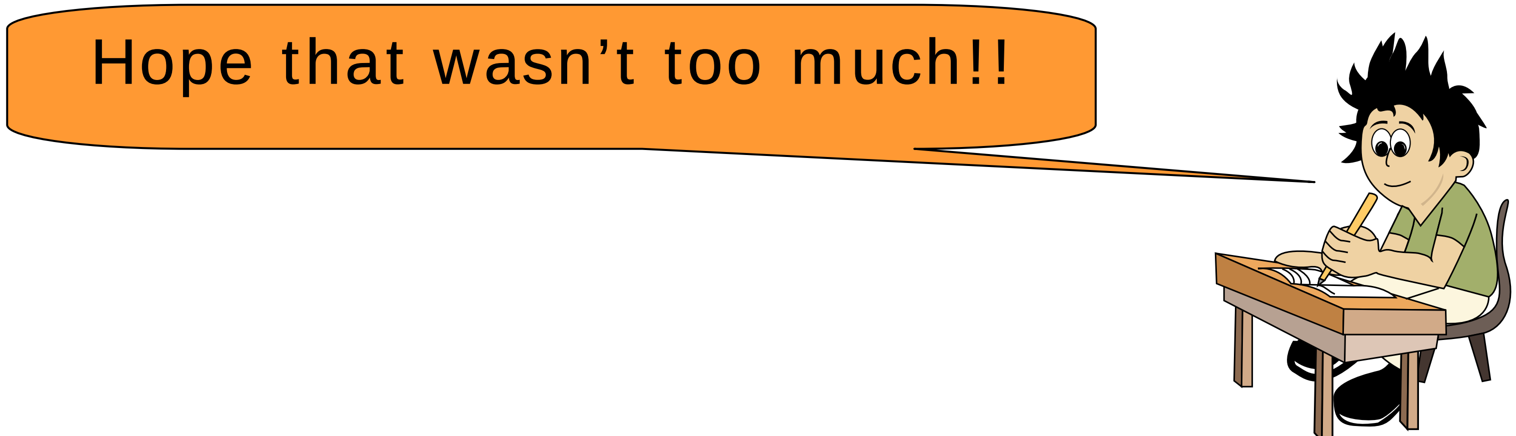
(3) Making Variables

(4) Manipulating Variables

(5) Basic Plotting

(6) Line Plots

(7) Image/Surface Plots



MATLAB Programming

Section 2: Programming

Outline

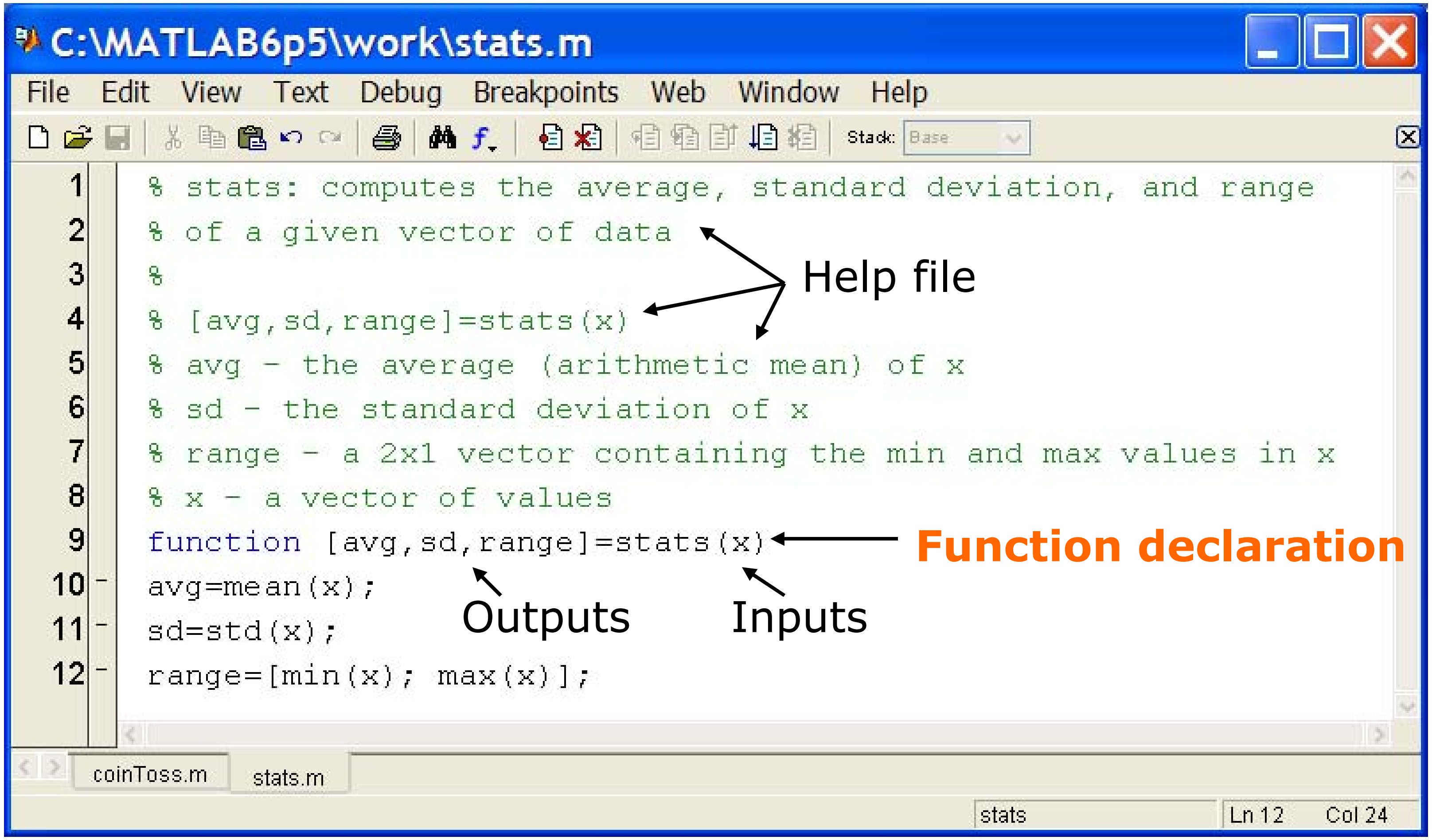
(1) Functions

(2) Flow Control

(3) Vectorization

User-defined Functions

- Functions look exactly like scripts, but for **ONE** difference
  - Functions must have a function declaration



```
% stats: computes the average, standard deviation, and range
% of a given vector of data
%
% [avg,sd,range]=stats(x)
% avg - the average (arithmetic mean) of x
% sd - the standard deviation of x
% range - a 2x1 vector containing the min and max values in x
% x - a vector of values
function [avg,sd,range]=stats(x)
avg=mean(x);
sd=std(x);
range=[min(x); max(x)];
```

Courtesy of The MathWorks, Inc. Used with permission.

User-defined Functions

- Some comments about the function declaration

Inputs must be specified

```
function [x, y, z] = funName(in1, in2)
```

Must have the reserved word: function

Function name should match MATLAB file name

If more than one output, must be in brackets

- No need for return: MATLAB 'returns' the variables whose names match those in the function declaration
- Variable scope: Any variables created within the function but not returned disappear after the function stops running

Functions: overloading

- MATLAB functions are generally overloaded
  - ➢ Can take a variable number of inputs
  - ➢ Can return a variable number of outputs

- What would the following commands return:

```
» a=zeros(2,4,8); %n-dimensional matrices are OK
» D=size(a)
» [m,n]=size(a)
» [x,y,z]=size(a)
» m2=size(a,2)
```

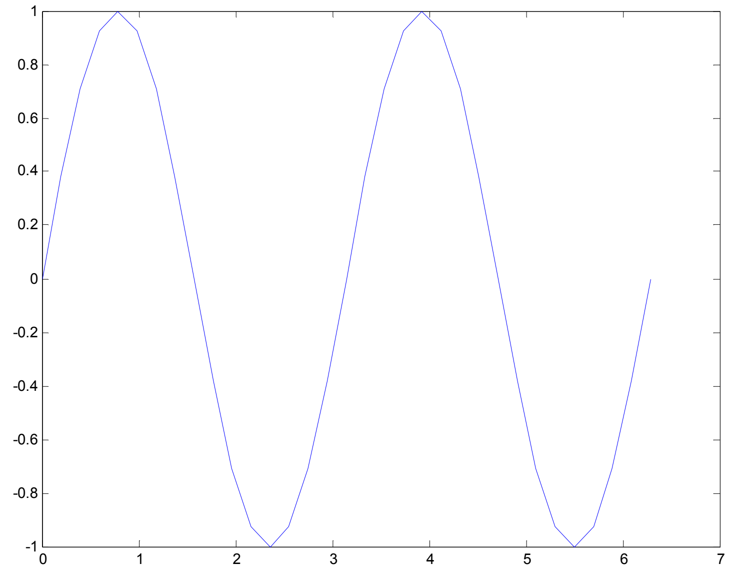
- You can overload your own functions by having variable input and output arguments (see `varargin`, `nargin`, `varargout`, `nargout`)

Functions: Excercise

- Write a function with the following declaration:
  `function plotSin(f1)`

- In the function, plot a sin wave with frequency f1, on the range [0,2π]: $\sin\left(f_1 x\right)$

- To get good sampling, use 16 points per period.



Functions: Excercise

- Write a function with the following declaration:
  `function plotSin(f1)`

- In the function, plot a sin wave with frequency f1, on the range [0,2π]: $\sin(f_1 x)$

- To get good sampling, use 16 points per period.

- In an MATLAB file saved as plotSin.m, write the following:
  - » `function plotSin(f1)`

```
x=linspace(0,2*pi,f1*16+1);
figure
plot(x,sin(f1*x))
```

Example:

Try the following more general example, which returns the mean (avg) and standard deviation (stdev) of the values in the vector x. Although there are two MATLAB functions to do this (mean and std), it is useful to have them combined into one. Write a function file stats.m:

```
function [avg, stdev] = stats( x )    % function definition line
% STATS         Mean and standard deviation                % H1 line
%               Returns mean (avg) and standard          % Help text
%               deviation (stdev) of the data in the
%               vector x, using Matlab functions

avg = mean(x);                                       % function body
stdev = std(x);
```

Now test it in the Command Window with some random numbers, e.g.,

```
r = rand(100,1);
[a, s] = stats(r);
```

EXAMPLE: Weight hanging from two bars

1- Find the tension forces, R1 and R2, in the two bars

$$a = c\tan\alpha,\ b = c\tan\beta,\ \gamma = 180 - (\alpha + \beta)$$

Moreover, the law of sines yields

$$\frac{\sin\alpha}{R_2} = \frac{\sin\beta}{R_1} = \frac{\sin\gamma}{w}$$

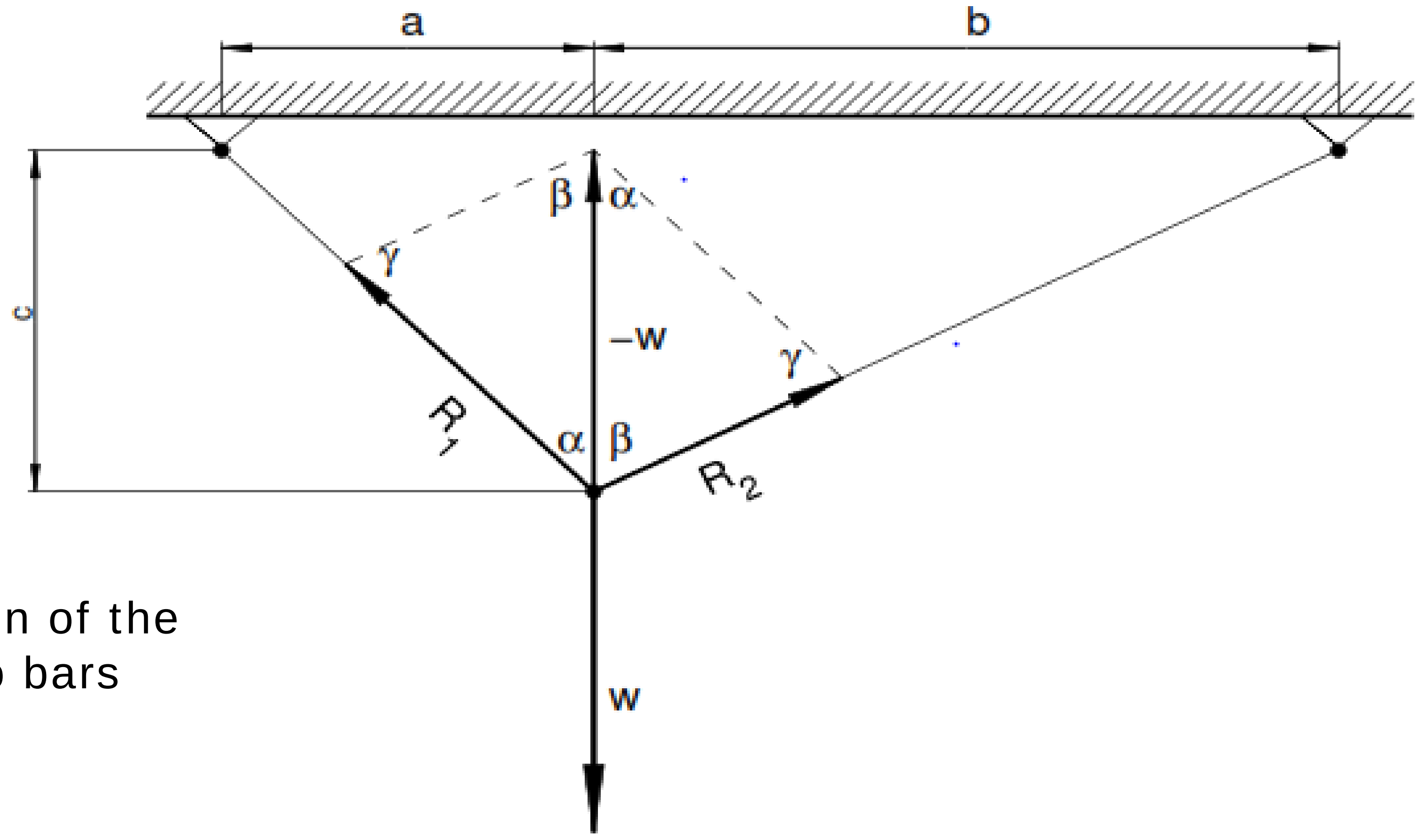


FIGURE: Calculation of the tensions in the two bars

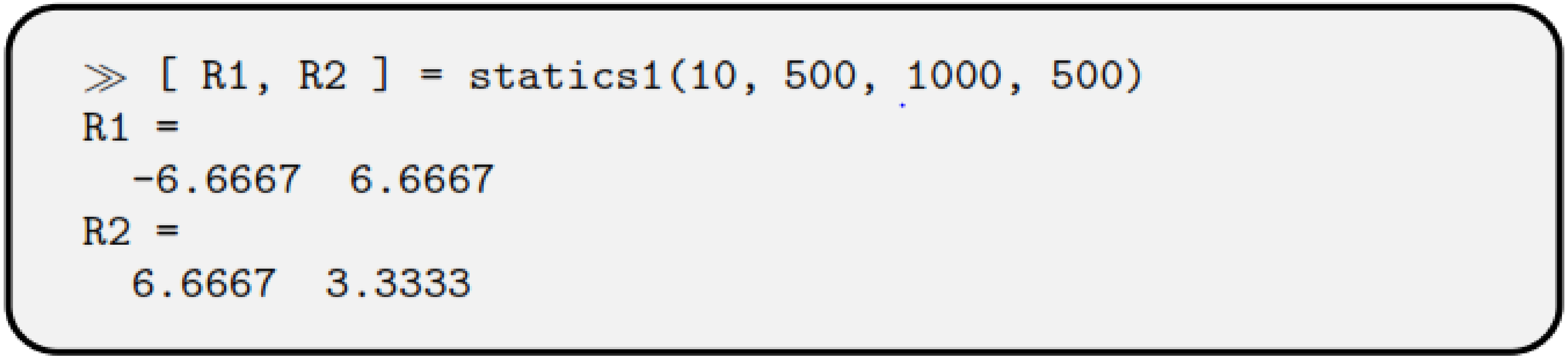
Solution:

```
function    [ R1 R2 ] = statics1(w, a, b, c)

STATICS1   Finds tensions in two bars connected to a point
           where a weight w hangs.
           a -  horizontal distance between first
           articulation and vertical of w
           b - horizontal distance between third
           articulation and vertical of w
           c - vertical distance between the plane of
           upper articulations and the point where w hangs.
           Written by Adrian Biran, May 2009
% calculate angles
alpha = atand(a/c);
beta    = atand(b/c);
gamma = 180 - alpha - beta;
% calculate magnitudes of tension vectors
r1          = sind(beta)*w/sind(gamma);
r2          = sind(alpha)*w/sind(gamma);
% define tension vectors
R1          = r1*[ -sind(alpha), cosd(alpha) ];
R2          = r2*[  sind(beta), cosd(beta) ];
```

2- Test the function with:

w =10 kN,a= 500 mm, b = 1000 mm, c = 500 mm.

Solution:



```
≫ [ R1, R2 ] = statics1(10, 500, 1000, 500)
R1 =
  -6.6667  6.6667
R2 =
  6.6667  3.3333
```

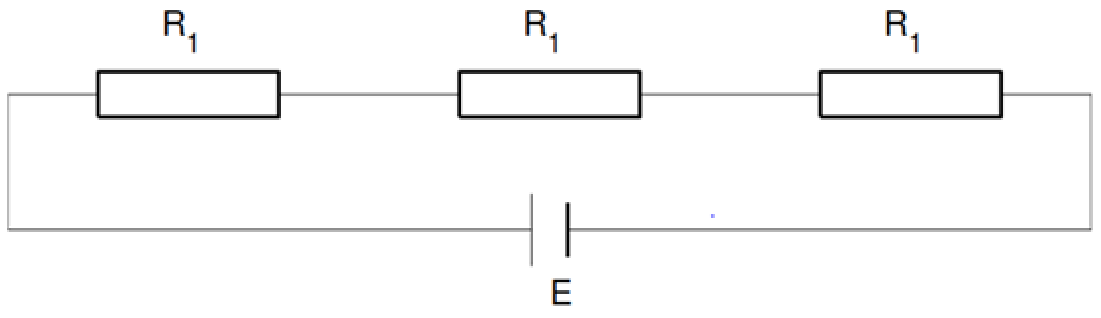
Example: Figure (A) shows three resistors (R_1, R_2 and R_3), connected in series with a dc source (E). The same current (i) passes through all three resistors. write a function in MATLAB called 'series' to calculate the equivalent resistance (Req), the current intensity (i), and the individual voltage drops (V_1, V_2 and V_3). Use the following equations:

$Req = R_1 + R_2 + R_3$

$i = E / Req$

$E = V_1 + V_2 + V_3 = R_1 X i + R_2 X i + R_3 X i$

The 'main function' should call the 'series' function and enter the values E =24 v, R_1 = 100 ohms, R_2 = 200 ohms and R_3 = 300 in the 'main function'.



Solution:

```
function main
clear all
close all
clc
E=24;
R=[100,200,300];
[Req,i,V]=series(R,E)
end
function [Req_f,i_f,V_f]=series(R_f,E_f)
Req_f=sum(R_f);
i_f=E_f/Req_f;
V_f=i_f*R_f;
end
```

Outline

(1) Functions

(2) Flow Control

(3) Vectorization

Relational Operators

- MATLAB uses *mostly* standard relational operators

➢ equal	==
➢ **not** equal	~=
➢ greater than	>
➢ less than	<
➢ greater or equal	>=
➢ less or equal	<=

• Logical operators	elementwise	short-circuit (scalars)
➢ And	&	&&
➢ Or	|	||
➢ **Not**	~	
➢ Xor	xor	
➢ All true	all	
➢ Any true	any	

- Boolean values: zero is false, nonzero is true
- See **help .** for a detailed list of operators

Relational and Logical Operations

These operations and functions provide answers to **True-False** questions. One important use of this capability is to control the flow or order of execution of a series of MATLAB commands (usually in an M-file) based on the results of true/false questions.

As inputs to all relational and logical expressions, MATLAB considers any nonzero number to be true, and zero to be False. The output of all relational and logical expressions produces ***one for True*** and ***zero for False***, and the array is flagged as *logical*. That is, the result contains numerical values 1 and 0 that can be used in mathematical statement, but also allow logical array addressing.

4-1 Relational Operations

Operation	Description
= =	Equal
~ =	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

Example:

```
>> a=1:9; b=9-a;
>> t=a>4       %finds elements of (a) that are greater than 4.

t =

     0     0     0     0     1     1     1     1     1
```

Zeros appear where a ≤ 4, and ones where a > 4.

```
>> t= (a==b)     %finds elements of (a) that are equal to those in (b).

t =
     0     0     0     0     0     0     0     0     0
```

4-2 Logical Operation

Operation	Description
&	Logical AND **and(a,b)**
|	Logical OR **or(a,b)**
~	Logical NOT
xor (a,b)	Logical EXCLUSIVE OR

Example:

```
>> a = [0  4  0  -3  -5  2];
>> b = ~a
b =
     1     0     1     0     0     0

>> c=a&b
c =
     0     0     0     0     0     0
```

Example: let `x=[ 2 -3 5 ;0 11 0],` then

a) find elements in x that are greater than 2
b) find the number of nonzero elements in x

solution

```
a)
  >> x>2
  ans =

           0     0     1
           0     1     0
```

```
b)
   >> t=~(~x);
   >> sum(sum(t))

   ans =

        4
```

4-3 <u>Bitwise Operation</u>

MATLAB also has a number of functions that perform bitwise logical operations.

If A, B unsigned integers then:

Operation	Description
bitand (A, B)	Bitwise AND
bitor (A, B)	Bitwise OR
bitset (A, BIT)	sets bit position **BIT** in **A** to 1
bitget (A, BIT)	returns the value of the bit at position **BIT** in A
xor (A, B)	Bitwise EXCLUSIVE OR

<u>Example:</u> **if A=5, B=6 then:**

```
>> bitget(A,3)
ans =  1
```

where $A=\overbrace{1\ 0\ \boxed{1}\ 0\ 0\ 0\ 0\ 0}^{5}$

```
>> bitget(A,(1:8))
ans =
          1    0    1    0    0    0    0    0
>> bitand(A,B)
ans =
          4
>> and(A,B)
ans =
          1
```

4-4 Logical Functions

MATLAB has a number of useful logical functions that operate on scalars, vectors, and matrices. Examples are given in the following list:-

Function	Description
`any(x)`	True if any element of a vector is a nonzero number or is logical 1 (TRUE)
`all(x)`	True if all elements of a vector are nonzero.
`find(x)`	Find indices of nonzero elements
`isnan(x)`	True for Not-a-Number
`isinf(x)`	True for infinite elements.
`isempty(x)`	True for empty array.

Example: Let A=[4 9 7 0 5],

```
>> any(A)
ans =     1

>> all(A)
ans =     0

>> find(A)
ans =     1     2     3     5
```

To remove zero elements from matrix

```
>> B=A(find(A));
>> B
B =      4     9     7     5
```

To find the **location** of maximum number of **B**

```
>> find(B==max(B))
    ans =     2
```

Exercises

1- write a program to read three bits x, y, z, then compute:

```
a)   v = (x and y) or z
b)   w = not (x or y) and z
c)   u = (x and not (y)) or (not (x) and y)
```

2- Write a program for three bits parity generator using even-parity bit.

3- Write a program to convert a three bits binary number into its equivalent gray code.

4- if `q=[1 5 6 8 3 2 4 5 9 10 1],x=[ 3 5 7 8 3 1 2 4 11 5 9]`, then:

a) find elements of (q) that are greater than 4.
b) find elements of (q) that are equal to those in (x).
c) find elements of (x) that are less than or equal to 7.

5- If `x=[10 3 ; 9 15],  y=[10 0; 9 3], z=[-1 0; -3 2],` what is the output of the following statements:

```
a) v = x > y
b) w = z >= y
c) u = ~z & y
d) t = x & y < z
```

Note the following points:

- *condition* is usually a *logical expression* (i.e., it contains a *relational operator*), which is either *true* or *false*. The relational operators are shown in Table . MATLAB allows you to use an arithmetic expression for *condition*. If the expression evaluates to 0, it is regarded as false; any other value is true. This is not generally recommended; the `if` statement is easier to understand (for you or a reader of your code) if *condition* is a logical expression.
- If *condition* is true, *statement* is executed, but if *condition* is false, nothing happens.
- *condition* may be a vector or a matrix, in which case it is true only if *all* of its elements are nonzero. A single zero element in a vector or matrix renders it false.

Exercises

The following statements all assign logical expressions to the variable **`x`**. See if you can correctly determine the value of **`x`** in each case before checking your answer with MATLAB.

```
(a) x = 3 > 2
(b) x = 2 > 3
(c) x = -4 <= -3
(d) x = 1 < 1
(e) x = 2 ~= 2
(f) x = 3 == 3
(g) x = 0 < 0.5 < 1
```

Did you get item (f)? **`3 == 3`** is a logical expression that is true since 3 is undoubtedly equal to 3. The value 1 (for true) is therefore assigned to **`x`**. After executing these commands type the command **`whos`** to find that the variable **`x`** is in the class of *logical* variables.

What about (g)? As a *mathematical* inequality,

$$0 < 0.5 < 1$$

is undoubtedly true from a nonoperational point of view. However, as a MATLAB operational expression, the left-hand < is evaluated first, **`0 < 0.5`**, giving 1 (true). Then the right-hand operation is performed, **`1 < 1`**, giving 0 (false). Makes you think, doesn't it?

if/else/elseif

- Basic flow-control, common to all languages
- MATLAB syntax is somewhat unique

IF

```
if cond
    commands
end
```

Conditional statement: evaluates to true or false

ELSE

```
if cond
    commands1
else
    commands2
end
```

ELSEIF

```
if cond1
    commands1
elseif cond2
    commands2
else
    commands3
end
```

- No need for parentheses: command blocks are between reserved words

```
if condition
   statementsA
else
   statementsB
end
```

Note that

- *statementsA* and *statementsB* represent one or more statements.
- If *condition* is true, *statementsA* are executed, but if *condition* is false, *statementsB* are executed. This is essentially how you force MATLAB to choose between two alternatives.
- `else` is optional.

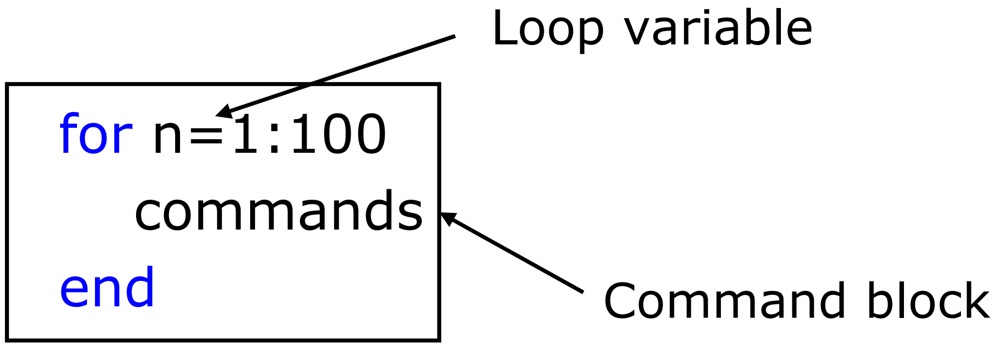
elseif

This is sometimes called an `elseif` *ladder*. It works as follows:

1. *condition1* is tested. If it is true, *statementsA* are executed; MATLAB then moves to the next statement after `end`.
2. If *condition1* is false, MATLAB checks *condition2*. If it is true, *statementsB* are executed, followed by the statement after `end`.
3. In this way, all conditions are tested until a true one is found. As soon as a true condition is found, no further `elseif`s are examined and MATLAB jumps off the ladder.
4. If none of the conditions is true, *statements* after `else` are executed.
5. Arrange the logic so that not more than one of the conditions is true.
6. There can be any number of `elseif`s, but at most one `else`.
7. `elseif` *must* be written as one word.
8. It is good programming style to indent each group of statements as shown.

for

- **for** loops: use for a known number of iterations
- MATLAB syntax:



```
for n=1:100
    commands
end
```

- The loop variable
  - Is defined as a vector
  - Is a scalar within the command block
  - Does not have to have consecutive values (but it's usually cleaner if they're consecutive)
- The command block
  - Anything between the **for** line and the **end**

Note the following:

- Don't forget the commas (semicolons will also do if appropriate). If you leave them out you will get an error message.
- Again, *statements* can be one or more statements separated by commas or semicolons.
- If you leave out `end`, MATLAB will wait for you to enter it. Nothing will happen until you do so.

while

- The while is like a more general for loop:
    - Don't need to know number of iterations

WHILE

```
while cond
    commands
end
```

- The command block will execute while the conditional expression is true
- Beware of infinite loops!

Exercise: Conditionals

- Modify your plotSin(f1) function to take two inputs: plotSin(f1,f2)
- If the number of input arguments is 1, execute the plot command you wrote before. Otherwise, display the line `'Two inputs were given'`
- Hint: the number of input arguments are in the built-in variable `nargin`

Exercise: Conditionals

- Modify your `plotSin(f1)` function to take two inputs: `plotSin(f1,f2)`
- If the number of input arguments is 1, execute the plot command you wrote before. Otherwise, display the line `'Two inputs were given'`
- Hint: the number of input arguments are in the built-in variable `nargin`

»
```
function plotSin(f1,f2)

x=linspace(0,2*pi,f1*16+1);
figure

if nargin == 1
    plot(x,sin(f1*x));
elseif nargin == 2
    disp('Two inputs were given');
end
```

MATLAB - The switch Statement

A switch block conditionally executes one set of statements from several choices. Each choice is covered by a case statement.

An evaluated switch_expression is a scalar or string.

An evaluated case_expression is a scalar, a string or a cell array of scalars or strings.

The switch block tests each case until one of the cases is true. A case is true when −

For numbers, **eq(case_expression,switch_expression)**.

For strings, **strcmp(case_expression,switch_expression)**.

For objects that support the **eq(case_expression,switch_expression)**.

For a cell array case_expression, at least one of the elements of the cell array matches switch_expression, as defined above for numbers, strings and objects.

When a case is true, MATLAB executes the corresponding statements and then exits the switch block.

The **otherwise** block is optional and executes only when no case is true.

Syntax

The syntax of switch statement in MATLAB is −

```
switch <switch_expression>
   case <case_expression>
      <statements>
   case <case_expression>
      <statements>
      ...
      ...
   otherwise
      <statements>
end
```

Example

Create a script file and type the following code in it −

```
grade = 'B';
   switch(grade)
   case 'A'
      fprintf('Excellent!\n' );
   case 'B'
      fprintf('Well done\n' );
   case 'C'
      fprintf('Well done\n' );
   case 'D'
      fprintf('You passed\n' );
   case 'F'
      fprintf('Better try again\n' );
   otherwise
      fprintf('Invalid grade\n' );
   end
```

When you run the file, it displays −

```
Well done
```

Outline

(1) Functions

(2) Flow Control

(3) Vectorization

Performance Measures

- It can be useful to know how long your code takes to run
  - ➢ To predict how long a loop will take
  - ➢ To pinpoint inefficient code

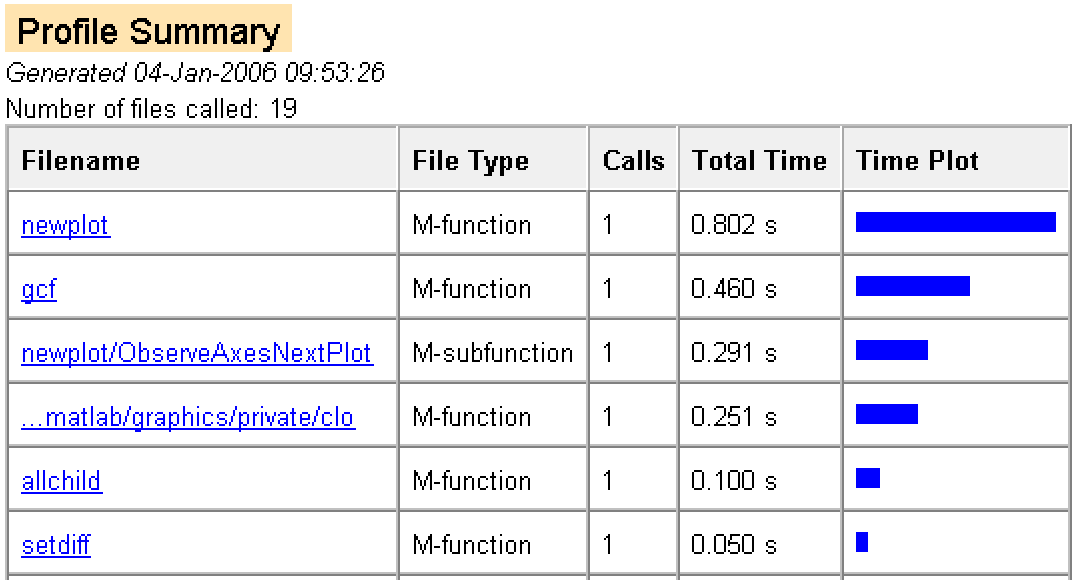
- You can time operations using **tic**/**toc**:

```
» tic
» CommandBlock1
» a=toc;
» CommandBlock2
» b=toc;
```

  - ➢ tic resets the timer
  - ➢ Each toc returns the current value in seconds
  - ➢ Can have multiple tocs per tic

Performance Measures

- For more complicated programs, use the profiler
  - » **`profile on`**
    - ➢ Turns on the profiler. Follow this with function calls
  - » **`profile viewer`**
    - ➢ Displays gui with stats on how long each subfunction took



Profile Summary

Generated 04-Jan-2006 09:53:26

Number of files called: 19

Filename	File Type	Calls	Total Time	Time Plot
newplot	M-function	1	0.802 s	▬▬▬▬▬▬
gcf	M-function	1	0.460 s	▬▬▬
newplot/ObserveAxesNextPlot	M-subfunction	1	0.291 s	▬▬
...matlab/graphics/private/clo	M-function	1	0.251 s	▬▬
allchild	M-function	1	0.100 s	▬
setdiff	M-function	1	0.050 s	▮

Courtesy of The MathWorks, Inc. Used with permission.

Revisiting find

- **find** is a very important function
  - Returns indices of nonzero values
  - Can simplify code and help avoid loops

- Basic syntax: index=find(cond)

```
» x=rand(1,100);
» inds = find(x>0.4 & x<0.6);
```

- `inds` will contain the indices at which x has values between 0.4 and 0.6. This is what happens:
  - x>0.4 returns a vector with 1 where true and 0 where false
  - x<0.6 returns a similar vector
  - The & combines the two vectors using an **and**
  - The find returns the indices of the 1's

Avoid for loops by vectorizing!

There are situations where a `for` loop is essential, as in many of the examples in this section so far. However, given the way MATLAB has been designed, `for` loops tend to be inefficient in terms of computing time. If you have written a `for` loop that involves the index of the loop in an expression, it may be possible to vectorize the expression, making use of array operations where necessary, as the following examples show.

Suppose you want to evaluate

$$\sum_{n=1}^{100\,000} n$$

(and can't remember the formula for the sum). Here's how to do it with a `for` loop (run the program, which also times how long it takes):

```
t0 = clock;
s = 0;
for n = 1:100000
    s = s + n;
end
etime(clock, t0)
```

The MATLAB function `clock` returns a six-element vector with the current date and time in the format year, month, day, hour, minute, seconds. Thus, `t0` records when the calculation starts.

The function `etime` returns the time in seconds elapsed between its two arguments, which must be vectors as returned by `clock`. On a Pentium II, it returned about 3.35 seconds, which is the total time for this calculation. (If you have a faster PC, it should take less time.)

Now try to vectorize this calculation (before looking at the solution). Here it is:

```
t0 = clock;
n = 1:100000;
s = sum( n );
etime(clock, t0)
```

This way takes only 0.06 seconds on the same PC—more than 50 times faster!

This way takes only 0.06 seconds on the same PC—more than 50 times faster!

There is a neater way of monitoring the time taken to interpret MATLAB statements: the `tic` and `toc` function. Suppose you want to evaluate

$$\sum_{n=1}^{100\,000} \frac{1}{n^2}$$

Here's the `for` loop version:

```
tic
s = 0;
for n = 1:100000
    s = s + 1/n^2;
end
toc
```

which takes about 6 seconds on the same PC. Once again, try to vectorize the sum:

```
tic
n = 1:100000;
s = sum( 1./n.^2 );
toc
```

The same PC gives a time of about 0.05 seconds for the vectorized version—more than 100 times faster! (Of course, the computation time in these examples is small regardless of the method applied. However, learning how to improve the efficiency of computation to solve more complex scientific or engineering problems will be helpful as you develop good programming skills.

Example: Avoiding Loops

- Given x= sin(linspace(0,10*pi,100)), how many of the entries are positive?

Using a loop and if/else

```
count=0;
for n=1:length(x)
    if x(n)>0
        count=count+1;
    end
end
```

Being more clever

```
count=length(find(x>0));
```

length(x)	Loop time	Find time
100	0.01	0
10,000	0.1	0
100,000	0.22	0
1,000,000	1.5	0.04

- Avoid loops!
- Built-in functions will make it faster to write and execute

Efficient Code

- Avoid loops
  - ➢ This is referred to as vectorization
- Vectorized code is more efficient for MATLAB
- Use indexing and matrix operations to avoid loops
- For example, to sum up every two consecutive terms:

```
» a=rand(1,100);
» b=zeros(1,100);
» for n=1:100
»     if n==1
»         b(n)=a(n);
»     else
»         b(n)=a(n-1)+a(n);
»     end
» end
```

➢ Slow and complicated

```
» a=rand(1,100);
» b=[0 a(1:end-1)]+a;
```

➢ Efficient and clean. Can also do this using `conv`

End of Section 2

(1) Functions

(2) Flow Control

(3) Vectorization

Introduction to programming in MATLAB

Section 3 : Solving Equations and Curve Fitting

Outline

(1) Linear Algebra

(2) Polynomials

(3) Optimization

(4) Differentiation/Integration

(5) Differential Equations

Systems of Linear Equations

- Given a system of linear equations
  - x+2y-3z=5
  - -3x-y+z=-8
  - x-y+z=0
- Construct matrices so the system is described by Ax=b

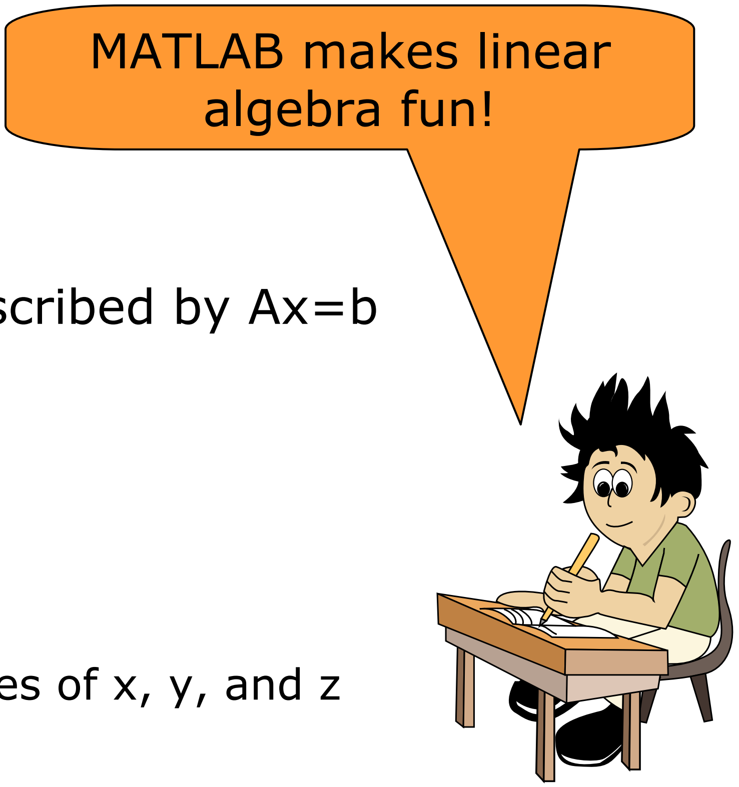
```
» A=[1 2 -3;-3 -1 1;1 -1 1];
» b=[5;-8;0];
```

- And solve with a single line of code!

```
» x=A\b;
```

  - x is a 3x1 vector containing the values of x, y, and z

- The \ will work with square or rectangular systems.
- Gives least squares solution for rectangular systems. Solution depends on whether the system is over or underdetermined.



More Linear Algebra

- Given a matrix
  - » `mat=[1 2 -3;-3 -1 1;1 -1 1];`
- Calculate the rank of a matrix
  - » `r=rank(mat);`
    - ➢ the number of linearly independent rows or columns
- Calculate the determinant
  - » `d=det(mat);`
    - ➢ mat must be square
    - ➢ if determinant is nonzero, matrix is invertible
- Get the matrix inverse
  - » `E=inv(mat);`
    - ➢ if an equation is of the form A*x=b with A a square matrix, x=A\b is the same as x=inv(A)*b

Exercise: Linear Algebra

- Solve the following systems of equations:

  - System 1:

$$\begin{aligned} x+4y&=34 \\ -3x+y&=2 \end{aligned}$$

  - System 2:

$$\begin{aligned} 2x-2y&=4 \\ -x+y&=3 \\ 3x+4y&=2 \end{aligned}$$

Exercise: Linear Algebra

- Solve the following systems of equations:

  - System 1:

$$x + 4y = 34$$
$$-3x + y = 2$$

```
» A=[1 4;-3 1];
» b=[34;2];
» rank(A)
» x=inv(A)*b;
```

  - System 2:

$$2x - 2y = 4$$
$$-x + y = 3$$
$$3x + 4y = 2$$

```
» A=[2 -2;-1 1;3 4];
» b=[4;3;2];
» rank(A)
```

  - rectangular matrix

```
» x1=A\b;
```

  - gives least squares solution

```
» error=abs(A*x1-b)
```

Outline

(1) Linear Algebra

(2) Polynomials

(3) Optimization

(4) Differentiation/Integration

(5) Differential Equations

Polynomials

- Many functions can be well described by a high-order polynomial

- MATLAB represents a polynomials by a vector of coefficients
    - ➢ if vector P describes a polynomial

$$\underset{\overset{\uparrow}{P(1)}}{a}x^3+\underset{\overset{\uparrow}{P(2)}}{b}x^2+\underset{\overset{\uparrow}{P(3)}}{c}x+\underset{\overset{\uparrow}{P(4)}}{d}$$

- P=[1 0 -2] represents the polynomial x^2-2

- P=[2 0 0 0] represents the polynomial $2x^3$

Polynomial Operations

- P is a vector of length N+1 describing an N-th order polynomial
- To get the roots of a polynomial
  - » `r=roots(P)`
    - ➢ r is a vector of length N
- Can also get the polynomial from the roots
  - » `P=poly(r)`
    - ➢ r is a vector length N
- To evaluate a polynomial at a point
  - » `y0=polyval(P,x0)`
    - ➢ x0 is a single value; y0 is a single value
- To evaluate a polynomial at many points
  - » `y=polyval(P,x)`
    - ➢ x is a vector; y is a vector of the same size

Polynomial Fitting

- MATLAB makes it very easy to fit polynomials to data

- Given data vectors X=[-1 0 2] and Y=[0 -1 3]

```
» p2=polyfit(X,Y,2);
```

  - ➢ finds the best second order polynomial that fits the points (-1,0),(0,-1), and (2,3)
  - ➢ see **help polyfit** for more information

```
» plot(X,Y,'o', 'MarkerSize', 10);
» hold on;
» x = -3:.01:3;
» plot(x,polyval(p2,x), 'r--');
```

Exercise: Polynomial Fitting

- Evaluate $y = x^2$ for x=-4:0.1:4.

- Add random noise to these samples. Use **randn**. Plot the noisy signal with **.** markers

- Fit a 2nd degree polynomial to the noisy data

- Plot the fitted polynomial on the same plot, using the same x values and a red line

Exercise: Polynomial Fitting

- Evaluate $y = x^2$ for x=-4:0.1:4.

```
» x=-4:0.1:4;
» y=x.^2;
```

- Add random noise to these samples. Use **randn**. Plot the noisy signal with **.** markers

```
» y=y+randn(size(y));
» plot(x,y,'.');
```

- Fit a 2nd degree polynomial to the noisy data

```
» p=polyfit(x,y,2);
```

- Plot the fitted polynomial on the same plot, using the same x values and a red line

```
» hold on;
» plot(x,polyval(p,x),'r')
```

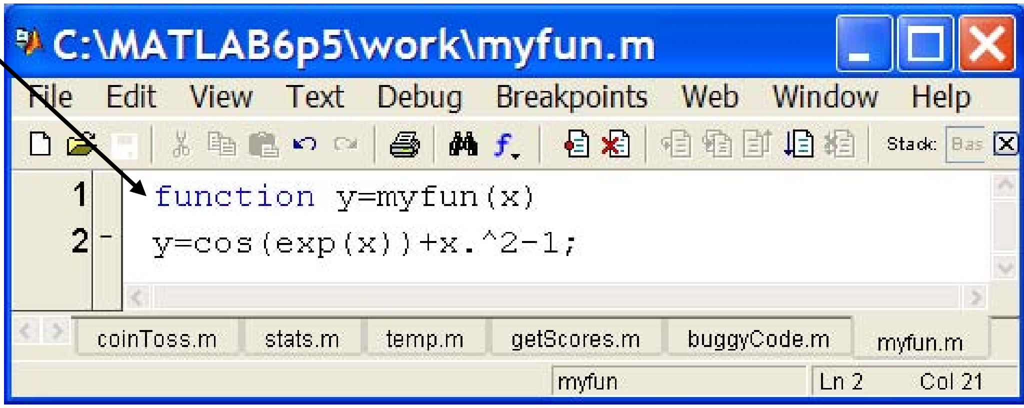
Outline

(1) Linear Algebra
(2) Polynomials
(3) Optimization
(4) Differentiation/Integration
(5) Differential Equations

Nonlinear Root Finding

- Many real-world problems require us to solve f(x)=0
- Can use **fzero** to calculate roots for *any* arbitrary function

- **fzero** needs a function passed to it.
- We will see this more and more as we delve into solving equations.
- Make a separate function file
  - » `x=fzero('myfun',1)`
  - » `x=fzero(@myfun,1)`
    - ➢ 1 specifies a point close to where you think the root is


C:\MATLAB6p5\work\myfun.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
coinToss.m stats.m temp.m getScores.m buggyCode.m myfun.m
myfun Ln 2 Col 21

```
function y=myfun(x)
y=cos(exp(x))+x.^2-1;
```

Courtesy of The MathWorks, Inc. Used with permission.

Anonymous Functions

- You do not have to make a separate function file
  - » `x=fzero(@myfun,1)`
    - ➢ What if myfun is really simple?

- Instead, you can make an anonymous function
  - » `x=fzero(@(x)(cos(exp(x))+x^2-1), 1 );`

$$\texttt{x=fzero(@(}\underset{\text{input}}{\uparrow\!\texttt{x}}\texttt{)}\underbrace{\texttt{(cos(exp(x))+x\^{}2-1)}}_{\text{function to evaluate}}\texttt{, 1);}$$

Exercise: Min-Finding

- Find the minimum of the function $f(x) = \cos(4x)\sin(10x)e^{-|x|}$ over the range −π to π. Use **fminbnd**.
- Plot the function on this range to check that this is the minimum.

Exercise: Min-Finding

- Find the minimum of the function $f(x)=\cos(4x)\sin(10x)e^{-|x|}$ over the range −π to π. Use `fminbnd`.
- Plot the function on this range to check that this is the minimum.

- Make the following function:

```
» function y=myFun(x)
» y=cos(4*x).*sin(10*x).*exp(-abs(x));
```

- Find the minimum in the command window:

```
» x0=fminbnd('myFun',-pi,pi);
```

- Plot to check if it's right

```
» figure; x=-pi:.01:pi; plot(x,myFun(x));
```

Outline

(1) Linear Algebra

(2) Polynomials

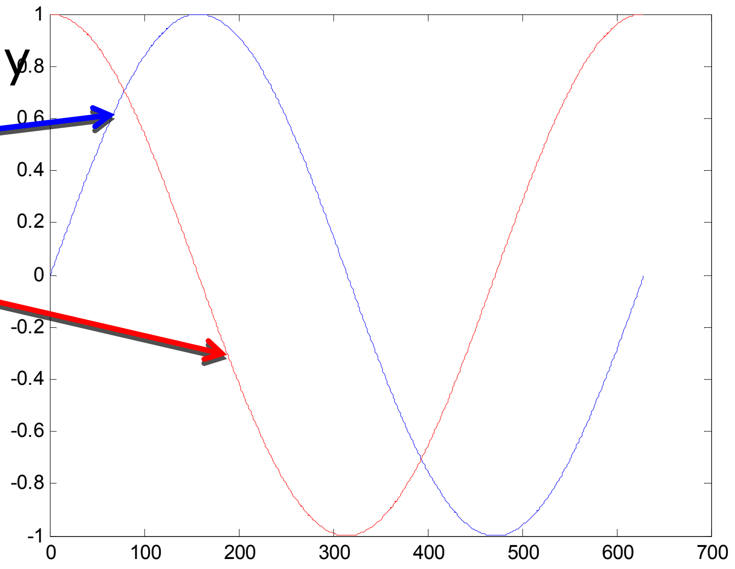
(3) Optimization

(4) Differentiation/Integration

(5) Differential Equations

Numerical Differentiation

- MATLAB can 'differentiate' numerically
  - » `x=0:0.01:2*pi;`
  - » `y=sin(x);`
  - » `dydx=diff(y)./diff(x);`
    - ➢ diff computes the first difference



- Can also operate on matrices
  - » `mat=[1 3 5;4 8 6];`
  - » `dm=diff(mat,1,2)`
    - ➢ first difference of mat along the 2nd dimension, dm=[2 2;4 -2]
    - ➢ see **help** for more details
    - ➢ The opposite of `diff` is the cumulative sum `cumsum`

- 2D gradient
  - » `[dx,dy]=gradient(mat);`

Numerical Integration

- MATLAB contains common integration methods

- Adaptive Simpson's quadrature (input is a function)
  - » `q=quad('myFun',0,10);`
    - ➢ q is the integral of the function `myFun` from 0 to 10
  - » `q2=quad(@(x) sin(x)*x,0,pi)`
    - ➢ q2 is the integral of `sin(x)*x` from 0 to pi
- Trapezoidal rule (input is a vector)
  - » `x=0:0.01:pi;`
  - » `z=trapz(x,sin(x));`
    - ➢ z is the integral of sin(x) from 0 to pi
  - » `z2=trapz(x,sqrt(exp(x))./x)`
    - ➢ z2 is the integral of $\sqrt{e^x}/x$ from 0 to pi

Outline

(1) Linear Algebra

(2) Polynomials

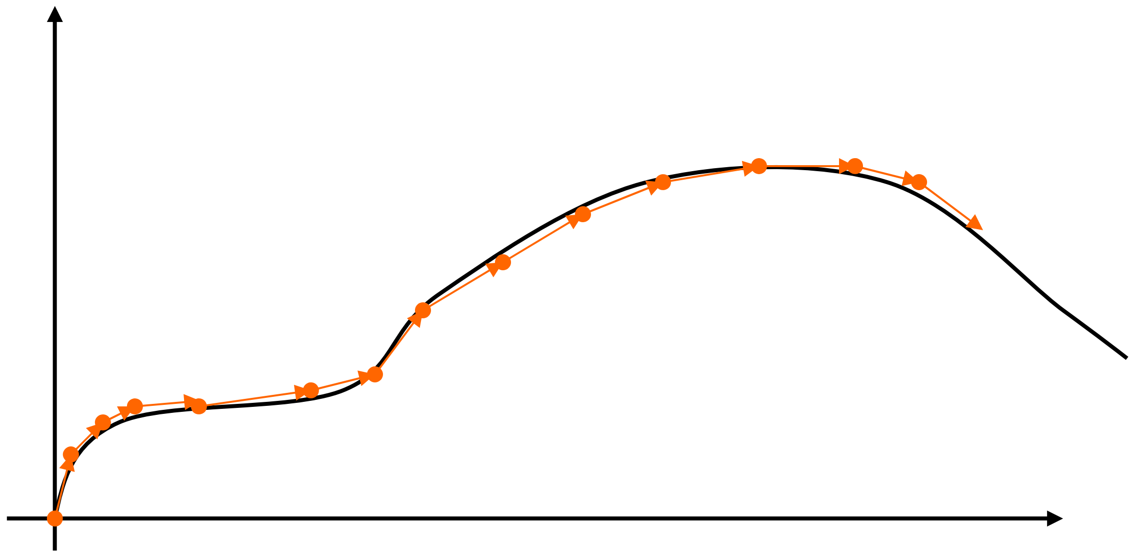
(3) Optimization

(4) Differentiation/Integration

(5) Differential Equations

ODE Solvers: Method

- Given a differential equation, the solution can be found by integration:



  - ➢ Evaluate the derivative at a point and approximate by straight line
  - ➢ Errors accumulate!
  - ➢ Variable timestep can decrease the number of iterations

ODE Solvers: MATLAB

- MATLAB contains implementations of common ODE solvers

- Using the correct ODE solver can save you lots of time and give more accurate results
  - » `ode23`
    - ➢ Low-order solver. Use when integrating over small intervals or when accuracy is less important than speed
  - » `ode45`
    - ➢ High order (Runge-Kutta) solver. High accuracy and reasonable speed. Most commonly used.
  - » `ode15s`
    - ➢ Stiff ODE solver (Gear's algorithm), use when the diff eq's have time constants that vary by orders of magnitude

ODE Solvers: Standard Syntax

- To use standard options and variable time step

```
» [t,y]=ode45('myODE',[0,10],[1;0])
```

ODE integrator:
23, 45, 15s

ODE function

Time range

Initial conditions

- Inputs:
  - ➢ ODE function name (or anonymous function). This function takes inputs (t,y), and returns dy/dt
  - ➢ Time interval: 2-element vector specifying initial and final time
  - ➢ Initial conditions: column vector with an initial condition for each ODE. This is the first input to the ODE function
- Outputs:
  - ➢ t contains the time points
  - ➢ y contains the corresponding values of the integrated variables.

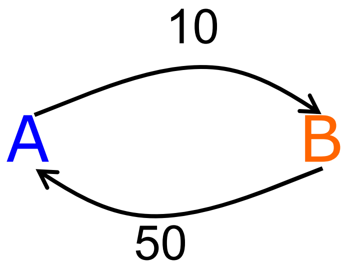
ODE Function

- The ODE function must return the value of the derivative at a given time and function value

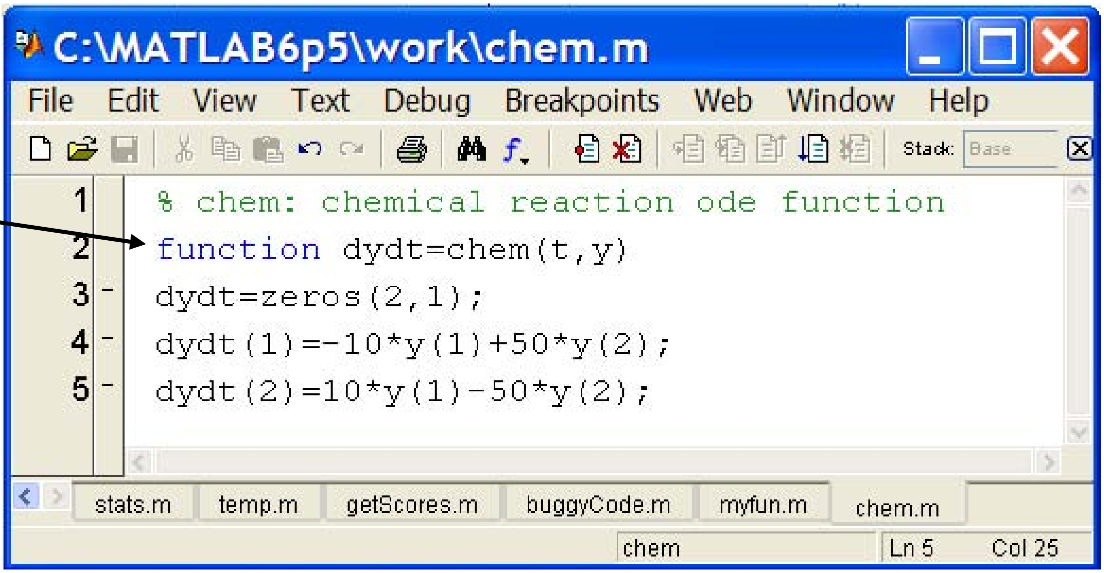
- Example: chemical reaction
  - Two equations

$$\frac{dA}{dt} = -10A + 50B$$

$$\frac{dB}{dt} = 10A - 50B$$



  - ODE file:
    - y has [A;B]
    - dydt has [dA/dt;dB/dt]



C:\MATLAB6p5\work\chem.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
stats.m temp.m getScores.m buggyCode.m myfun.m chem.m
chem Ln 5 Col 25

```
% chem: chemical reaction ode function
function dydt=chem(t,y)
dydt=zeros(2,1);
dydt(1)=-10*y(1)+50*y(2);
dydt(2)=10*y(1)-50*y(2);
```

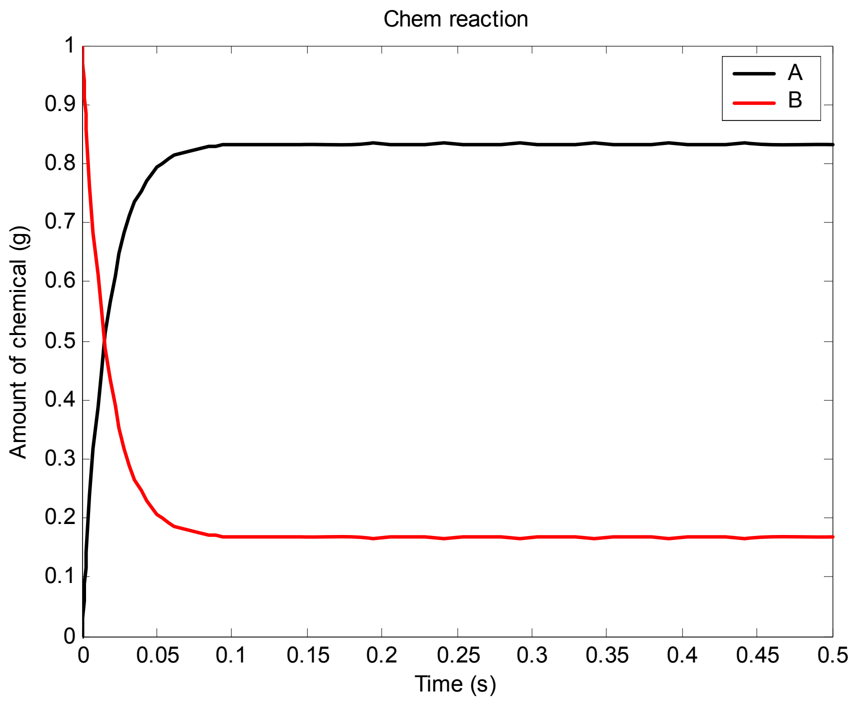
Courtesy of The MathWorks, Inc. Used with permission.

ODE Function: viewing results

- To solve and plot the ODEs on the previous slide:
  - » `[t,y]=ode45('chem',[0 0.5],[0 1]);`
    - ➢ assumes that only chemical B exists initially
  - » `plot(t,y(:,1),'k','LineWidth',1.5);`
  - » `hold on;`
  - » `plot(t,y(:,2),'r','LineWidth',1.5);`
  - » `legend('A','B');`
  - » `xlabel('Time (s)');`
  - » `ylabel('Amount of chemical (g)');`
  - » `title('Chem reaction');`

ODE Function: viewing results

- The code on the previous slide produces this figure



Higher Order Equations

- Must make into a system of first-order equations to use ODE solvers
- Nonlinear is OK!
- Pendulum example:

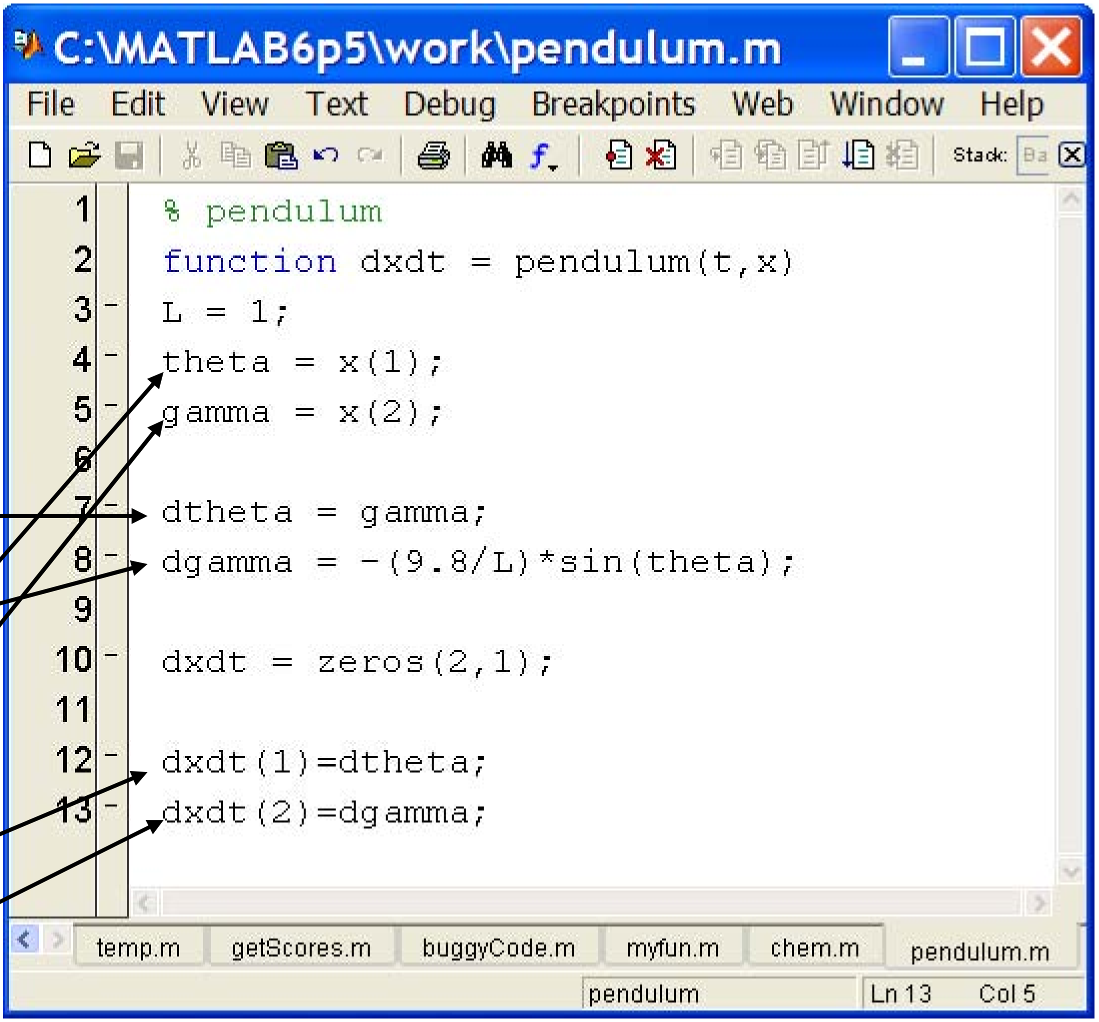
$$\ddot{\theta} + \frac{g}{L} sin(\theta) = 0$$

$$\ddot{\theta} = -\frac{g}{L} sin(\theta)$$

$$let \quad \dot{\theta} = \gamma$$

$$\dot{\gamma} = -\frac{g}{L} sin(\theta)$$

$$\boxed{\vec{x} = \begin{bmatrix} \theta \\ \gamma \end{bmatrix} \quad \frac{d\vec{x}}{dt} = \begin{bmatrix} \dot{\theta} \\ \dot{\gamma} \end{bmatrix}}$$


C:\MATLAB6p5\work\pendulum.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Ba
temp.m getScores.m buggyCode.m myfun.m chem.m pendulum.m
pendulum Ln 13 Col 5

```
% pendulum
function dxdt = pendulum(t,x)
L = 1;
theta = x(1);
gamma = x(2);

dtheta = gamma;
dgamma = -(9.8/L)*sin(theta);

dxdt = zeros(2,1);

dxdt(1)=dtheta;
dxdt(2)=dgamma;
```

Courtesy of The MathWorks, Inc. Used with permission.

Plotting the Output

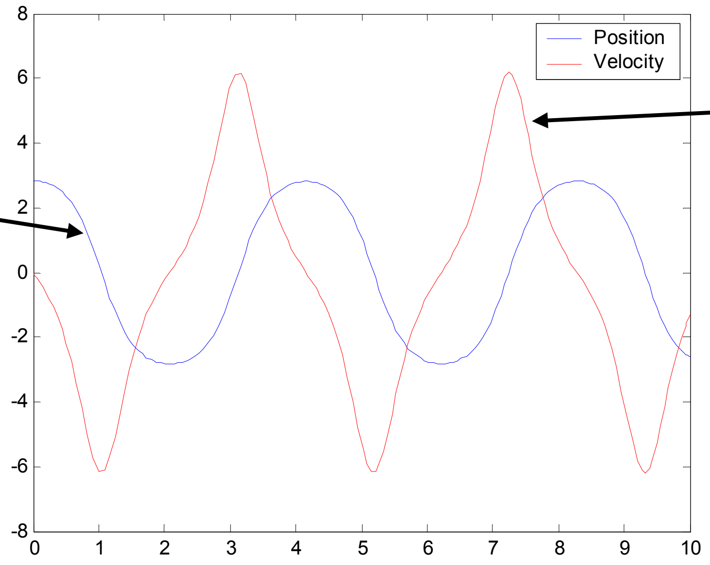
- We can solve for the position and velocity of the pendulum:

```
» [t,x]=ode45('pendulum',[0 10],[0.9*pi 0]);
```

  - ➢ assume pendulum is almost horizontal

```
» plot(t,x(:,1));
» hold on;
» plot(t,x(:,2),'r');
» legend('Position','Velocity');
```

Position in terms of angle (rad)



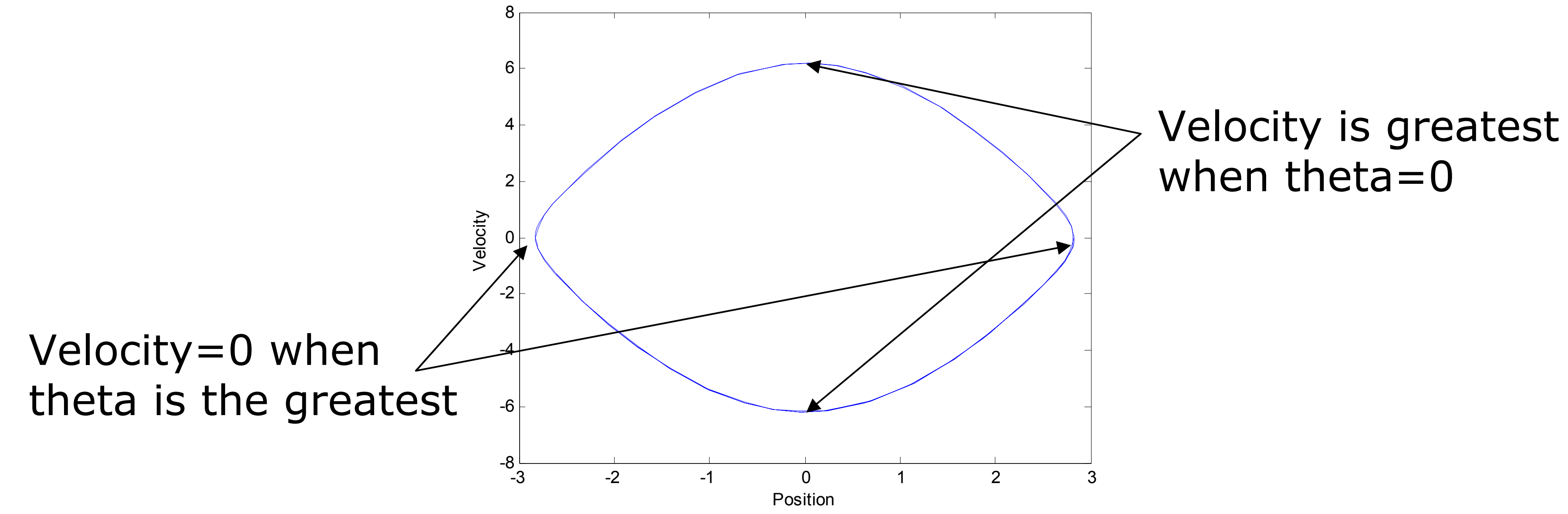
Velocity (m/s)

Plotting the Output

- Or we can plot in the phase plane:

```
» plot(x(:,1),x(:,2));
» xlabel('Position');
» yLabel('Velocity');
```

- The phase plane is just a plot of one variable versus the other:



ODE Solvers: Custom Options

- MATLAB's ODE solvers use a variable timestep
- Sometimes a fixed timestep is desirable

```
» [t,y]=ode45('chem',[0:0.001:0.5],[0 1]);
```

  - ➢ Specify the timestep by giving a vector of times
  - ➢ The function value will be returned at the specified points
  - ➢ Fixed timestep is usually slower because function values are interpolated to give values at the desired timepoints

- You can customize the error tolerances using odeset

```
» options=odeset('RelTol',1e-6,'AbsTol',1e-10);
» [t,y]=ode45('chem',[0 0.5],[0 1],options);
```

  - ➢ This guarantees that the error at each step is less than `RelTol` times the value at that step, and less than `AbsTol`
  - ➢ Decreasing error tolerance can considerably slow the solver
  - ➢ See doc odeset for a list of options you can customize

Exercise: ODE

- Use `ode45` to solve for $y(t)$ on the range t=[0 10], with initial condition $y(0)=10$ and $dy/dt=-t\,y/10$
- Plot the result.

Exercise: ODE

- Use `ode45` to solve for $y(t)$ on the range t=[0 10], with initial condition $y(0)=10$ and $dy/dt=-t\,y/10$
- Plot the result.

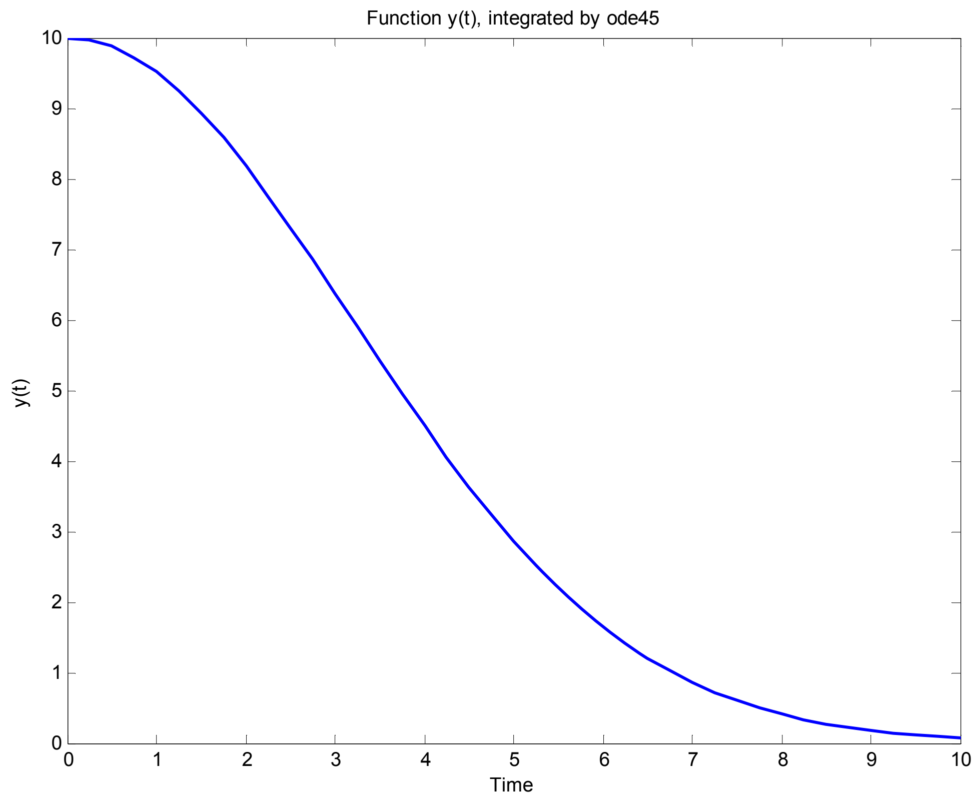
- Make the following function
  - » `function dydt=odefun(t,y)`
  - » `dydt=-t*y/10;`
- Integrate the ODE function and plot the result
  - » `[t,y]=ode45(‘odefun’,[0 10],10);`

- Alternatively, use an anonymous function
  - » `[t,y]=ode45(@(t,y) -t*y/10,[0 10],10);`

- Plot the result
  - » `plot(t,y);xlabel('Time');ylabel('y(t)');`

Exercise: ODE

- The integrated function looks like this:



End of Section 3

(1) Linear Algebra

(2) Polynomials

(3) Optimization

(4) Differentiation/Integration

(5) Differential Equations

MATLAB Programming

Section 3: Data Structures, Symbolics, Simulink®, File I/O, Building GUIs and Online Resources

Outline

(1) Data Structures

(2) Symbolic Math

(3) Simulink

(4) File I/O

(5) Graphical User Interfaces

(6) Online Resources

Advanced Data Structures

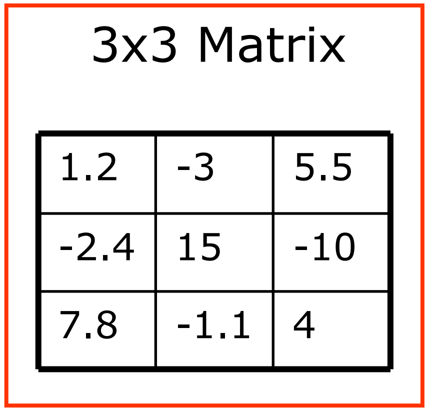
- We have used 2D matrices
  - ➢ Can have n-dimensions
  - ➢ Every element must be the same type (ex. integers, doubles, characters…)
  - ➢ Matrices are space-efficient and convenient for calculation
  - ➢ Large matrices with many zeros can be made sparse:

```
» a=zeros(100); a(1,3)=10;a(21,5)=pi; b=sparse(a);
```

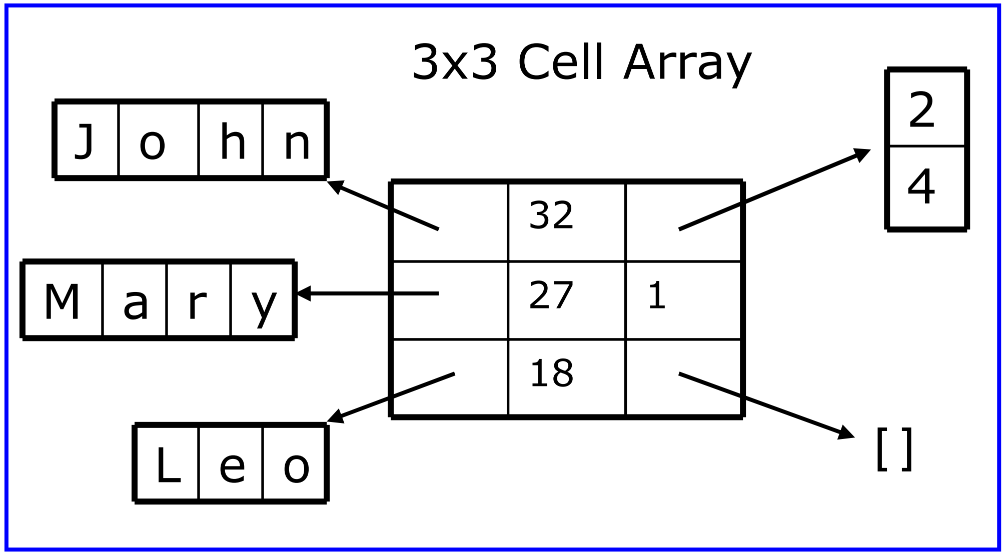
- Sometimes, more complex data structures are more appropriate
  - ➢ **Cell array**: it's like an array, but elements don't have to be the same type
  - ➢ **Structs**: can bundle variable names and values into one structure
    - – Like object oriented programming in MATLAB

Cells: organization

- A cell is just like a matrix, but each field can contain anything (even other matrices):



1.2	-3	5.5
-2.4	15	-10
7.8	-1.1	4



- One cell can contain people's names, ages, and the ages of their children
- To do the same with matrices, you would need 3 variables and padding

Cells: initialization

- To initialize a cell, specify the size
  ```
  » a=cell(3,10);
  ```
  - a will be a cell with 3 rows and 10 columns

- or do it manually, with curly braces {}
  ```
  » c={'hello world',[1 5 6 2],rand(3,2)};
  ```
  - c is a cell with 1 row and 3 columns

- Each element of a cell can be anything

- To access a cell element, use curly braces {}
  ```
  » a{1,1}=[1 3 4 -10];
  » a{2,1}='hello world 2';
  » a{1,2}=c{3};
  ```

Structs

- Structs allow you to name and bundle relevant variables
  - ➢ Like C-structs, which are objects with fields

- To initialize an empty struct:

```
» s=struct([]);
```

  - ➢ size(s) will be 0x0
  - ➢ initialization is optional but is recommended when using large structs

- To add fields

```
» s.name = 'Jack Bauer';
» s.scores = [95 98 67];
» s.year = 'G3';
```

  - ➢ Fields can be anything: matrix, cell, even struct
  - ➢ Useful for keeping variables together

- For more information, see **doc struct**

Struct Arrays

- To initialize a struct array, give field, values pairs

```
» ppl=struct('name',{'John','Mary','Leo'},...
  'age',{32,27,18},'childAge',{[2;4],1,[]});
```

  - ➢ size(s2)=1x3
  - ➢ every cell must have the same size

```
» person=ppl(2);
```

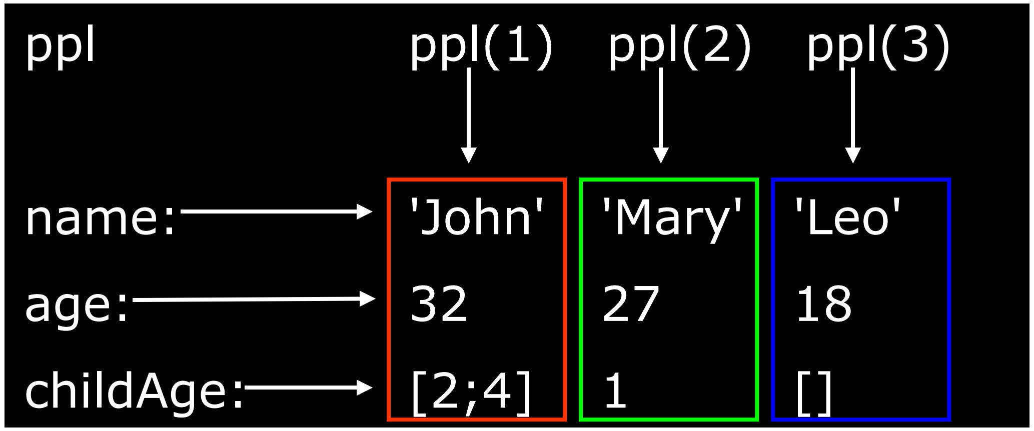
  - ➢ person is now a struct with fields name, age, children
  - ➢ the values of the fields are the second index into each cell

```
» person.name
```

  - ➢ returns 'Mary'

```
» ppl(1).age
```

  - ➢ returns 32



Structs: access

- To access 1x1 struct fields, give name of the field
  - » `stu=s.name;`
  - » `scor=s.scores;`
    - ➢ 1x1 structs are useful when passing many variables to a function. put them all in a struct, and pass the struct

- To access nx1 struct arrays, use indices
  - » `person=ppl(2);`
    - ➢ person is a struct with name, age, and child age
  - » `personName=ppl(2).name;`
    - ➢ personName is 'Mary'
  - » `a=[ppl.age];`
    - ➢ a is a 1x3 vector of the ages; this may not always work, the vectors must be able to be concatenated.

Exercise: Cells

- Write a script called `sentGen`
- Make a 3x2 cell, and put three **names** into the first column, and **adjectives** into the second column
- Pick two random integers (values 1 to 3)
- Display a sentence of the form '[name] is [adjective].'
- Run the script a few times

Exercise: Cells

- Write a script called `sentGen`
- Make a 3x2 cell, and put three **names** into the first column, and **adjectives** into the second column
- Pick two random integers (values 1 to 3)
- Display a sentence of the form '[name] is [adjective].'
- Run the script a few times

```
» c=cell(3,2);
» c{1,1}='John';c{2,1}='Mary-Sue';c{3,1}='Gomer';
» c{1,2}='smart';c{2,2}='blonde';c{3,2}='hot'
» r1=ceil(rand*3);r2=ceil(rand*3);
» disp([ c{r1,1}, ' is ', c{r2,2}, '.' ]);
```

Outline

(1) Data Structures

(2) Symbolic Math

(3) Simulink

(4) File I/O

(5) Graphical User Interfaces

(6) Online Resources

Symbolic Math Toolbox

- Symbolics vs. Numerics

	Advantages	Disadvantages
Symbolic	•Analytical solutions •Lets you intuit things about solution form	•Sometimes can't be solved •Can be overly complicated
Numeric	•Always get a solution •Can make solutions accurate •Easy to code	•Hard to extract a deeper understanding •Num. methods sometimes fail •Can take a while to compute

Symbolic Variables

- To make symbolic variables, use **`sym`**
  - » **`a=sym('1/3');`**
  - » **`b=sym('4/5');`**
  - » **`mat=sym([1 2;3 4]);`**
    - ➢ fractions remain as fractions
  - » **`c=sym('c','positive');`**
    - ➢ can add tags to narrow down scope
    - ➢ see **help sym** for a list of tags

- Or use **`syms`**
  - » **`syms x y real`**
    - ➢ shorthand for x=sym('x','real'); y=sym('y','real');

Symbolic Expressions

```
» d=a*b
```

- ➢ does 1/3*4/5=4/15;

```
d =
4/15
```

```
» expand((a-c)^2);
```

- ➢ multiplies out

```
ans =
1/9-2/3*c+c^2
```

```
» factor(ans)
```

- ➢ factors the expression

```
ans =
1/9*(3*c-1)^2
```

```
» matInv=inv(mat)
```

- ➢ Computes inverse symbolically

```
ans =
[    -2,      1]
[   3/2,   -1/2]
```

Cleaning up Symbolic Statements

- makes it look nicer

```
                2
1/9 - 2/3 c + c
```

» **collect(3*x+4*y-1/3*x^2-x+3/2*y)**

- collects terms

```
ans =
2*x+11/2*y-1/3*x^2
```

» **simplify(cos(x)^2+sin(x)^2)**

- simplifies expressions

```
ans =
1
```

» **subs(‘c^2’,c,5)**

```
ans=
25
```

- Replaces variables with numbers or expressions. To do multiple substitutions pass a cell of variable names followed by a cell of values

» **subs(‘c^2’,c,x/7)**

```
ans=
1/49*x^2
```

More Symbolic Operations

```
» mat=sym('[a b;c d]');
```

```
» mat2=mat*[1 3;4 -2];
```

- compute the product

```
mat2 =
[    a+4*b, 3*a-2*b]
[    c+4*d, 3*c-2*d]
```

```
» d=det(mat)
```

- compute the determinant

```
d =
a*d-b*c
```

```
» i=inv(mat)
```

- find the inverse

```
i =
[  d/(a*d-b*c), -b/(a*d-b*c)]
[ -c/(a*d-b*c),  a/(a*d-b*c)]
```

- You can access symbolic matrix elements as before

```
» i(1,2)
```

```
ans =
-b/(a*d-b*c)
```

Exercise: Symbolics

given by: $(x-a)^2+(y-b)^2=r^2$

- Use **`solve`** to solve this equation for x and then for y

- It's always annoying to integrate by parts. Use **`int`** to do the following integral symbolically and then compute the value by **`subs`**tituting 0 for a and 2 for b: $\int_a^b xe^x dx$

Exercise: Symbolics

given by: $\left(x-a\right)^2+\left(y-b\right)^2=r^2$

- Use **solve** to solve this equation for x and then for y

```
» syms a b r x y
» solve('(x-a)^2+(y-b)^2=r^2','x')
» solve('(x-a)^2+(y-b)^2=r^2','y')
```

- It's always annoying to integrate by parts. Use **int** to do the following integral symbolically and then compute the value by **subs**tituting 0 for a and 2 for b:

$$\int_a^b xe^x dx$$

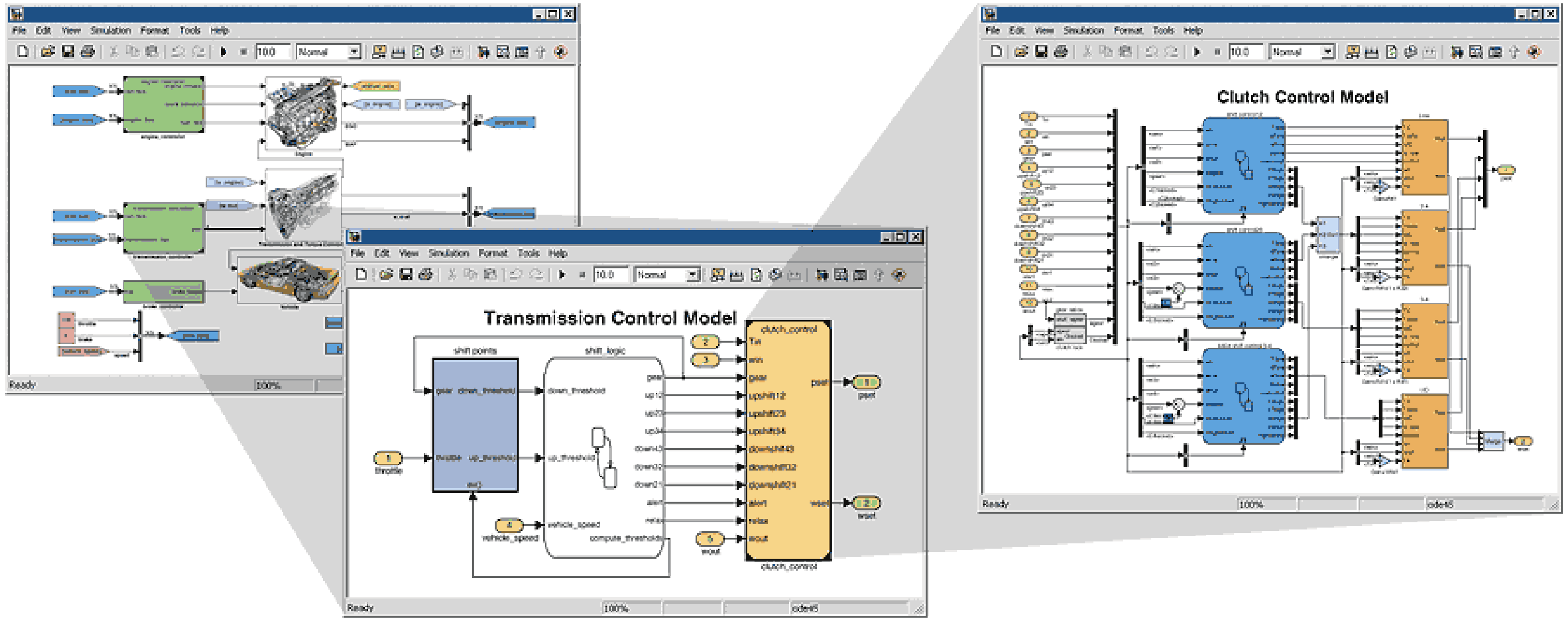
```
» Q=int(x*exp(x),a,b)
» subs(Q,{a,b},{0,2})
```

Outline

(1) Data Structures
(2) Symbolic Math
(3) Simulink
(4) File I/O
(5) Graphical User Interfaces
(6) Online Resources

SIMULINK

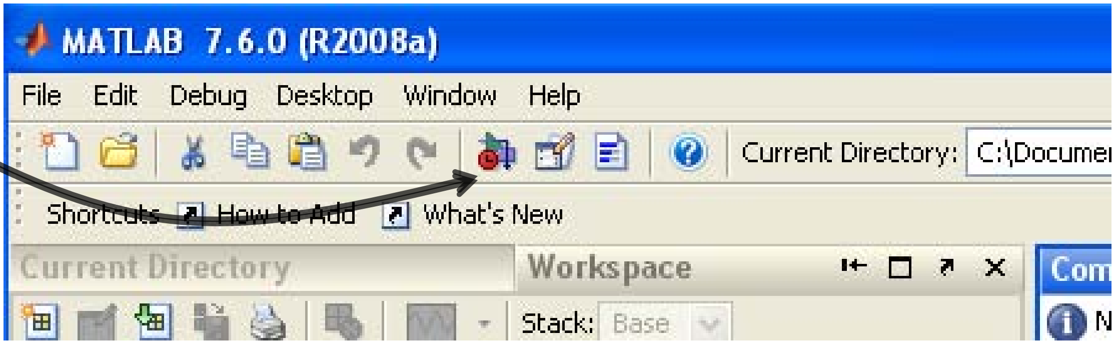
- Interactive graphical environment
- Block diagram based MATLAB add-on environment
- Design, simulate, implement, and test control, signal processing, communications, and other time-varying systems



Courtesy of The MathWorks, Inc. Used with permission.

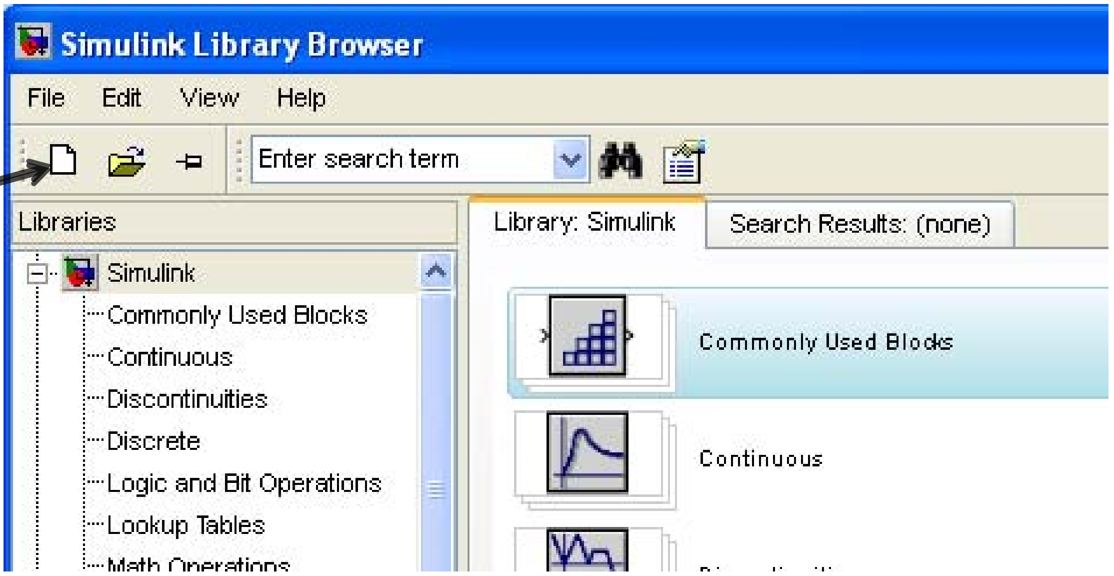
Getting Started

- In MATLAB, Start Simulink


MATLAB 7.6.0 (R2008a)
File Edit Debug Desktop Window Help
Current Directory: C:\Documer
Shortcuts How to Add What's New
Current Directory
Workspace
Stack: Base

Courtesy of The MathWorks, Inc. Used with permission.

- Create a new Simulink file, similar to how you make a new script

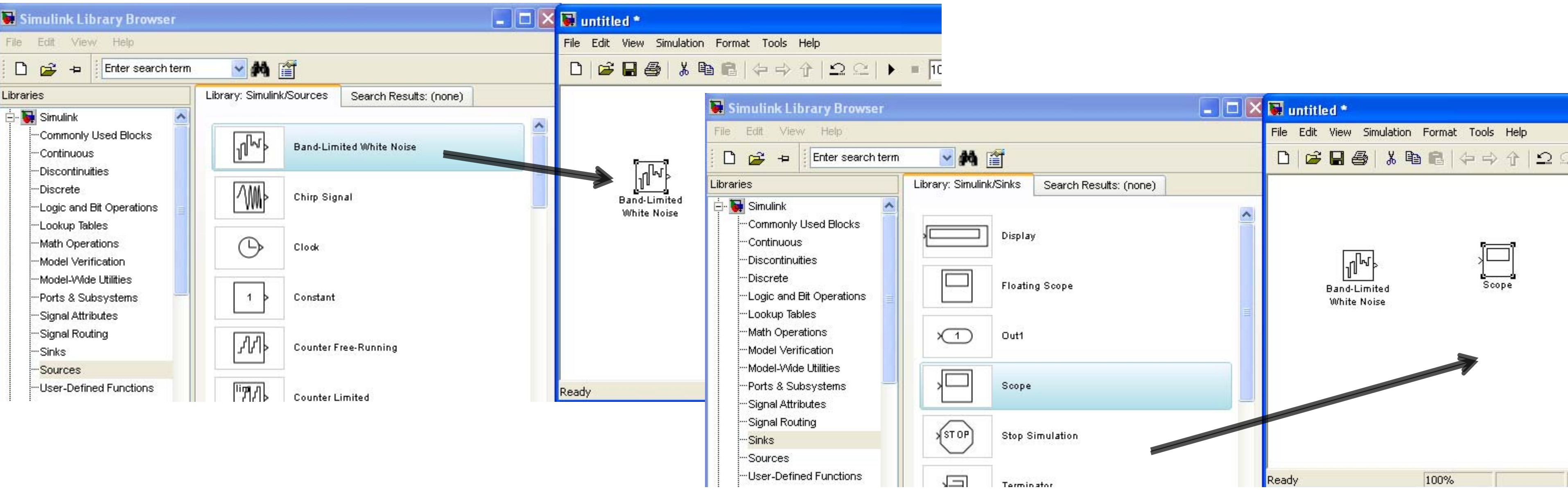


Courtesy of The MathWorks, Inc. Used with permission.

Simulink Library Browser

The **Library Browser** contains various blocks that you can put into your model

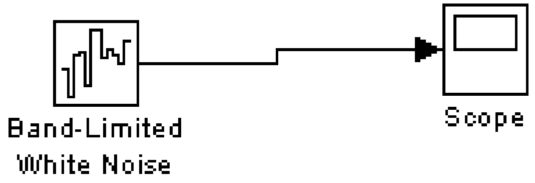
- Examine some blocks:
  - Click on a library: "Sources"
    - Drag a block into Simulink: "Band limited white noise"
  - Visualize the block by going into "Sinks"
    - Drag a "Scope" into Simulink



Courtesy of The MathWorks, Inc. Used with permission.

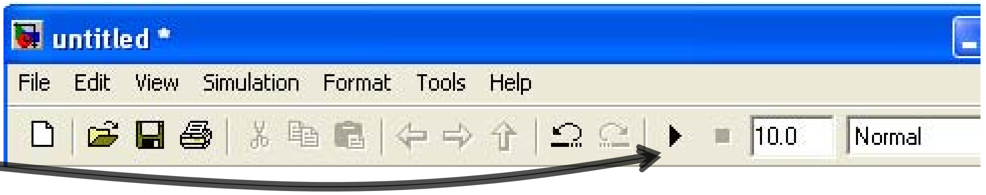
Connections

- Click on the carat/arrow on the right of the band limited white noise box



- Drag the line to the scope
  - You'll get a hint saying you can quickly connect blocks by hitting Ctrl
  - Connections between lines represent signals

- Click the play button

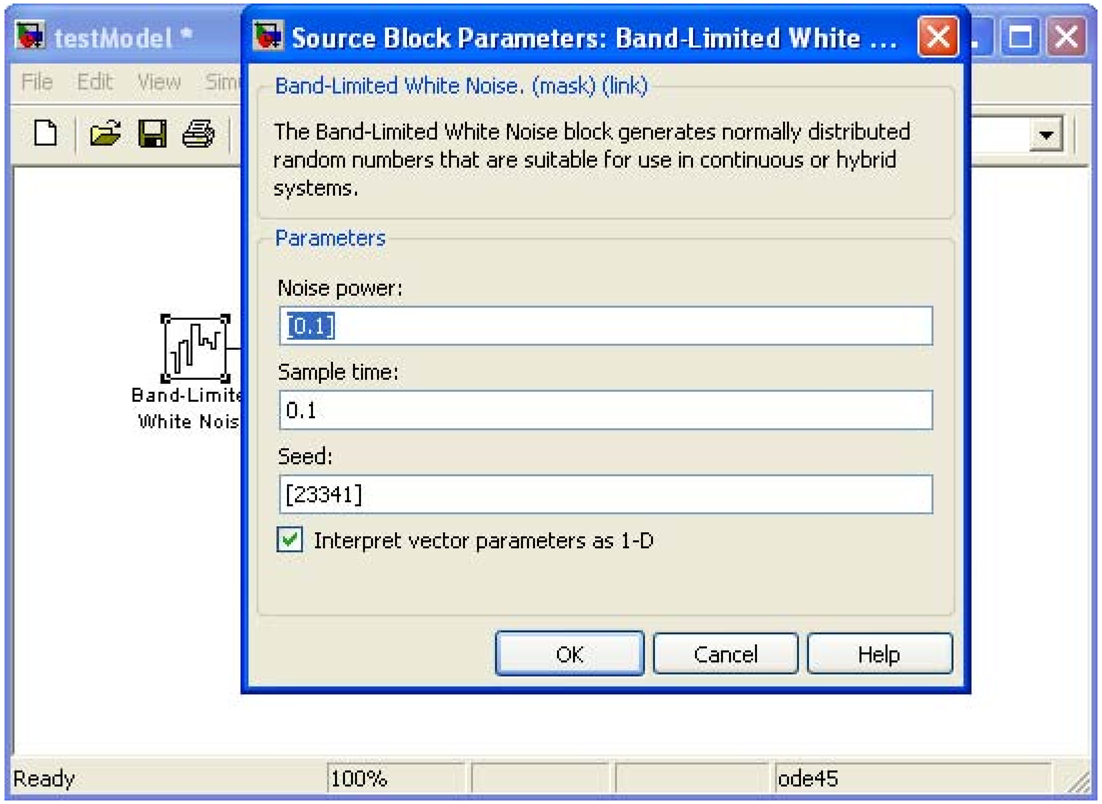


Courtesy of The MathWorks, Inc. Used with permission.

- Double click on the scope.
  - This will open up a chart of the variable over the simulation time

Connections, Block Specification

- To split connections, hold down ‘Ctrl’when clicking on a connection, and drag it to the target block; or drag backwards from the target block
- To modify properties of a block, double-click it and fill in the property values.



Source Block Parameters: Band-Limited White ...

Band-Limited White Noise. (mask) (link)

The Band-Limited White Noise block generates normally distributed random numbers that are suitable for use in continuous or hybrid systems.

Parameters

Noise power: [0.1]

Sample time: 0.1

Seed: [23341]

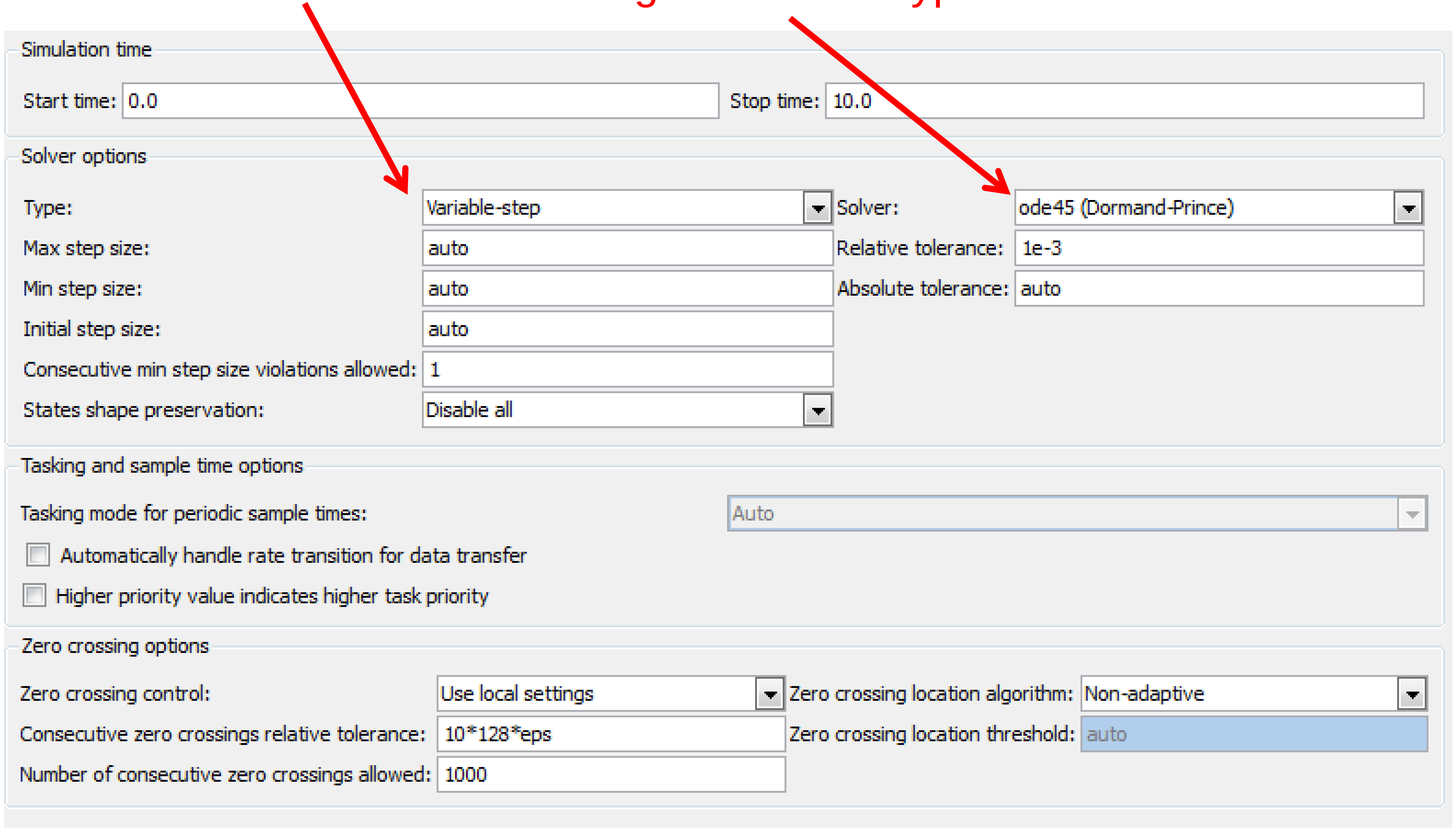
✓ Interpret vector parameters as 1-D

Courtesy of The MathWorks, Inc. Used with permission.

Behind the curtain

- Go to "Simulation"->"Configuration Parameters" at the top menu

See ode45? Change the solver type here



Simulation time

Start time: 0.0 | Stop time: 10.0

Solver options

Type:	Variable-step	Solver:	ode45 (Dormand-Prince)
Max step size:	auto	Relative tolerance:	1e-3
Min step size:	auto	Absolute tolerance:	auto
Initial step size:	auto		
Consecutive min step size violations allowed:	1		
States shape preservation:	Disable all		

Tasking and sample time options

Tasking mode for periodic sample times: Auto

☐ Automatically handle rate transition for data transfer

☐ Higher priority value indicates higher task priority

Zero crossing options

Zero crossing control:	Use local settings	Zero crossing location algorithm:	Non-adaptive
Consecutive zero crossings relative tolerance:	10*128*eps	Zero crossing location threshold:	auto
Number of consecutive zero crossings allowed:	1000		

Exercise: Simulink

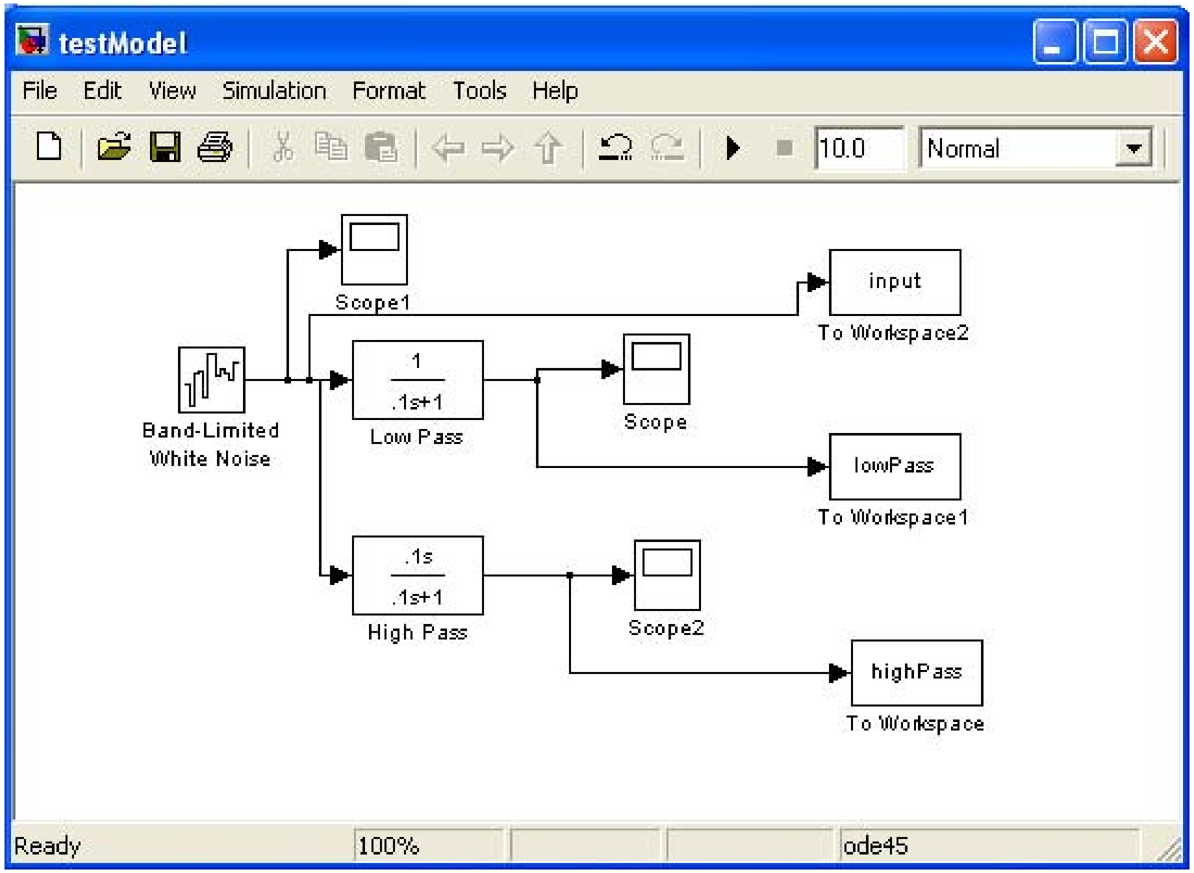
• Take your white noise signal, and split it into high frequency and low frequency components. Use the **Transfer Function** block from **Continuous** and use these transfer functions:

$$LP = \frac{1}{0.1s+1} \qquad HP = \frac{0.1s}{0.1s+1}$$

- Hook up scopes to the input and the two outputs
- Send the two outputs to the workspace by using the **to Workspace** block from **Sink**

Exercise: Simulink

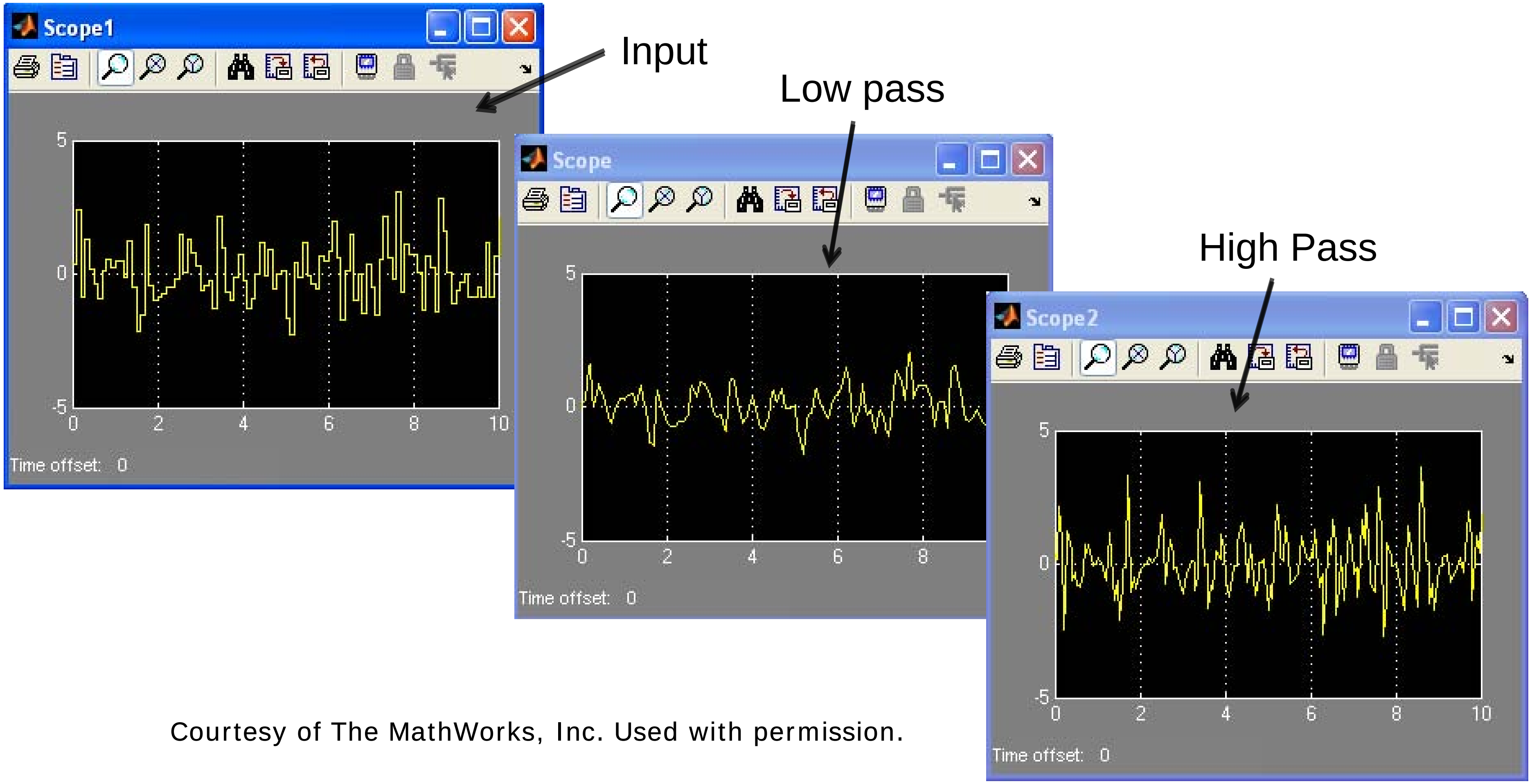
- The diagram should look like this. To change the **transfer function** parameters, double click the blocks and specify the numerator and denominator as polynomials in s (remember how we defined polynomial vectors before)



Courtesy of The MathWorks, Inc. Used with permission.

Exercise: Simulink

- After running the simulation, double-clicking the scopes will show:



Courtesy of The MathWorks, Inc. Used with permission.

Toolboxes

- Math
    - ➢ Takes the signal and performs a math operation
        - » **`Add, subtract, round, multiply, gain, angle`**
- Continuous
    - ➢ Adds differential equations to the system
        - » **`Integrals, Derivatives, Transfer Functions, State Space`**
- Discontinuities
    - ➢ Adds nonlinearities to your system
- Discrete
    - ➢ Simulates discrete difference equations
    - ➢ Useful for digital systems

Building systems

- Sources
  - » **Step input, white noise, custom input, sine wave, ramp input,**
    - ➢ Provides input to your system
- Sinks
  - » **Scope: Outputs to plot**
  - » **simout: Outputs to a MATLAB vector on workspace**
  - » **MATLAB mat file**

Outline

(1) Data Structures
(2) **Symbolic Math**
(3) **Simulink**
(4) **File I/O**
(5) **Graphical User Interfaces**
(6) **Online Resources**

Importing Data

- MATLAB is a great environment for processing data. If you have a text file with some data:

```
jane joe jimmy
10 11 12
5 4 2
5 6 4
```

- To import data from files on your hard drive, use importdata

```
» a=importdata('textFile.txt');
```

  - a is a struct with data, textdata, and colheaders fields

```
a =
         data: [3x3 double]
     textdata: {'jane'   'joe'   'jimmy'}
   colheaders: {'jane'   'joe'   'jimmy'}
```

```
» x=a.data;
» names=a.colheaders;
```

Importing Data

- With **`importdata`**, you can also specify delimiters. For example, for comma separated values, use:
  - » **`a=importdata('filename', ', ');`**
    - ➢ The second argument tells matlab that the tokens of interest are separated by commas or spaces

- **`importdata`** is very robust, but sometimes it can have trouble. To read files with more control, use **`fscanf`** (similar to C/Java), **`textread`**, **`textscan`**. See **help** or **doc** for information on how to use these functions

Writing Excel Files

- MATLAB contains specific functions for reading and writing Microsoft Excel files
- To write a matrix to an Excel file, use **`xlswrite`**

```
» [s,m]=xlswrite('randomNumbers',rand(10,4),...
  'Sheet1'); % we specify the sheet name
```

- You can also write a cell array if you have mixed data:

```
» C={'hello','goodbye';10,-2;-3,4};
» [s,m]=xlswrite('randomNumbers',C,'mixedData');
```

- **`s`** and **`m`** contain the 'success' and 'message' output of the write command
- See **`doc xlswrite`** for more usage options

Reading Excel Files

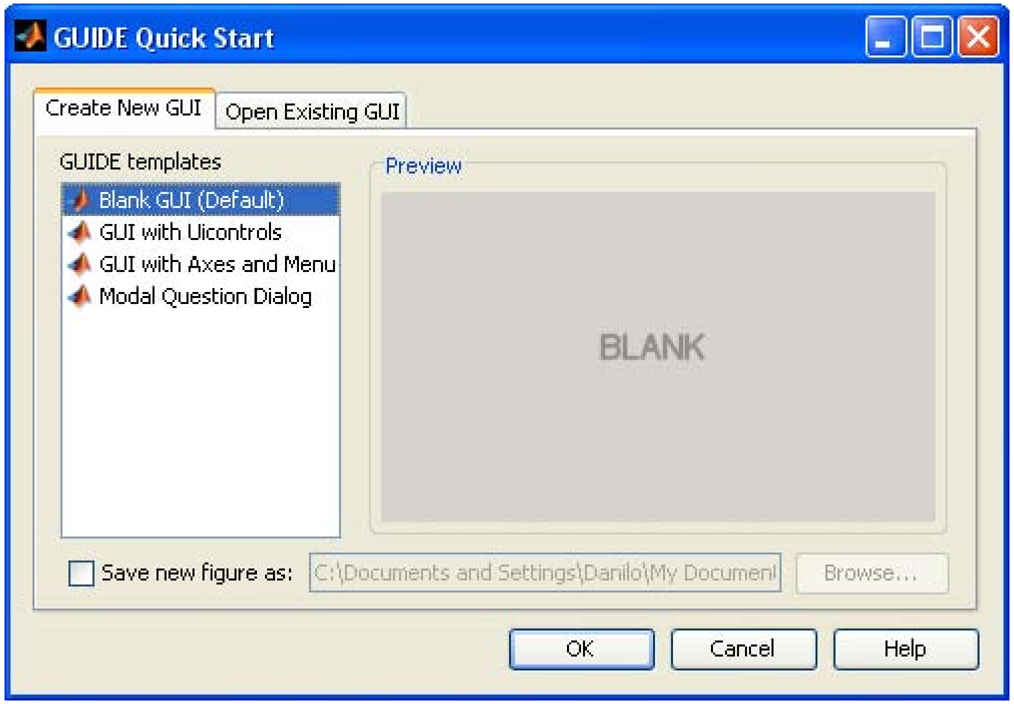
- Reading excel files is equally easy
- To read from an Excel file, use **`xlsread`**
  - » **`[num,txt,raw]=xlsread('randomNumbers.xls');`**
    - ➢ Reads the first sheet
    - ➢ **`num`** contains numbers, **`txt`** contains strings, **`raw`** is the entire cell array containing everything
  - » **`[num,txt,raw]=xlsread('randomNumbers.xls',...`**
    `'mixedData');`
    - ➢ Reads the **mixedData** sheet
  - » **`[num,txt,raw]=xlsread('randomNumbers.xls',-1);`**
    - ➢ Opens the file in an Excel window and lets you click on the data you want!
- See **`doc xlsread`** for even more fancy options

Outline

(1) Data Structures
(2) Symbolic Math
(3) Simulink
(4) File I/O
(5) Graphical User Interfaces
(6) Online Resources

Making GUIs

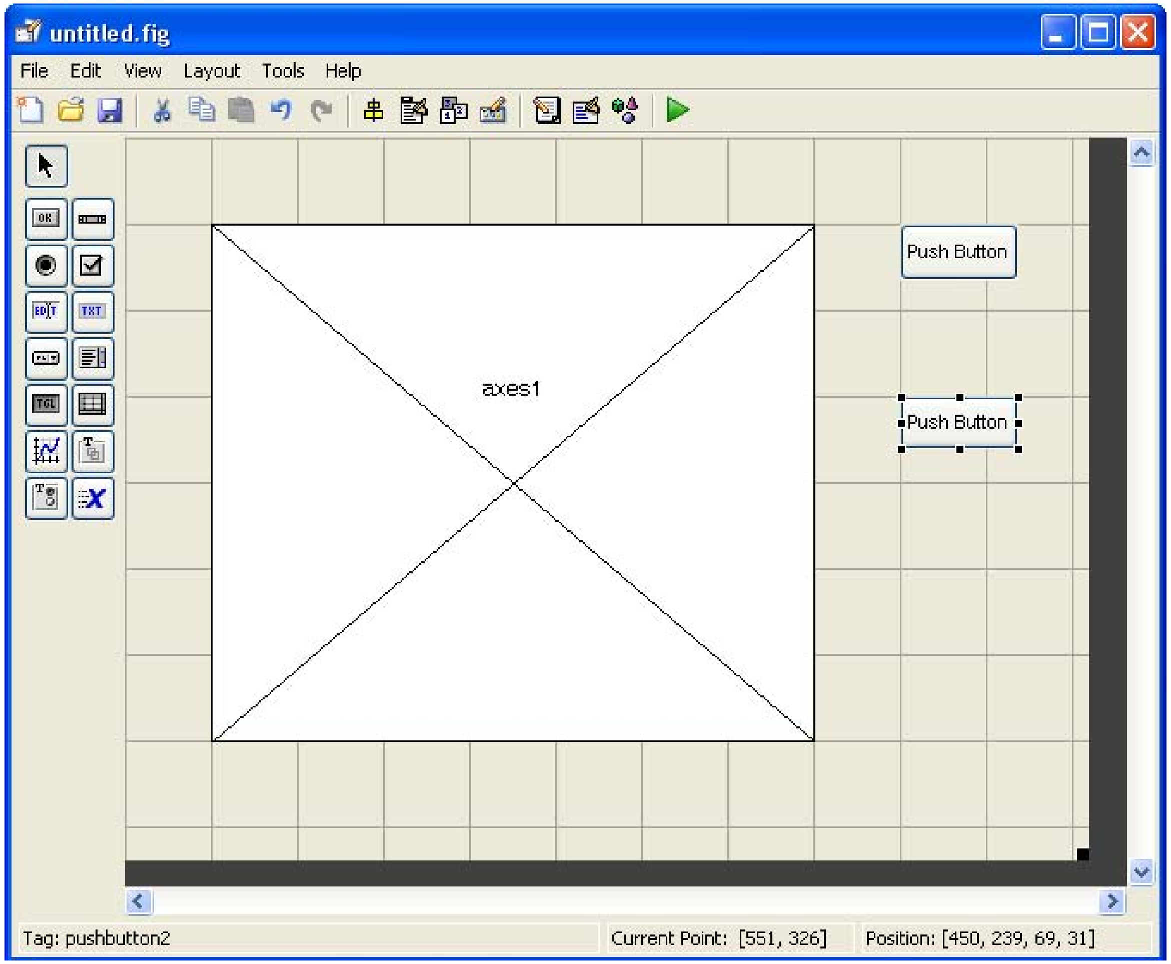
- It's really easy to make a graphical user interface in MATLAB
- To open the graphical user interface development environment, type **guide**
  - » **guide**
    - ➢ Select Blank GUI



Courtesy of The MathWorks, Inc. Used with permission.

Draw the GUI

- Select objects from the left, and draw them where you want them

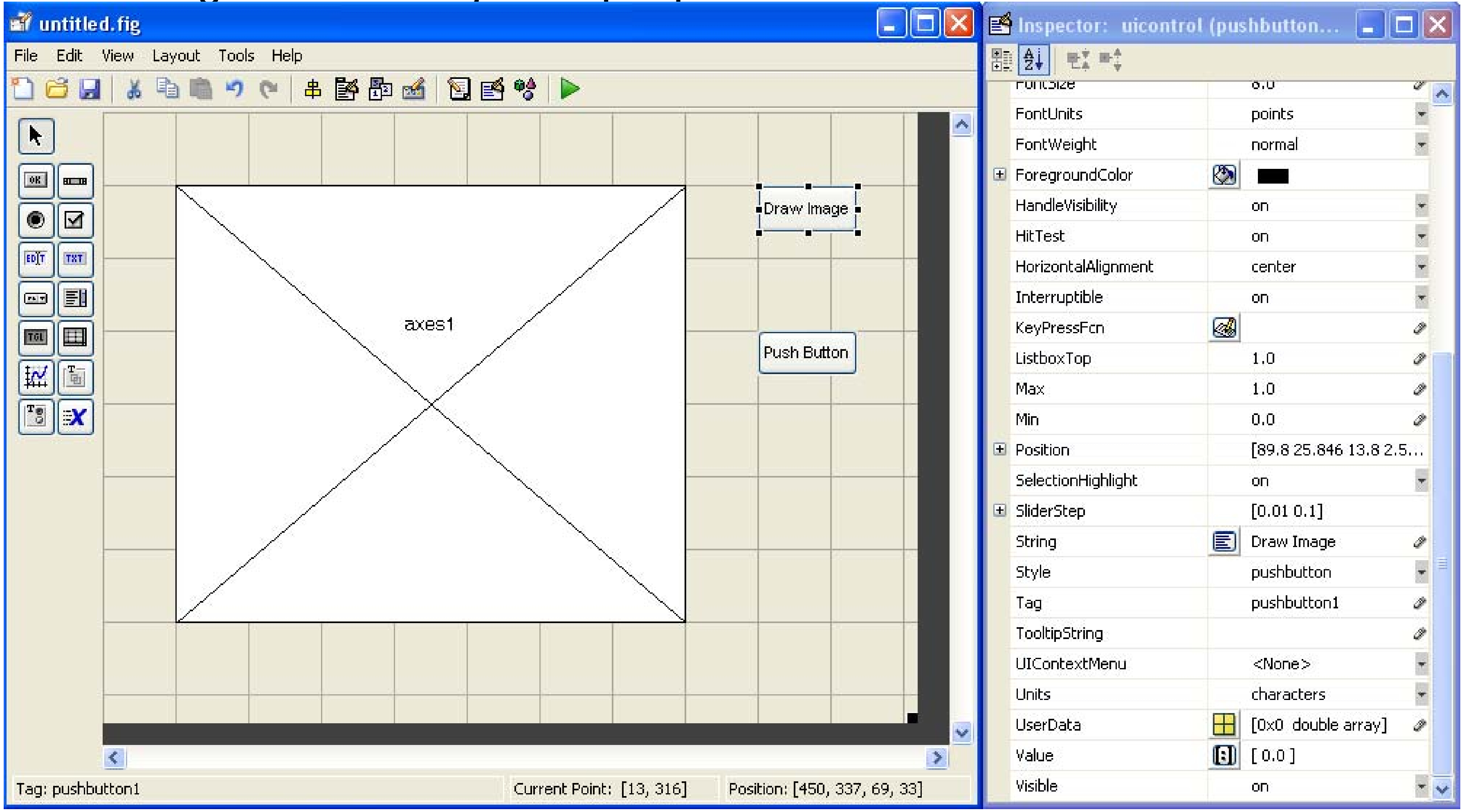


Tag: pushbutton2 Current Point: [551, 326] Position: [450, 239, 69, 31]

Courtesy of The MathWorks, Inc. Used with permission.

Change Object Settings

- Double-click on objects to open the Inspector. Here you can change all the object's properties.


untitled.fig
File Edit View Layout Tools Help
axes1
Draw Image
Push Button
Inspector: uicontrol (pushbutton...

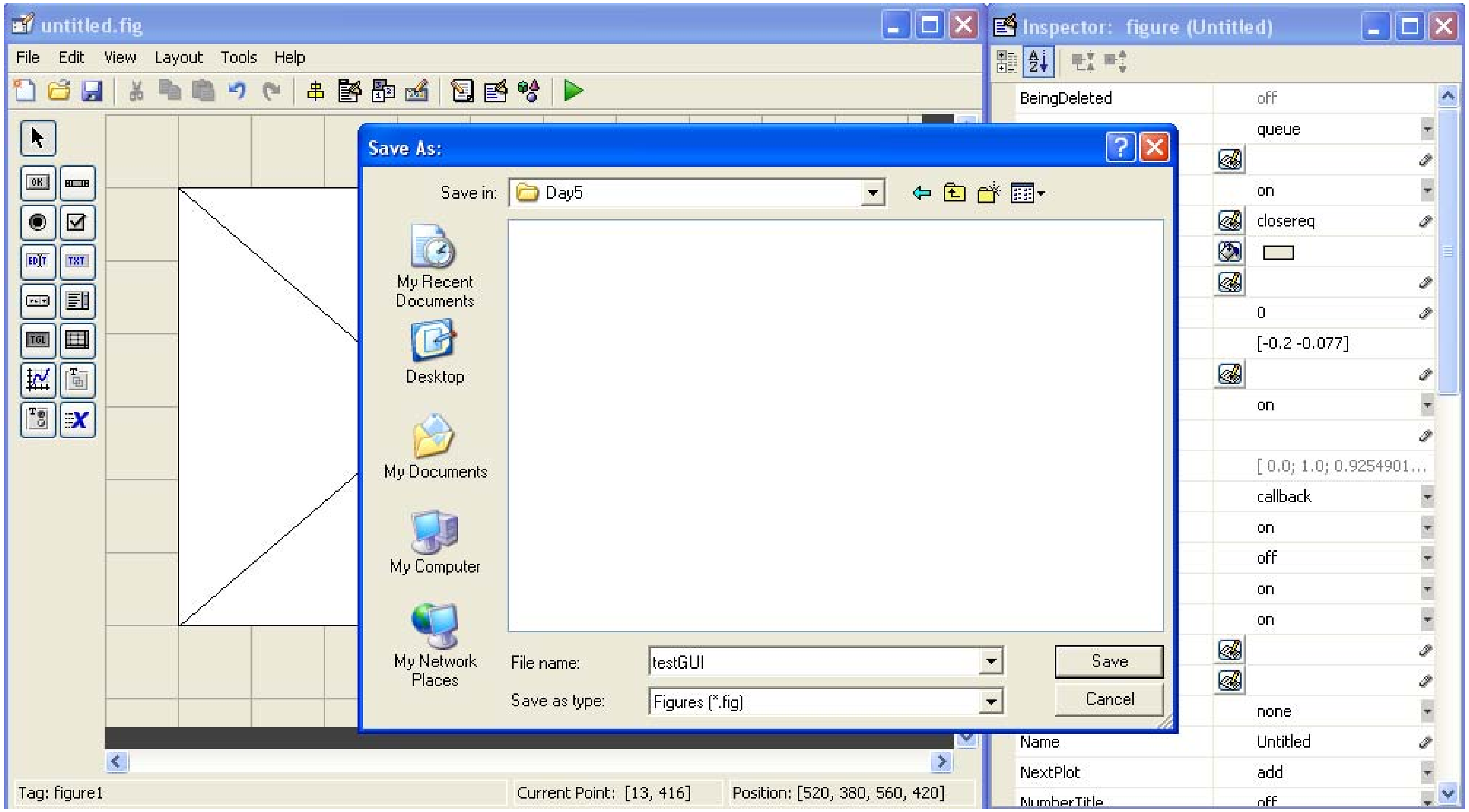
Tag: pushbutton1 | Current Point: [13, 316] | Position: [450, 337, 69, 33]

FontSize	8.0
FontUnits	points
FontWeight	normal
ForegroundColor	■
HandleVisibility	on
HitTest	on
HorizontalAlignment	center
Interruptible	on
KeyPressFcn	
ListboxTop	1.0
Max	1.0
Min	0.0
Position	[89.8 25.846 13.8 2.5...
SelectionHighlight	on
SliderStep	[0.01 0.1]
String	Draw Image
Style	pushbutton
Tag	pushbutton1
TooltipString	
UIContextMenu	<None>
Units	characters
UserData	[0x0 double array]
Value	[0.0]
Visible	on

Courtesy of The MathWorks, Inc. Used with permission.

Save the GUI

- When you have modified all the properties, you can save the GUI
- MATLAB saves the GUI as a .fig file, and generates an MATLAB file!



Save in: Day5

File name: testGUI

Save as type: Figures (*.fig)

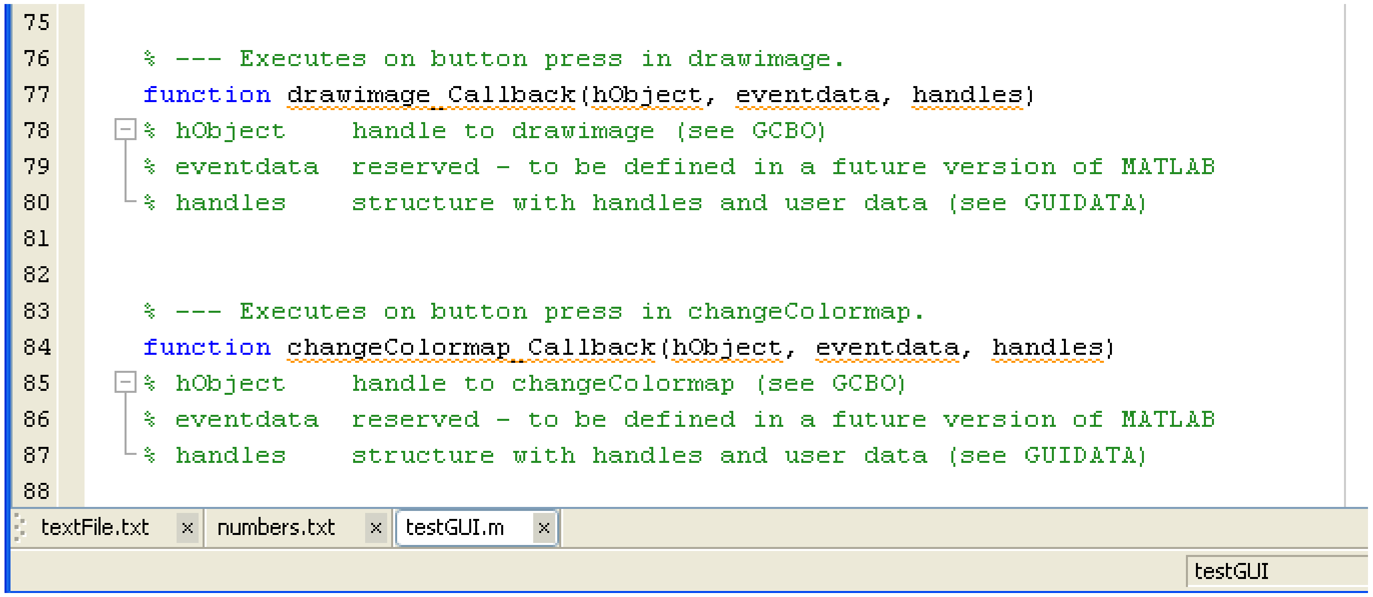
Tag: figure1 | Current Point: [13, 416] | Position: [520, 380, 560, 420]

BeingDeleted	off
	queue
	on
	closereq
	0
	[-0.2 -0.077]
	on
	[0.0; 1.0; 0.9254901...
	callback
	on
	off
	on
	on
	none
Name	Untitled
NextPlot	add
NumberTitle	off

Courtesy of The MathWorks, Inc. Used with permission.

Add Functionality to MATLAB file

- To add functionality to your buttons, add commands to the 'Callback' functions in the MATLAB file. For example, when the user clicks the Draw Image button, the **drawimage_Callback** function will be called and executed
- All the data for the GUI is stored in the handles, so use **set** and **get** to get data and change it if necessary
- Any time you change the handles, save it using **guidata**
  - » **guidata(handles.Figure1,handles);**

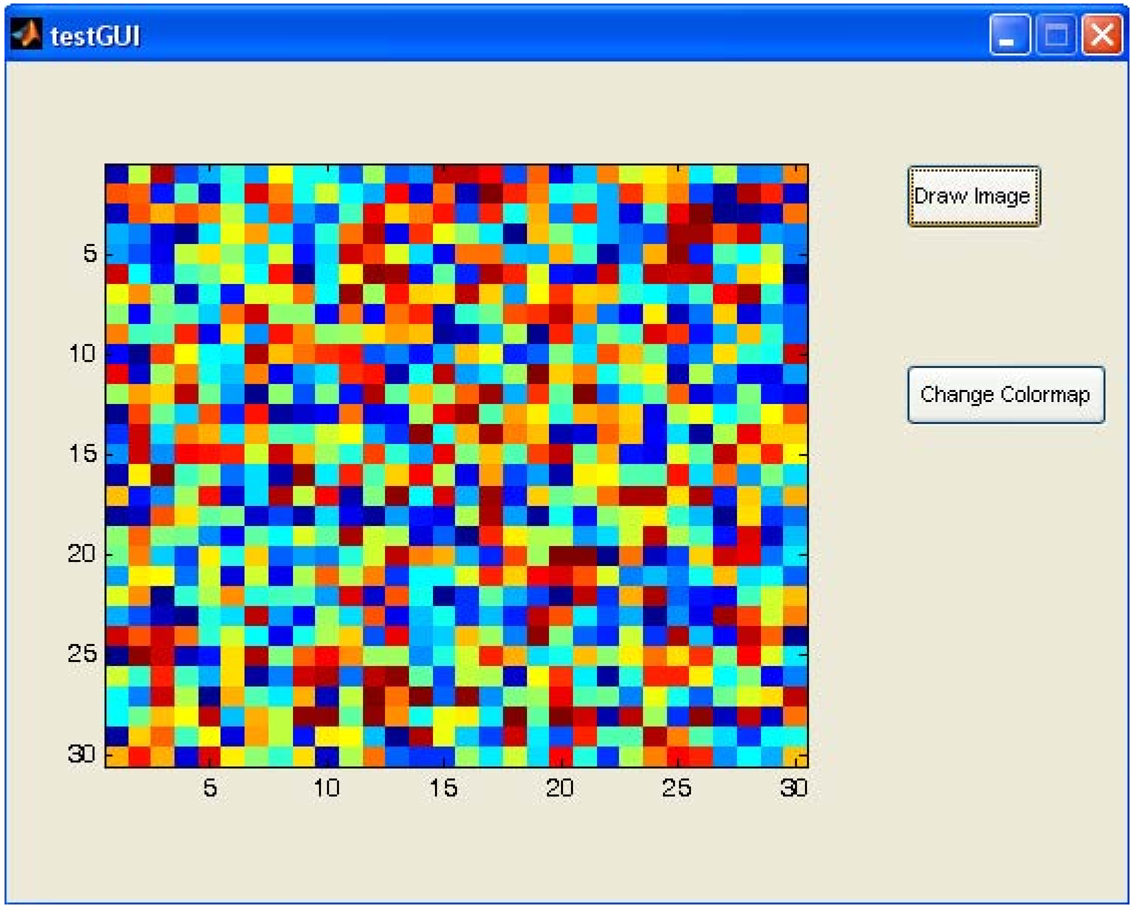


```
% --- Executes on button press in drawimage.
function drawimage_Callback(hObject, eventdata, handles)
% hObject    handle to drawimage (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)


% --- Executes on button press in changeColormap.
function changeColormap_Callback(hObject, eventdata, handles)
% hObject    handle to changeColormap (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Courtesy of The MathWorks, Inc. Used with permission.

Running the GUI

- To run the GUI, just type its name in the command window and the GUI will pop up. The debugger is really helpful for writing GUIs because it lets you see inside the GUI



Outline

(1) Data Structures
(2) Symbolic Math
(3) Simulink
(4) File I/O
(5) Graphical User Interfaces
(6) Online Resources

Central File Exchange

- The website – the MATLAB Central File Exchange!!
- Lots of people's code is there
- Tested and rated – use it to expand MATLAB's functionality

- http://www.mathworks.com/matlabcentral/

Outline

(1) Data Structures
(2) **Symbolic Math**
(3) **Simulink**
(4) **File I/O**
(5) **Graphical User Interfaces**
(6) **Online Resources**

Now you know EVERYTHING!