6.057
Introduction to MATLAB

Orhan Celiker, IAP 2019

Course Layout

Problem sets

- One per day, should take about 4 hours to complete
- Submit Word or PDF, include code and figures
- Some questions optional, but highly recommended!

Requirements for passing

- Attend 3/4 lectures (Friday is optional)
- Complete all problem sets (graded on a 3-level scale: -, √, +)...
- ... and achieve √ average

Prerequisites: You'll be fine!

MATLAB Basics

- MATLAB can be thought of as a super-powerful graphing calculator
  - Remember the TI-83 from calculus?
  - With many more buttons (built-in functions)

- In addition, it is a programming language
  - MATLAB is an interpreted language, like Python
  - Commands are executed line-by-line

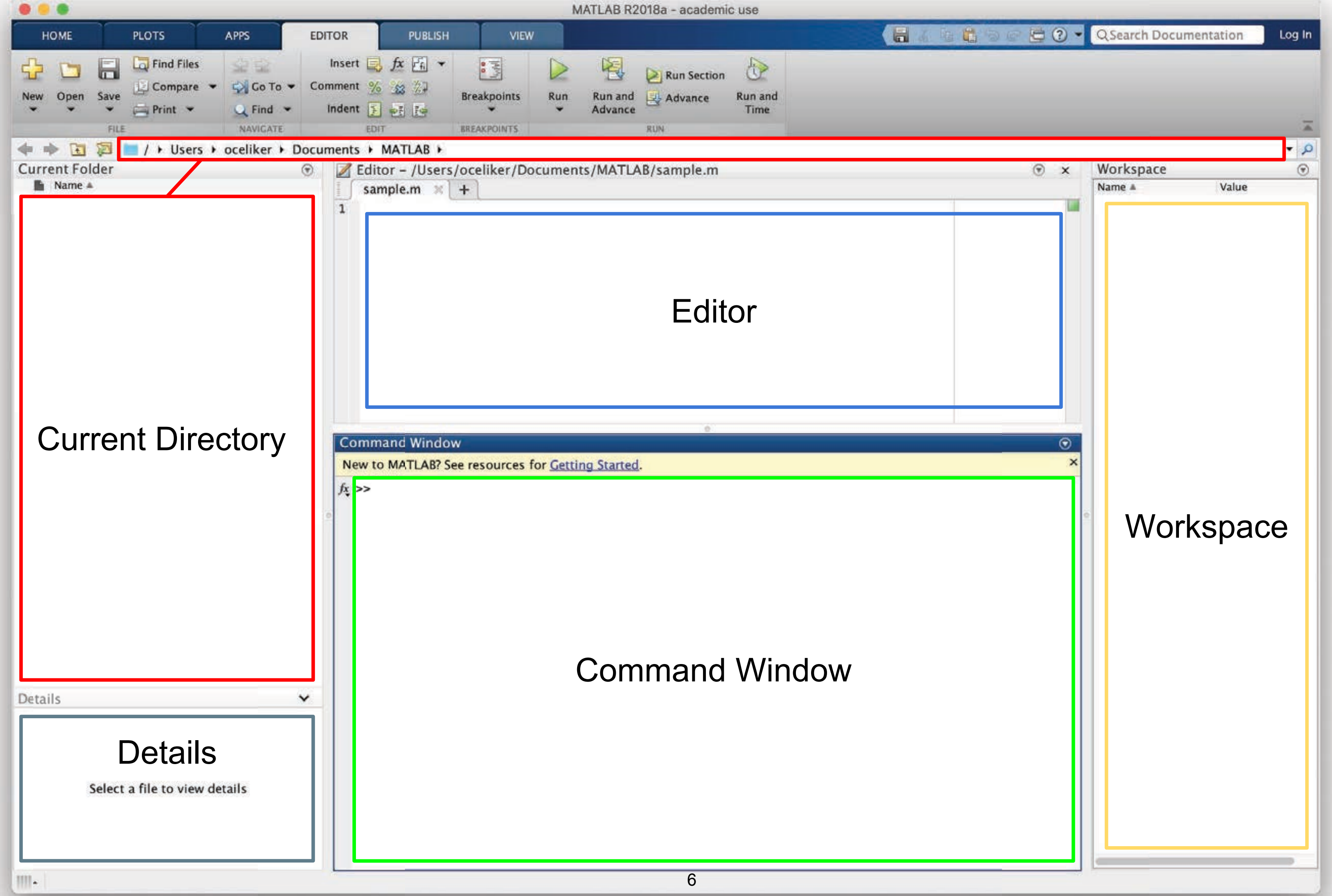
Outline

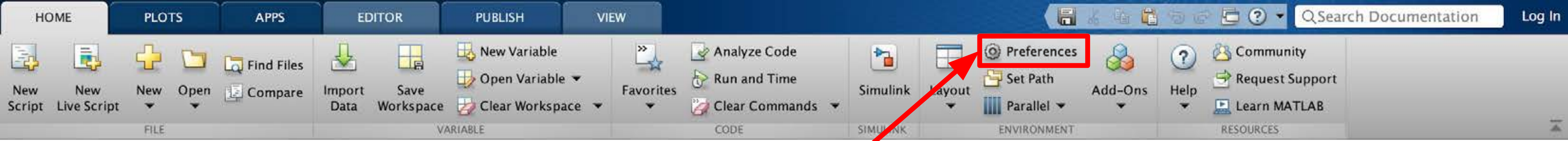
I. <u>Getting Started</u>
II. Scripts
III. Making Variables
IV. Manipulating Variables
V. Basic Plotting

Getting Started

- To get MATLAB Student Version for yourself

- You can also use MATLAB online
    - https://matlab.mathworks.com (requires Mathworks account with license)

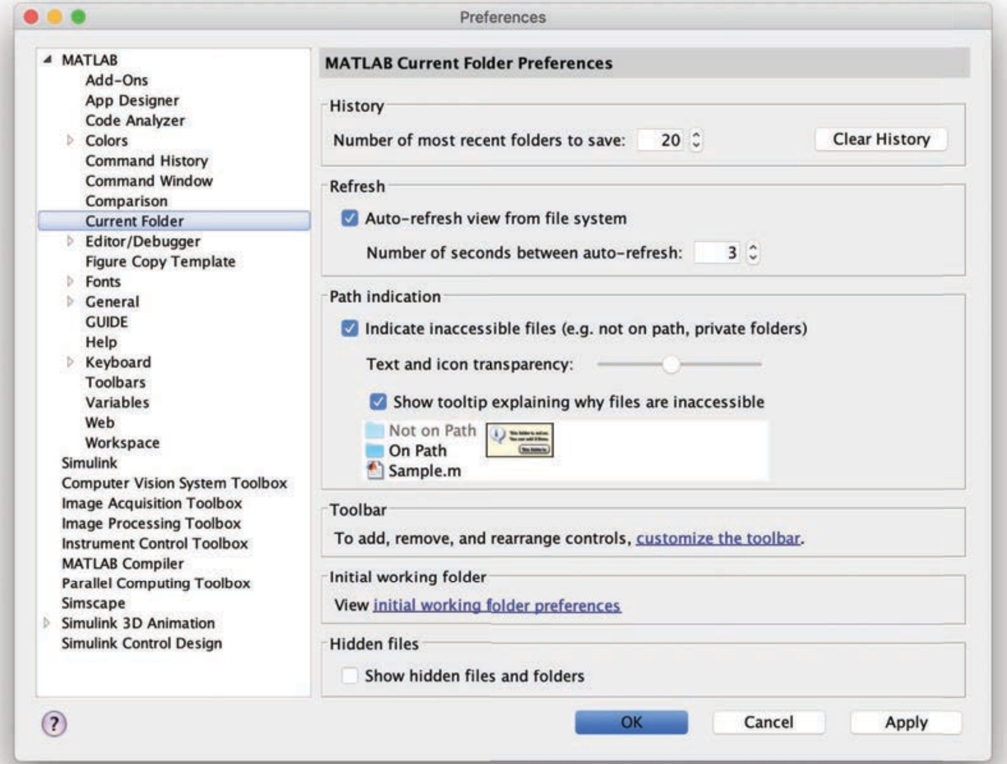

MATLAB R2018a - academic use
HOME
PLOTS
APPS
EDITOR
PUBLISH
VIEW
Search Documentation
Log In
New
Open
Save
Find Files
Compare
Print
FILE
Go To
Find
NAVIGATE
Insert
Comment
Indent
EDIT
Breakpoints
BREAKPOINTS
Run
Run and Advance
Run Section
Advance
Run and Time
RUN
/ Users oceliker Documents MATLAB
Current Folder
Name
Current Directory
Details
Details
Select a file to view details
Editor – /Users/oceliker/Documents/MATLAB/sample.m
sample.m
Editor
Command Window
New to MATLAB? See resources for Getting Started.
Command Window
Workspace
Name
Value
Workspace

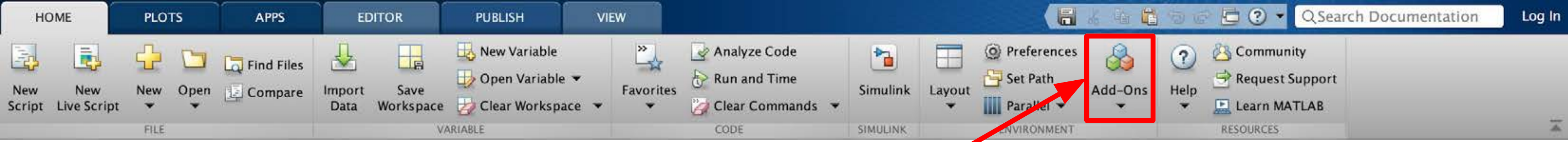


Customization

- In the top ribbon, navigate to:
  Home -> Environment -> Preferences

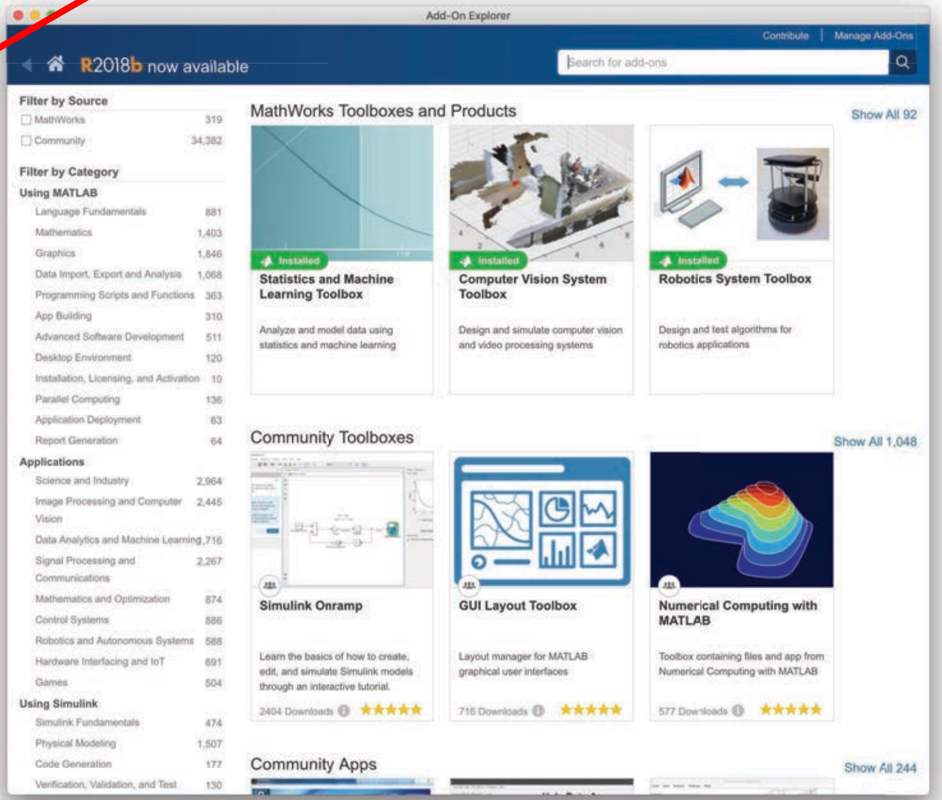
- Allows you to customize your MATLAB experience (colors, fonts, etc.)





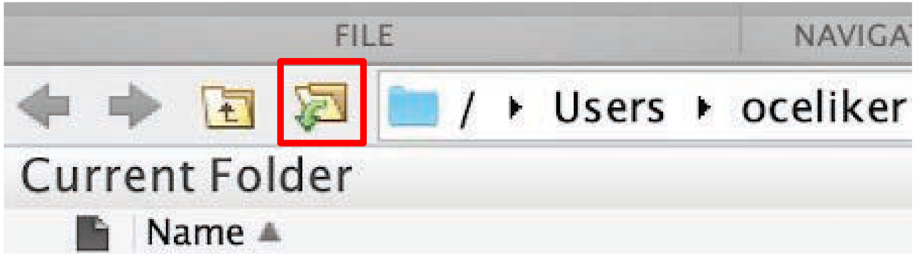
Installing Toolboxes

- In the top ribbon, navigate to:
  Home -> Environment -> Add-Ons
- Allows you to install toolboxes included with your license
- Recommended toolboxes:
  - Curve Fitting Toolbox
  - Computer Vision System Toolbox
  - Image Processing Toolbox
  - Optimization Toolbox
  - Signal Processing Toolbox
  - and anything related to your field!



Making Folders

- Use folders to keep your programs organized
- To make a new folder, click "Browse" next to the file path



- Click the Make New Folder button, and change the name of the folder. In the MATLAB folder (which should be open by default), make the following folder structure:

  MATLAB
  ↳ IAP MATLAB
  ↳ Day1

Help/Docs

- `help`
  - The most important command 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 documentation
- To get a nicer version of help with examples and easy-to-read description:
  - `doc sin`
- To search for a function by specifying keywords:
  - `docsearch sin trigonometric`

Outline

I. Getting Started
II. Scripts
III. Making Variables
IV. Manipulating Variables
V. Basic Plotting

Scripts: Overview

- Scripts are
  - Collection of commands executed in sequence
  - Written in the MATLAB editor
  - Saved as m-files (.m extension)
- To create an m-file from the command line:
  - `edit MyFileName.m`
  - or click the "New Script" button on the top left

Scripts: Some notes

- COMMENT!
  - Anything following a % sign is interpreted as a comment
  - The first contiguous comment becomes the script's help file
  - Comment thoroughly to avoid wasting time later!
  - Mark beginning of a code block by using %%
- Note that scripts are somewhat static, with no explicit input and output
- All variables created or modified in a script retain their values after script execution

Exercise: Scripts

- Make a script with the name **helloWorld.m**
- When run, the script should show the following text:

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

Hint: Use `disp(...)` to display strings. Strings are written between single quotes, e.g. `'This is a string'`

Outline

I. Getting Started
II. Scripts
III. <u>Making Variables</u>
IV. Manipulating Variables
V. Basic Plotting

Variable Types

- MATLAB is a "weakly typed" language
  - No need to initialize variables!
- MATLAB supports various types; the most popular ones are
  - 3.84
    - 64-bit double (default)
  - 'A'
    - 16-bit char
- Most variables you'll deal with are vectors, matrices, doubles or chars
- Other types are also supported: complex, symbolic, 16-bit and 8-bit integers (uint16 & uint8), etc.

Naming Variables

- To create a variable, simply assign a value to a name:

```
myNumberVariable = 3.14
myStringVariable = 'hello world!'
```

- Variable name rules
  - First character must be a LETTER
  - After that, any combination of numbers, letters and _
  - Names are CASE-SENSITIVE (e.g. `var1` is different than `Var1`)

Naming Variables (cont.)

Built-in variables (don't use these names for anything else!):

`i, j:` can be used to indicate complex numbers*

`pi:` has the value 3.1415...

`ans:` stores the result of the last unassigned value

`Inf, -Inf:` infinities

`NaN:` "Not a Number"

ops, use `ii, jj, kk`, etc. for loop counters.

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;`

Arrays

- Like other programming languages, arrays are an important part of MATLAB
- Two types of arrays:
  - Matrix of numbers (either double or complex)
  - 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 square brackets
    - `row = [ 1 2 3.2 4 6 5.4 ];`
    - `row = [ 1, 2, 4, 7, 4.3, 1.1 ];`
- Command window:

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

row =

    1.0000    2.0000    5.4000   -6.6000
```

- Workspace:

Workspace

Stack: Base

Name	Size	Bytes	Class
row	1x4	32	double array

Column vectors

- Column vector: semicolon-separated values between square brackets
  - `col = [ 1; 2; 3.2; 4; 6; 5.4 ];`

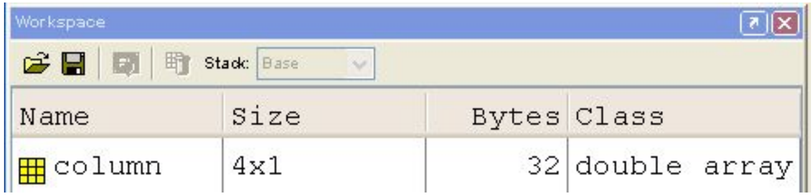
- Command window:

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

column =

     4
     2
     7
     4
```

- Workspace:



Size and length

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

```
>> size(row)

ans =

     1     4

>> length(row)

ans =

     4
```

```
>> size(column)

ans =

     4     1

>> length(column)

ans =

     4
```

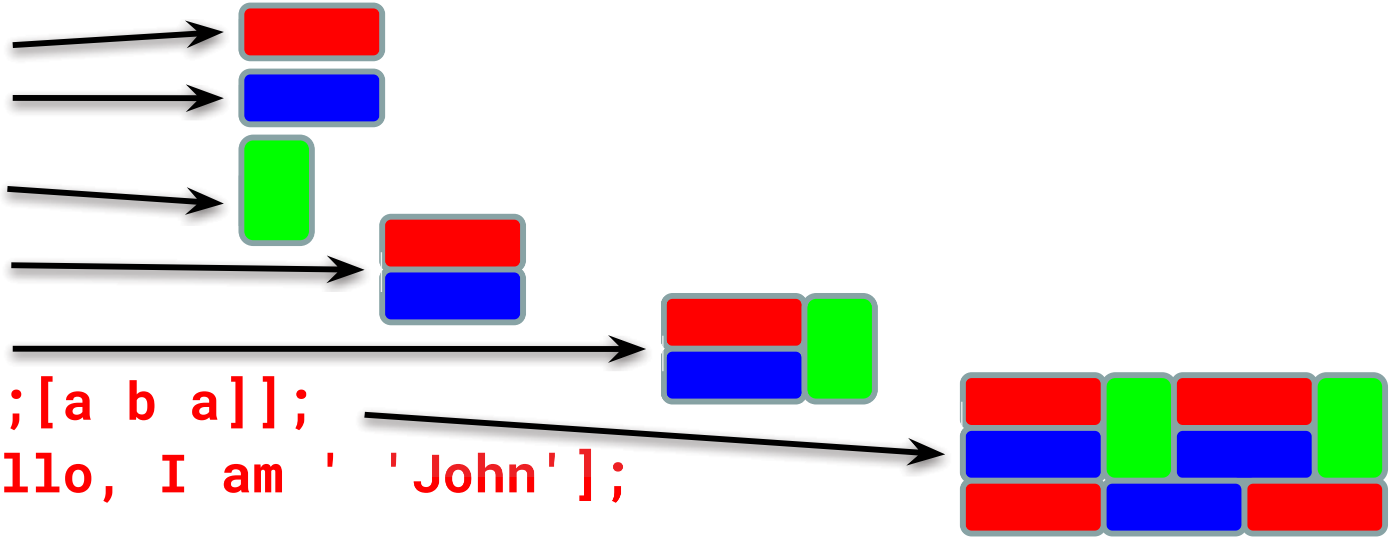
Matrices

- 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

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 in the current directory
  - Default working directory is MATLAB unless you navigate to another folder
  - Make sure you are in the correct folder. Right now we should be in \MATLAB\IAP MATLAB\Day 1
- Use clear to save variables to a file
  - `clear a b`
  - Look at workspace: variables a and b are gone
- Use load to load variables into the workspace
  - `load myFile`
  - Look at workspace: a and b are back

Exercise: Variables

Get and save the current date and time

- Create a variable `start` using the function `clock`
- What is the size of `start`? Is it a row or column?
- What does `start` contain? See `help clock`
- Convert the vector start to a string. Use the function `datestr` and name the new variable `startString`
- Save `start` and `startString` into a mat file named `startTime`

Exercise: Variables II

- In helloWorld.m, read in variables you saved using `load`
- Display the following text:

  I started learning MATLAB on [date, time]

- Hint: Use the `disp` command again
- Remember that strings are just vectors of characters, so you can join two strings by making a row vector with the two strings as sub-vectors.

Outline

I. Getting Started
II. Scripts
III. Making Variables
IV. <u>Manipulating Variables</u>
V. Basic Plotting

Basic Scalar Operations

- Arithmetic operations (+, -, *, /)
  - `7/45`
  - `(1+1i)*(1+2i)`
  - `1/0`
  - `0/0`
- Exponentiation
  - `4^2`
  - `(3+4*1j)^2`
- Complicated expressions: use parentheses
  - `((2+3)*3)^0.1`

Built-in Functions

- MATLAB has an <u>enormous</u> library of built-in functions
- Call using parentheses, passing parameters to function
  - `sqrt(2)`
  - `log(2), log10(0.23)`
  - `cos(1.2), atan(-.8)`
  - `exp(2+4*1i)`
  - `round(1.4), floor(3.3), ceil(4.23)`
  - `angle(1i); abs(1+1i);`

Exercise: Scalars

helloWorld script:

- Your learning time constant is 1.5 days. Calculate the number of seconds in 1.5 days and name this variable tau
- This class lasts 5 days. Calculate the number of seconds in 5 days and name this variable endOfClass
- This equation describes your knowledge as a function of time t:

$$k = 1 - e^{-t/\tau}$$

- How well will you know MATLAB at endOfClass? Name this variable knowledgeAtEnd (use exp)
- Using the value of knowledgeAtEnd, display the phrase:

```
At the end of 6.057, I will know X% of MATLAB
```

Hint: to convert a number to a string, use num2str

Transpose

- The transpose operator 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
  - Transposes and conjugates all complex numbers
- For vectors of real numbers .' and ' give same result
  - For transposing a vector, always use .' to be safe

Addition and Subtraction

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

$$\begin{array}{r} [12 \quad 3 \quad 32 \quad -11] \\ +[2 \quad 11 \quad -30 \quad 32] \\ \hline =[14 \quad 14 \quad 2 \quad 21] \end{array}$$

$$\begin{bmatrix} 12 \\ 1 \\ -10 \\ 0 \end{bmatrix} - \begin{bmatrix} 3 \\ -1 \\ 13 \\ 33 \end{bmatrix} = \begin{bmatrix} 9 \\ 2 \\ -23 \\ -33 \end{bmatrix}$$

Addition and Subtraction

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

Element-wise functions

- 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
```

Operators

- Multiplication can be done in a standard way or element-wise
- Standard multiplication (*) is matrix 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: for now, 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

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

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

Exercise: Vector Operations

Calculate how many seconds elapsed since start of class

- In helloWorld.m, make variables called secPerMin, secPerHour, secPerDay, secPerMonth (assume 30.5 days per month), and secPerYear (12 months in year), which have the number of seconds in each time period
- Assemble a row vector called secondConversion that has elements in this order: secPerYear, secPerMonth, secPerDay, secPerHour, secPerMin, 1
- Make a currentTime vector by using clock
- Compute elapsedTime by subtracting currentTime from start
- Compute t (the elapsed time in seconds) by taking the dot product of secondConversion and elapsedTime (transpose one of them to get the dimensions right)

Exercise: Vector Operations

Display the current state of your knowledge

- Calculate currentKnowledge using the same relationship as before, and the t we just calculated:

$$k = 1 - e^{-t/\tau}$$

- Display the following text:
  At this time, I know X% of MATLAB

Automatic Initialization

- Initialize a vector of **ones**, **zeros**, or **random** 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 (representing uninitialized variables)

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 is not specified, default is 1
- To initialize logarithmically spaced values use **logspace**
    - ➢ Similar to **linspace**, but see **help**

Exercise: Vector Functions

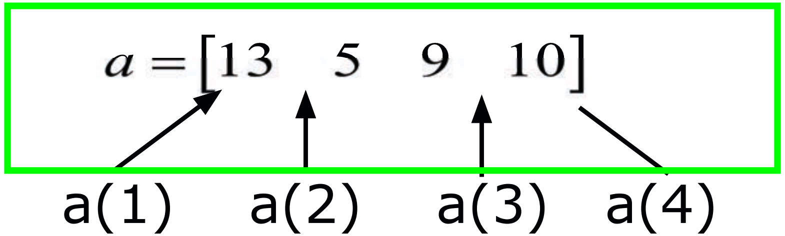
Calculate your learning trajectory

- In helloWorld.m, make a linear time vector `tVec` that has 10,000 samples between 0 and `endOfClass`
- Calculate the value of your knowledge (call it `knowledgeVec`) at each of these time points using the same equation as before:

$$k = 1 - e^{-t/\tau}$$

Vector Indexing

- <u>MATLAB indexing starts with **1**, not **0**</u>
  - ➢ We will not respond to any emails where this is the problem.
- a(n) returns the nth element

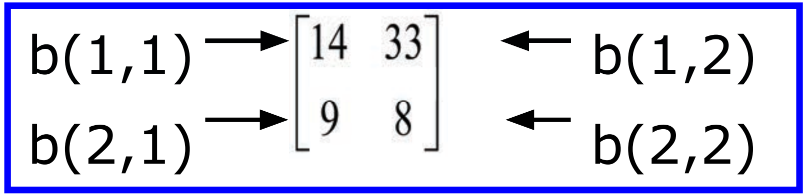


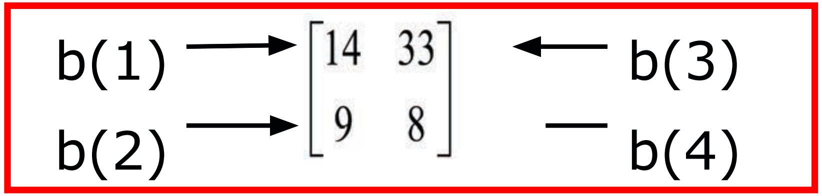
- The index argument can be a vector. In this case, each element is looked up individually, and returned as a vector of the same size as the index vector.

```
» x=[12 13 5 8];
```

Matrix Indexing

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





- Picking submatrices


```
» A = rand(5) % shorthand for 5x5 matrix
```

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
  - » `vec = [5 3 1 9 7]`
- To get the minimum value and its index (similar for `max`):
  - » `[minVal,minInd] = min(vec);`
- To find the indices of specific values or ranges
  - » `ind = find(vec == 9); vec(ind) = 8;`
  - » `ind = find(vec > 2 & vec < 6);`
    - ➢ **find** expressions can be very complex, more on this later
    - ➢ When possible, **logical indexing** is faster than **find**!
    - ➢ E.g., `vec(vec == 9) = 8;`

Exercise: Indexing

When will you know 50% of MATLAB?

- First, find the index where **knowledgeVec** is closest to 0.5. Mathematically, what you want is the index where the value of ~ $\left|knowledgeVec - 0.5\right|$ is at a minimum (use **abs** and **min**)
- Next, use that index to look up the corresponding time in **tVec** and name this time **halfTime**
- Finally, display the string: Convert **halfTime** to days by using secPerDay. I will know half of MATLAB after X days

Outline

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

Did everyone sign in?

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

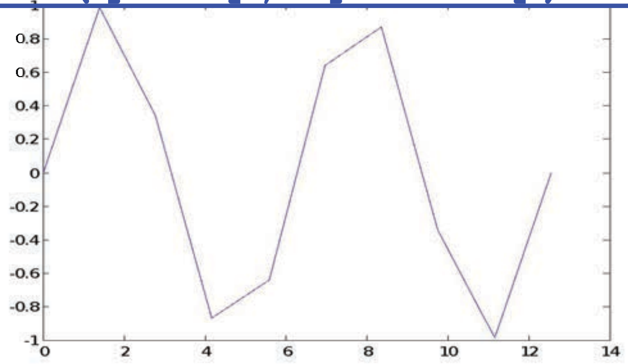
```
» plot(x,y);
```

MATLAB makes visualizing data fun and easy!

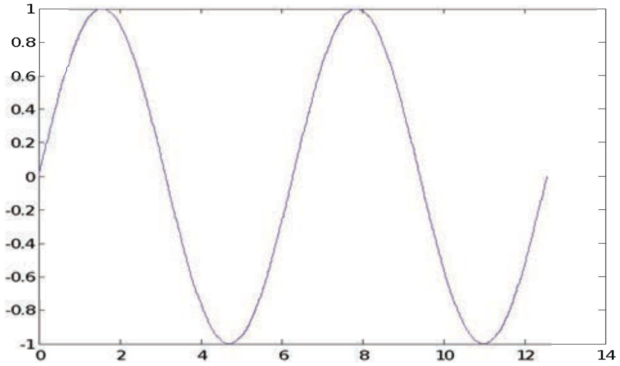
What does plot do?

- **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])`

10 x values:



1000 x values:



Exercise: Plotting

Plot the learning trajectory

- In helloWorld.m, open a new figure (use `figure`)
- Plot knowledge trajectory using `tVec` and `knowledgeVec`
- When plotting, convert `tVec` to days by using `secPerDay`
- Zoom in on the plot to verify that `halfTime` was calculated correctly

End of Lecture 1

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

Hope that wasn't too much and you enjoyed it!!

6.057
Introduction to programming in MATLAB

Lecture 2: Visualization and Programming

Orhan Celiker

IAP 2019

Homework 1 Recap

Some things that came up:

- Plotting a straight line
  - » `x = 1:10`
  - » `plot(x, 0)`
    - ➢ Not an error, but probably not what you meant

- Use of semicolon – never required if one command per line. You can also put multiple commands on one line; in this case, a semicolon is necessary to separate commands:
  - » `x=1:10; y=(x-5).^2; z = x.*y;`

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

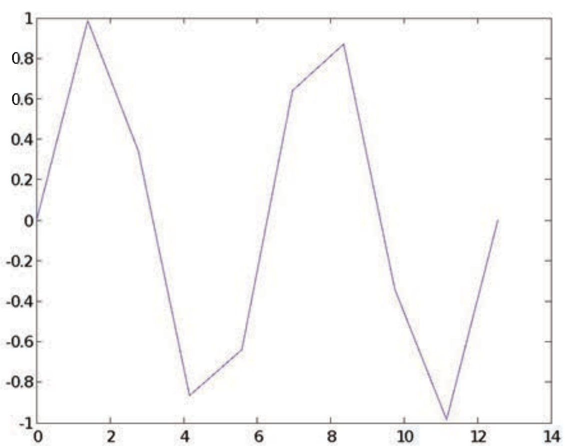
```
» plot(x,y);
```

MATLAB makes visualizing data fun and easy!

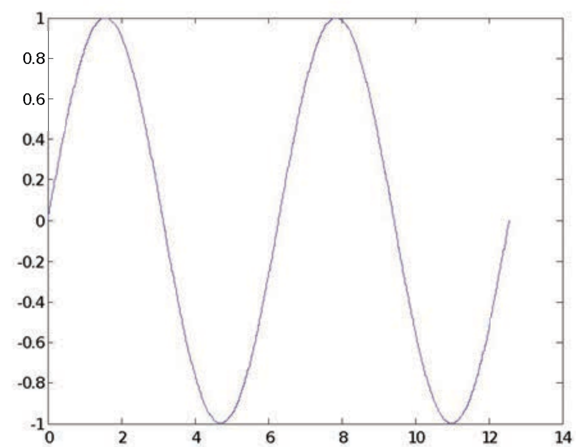
What does plot do?

- **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:



Exercise: Plotting

Plot the learning trajectory

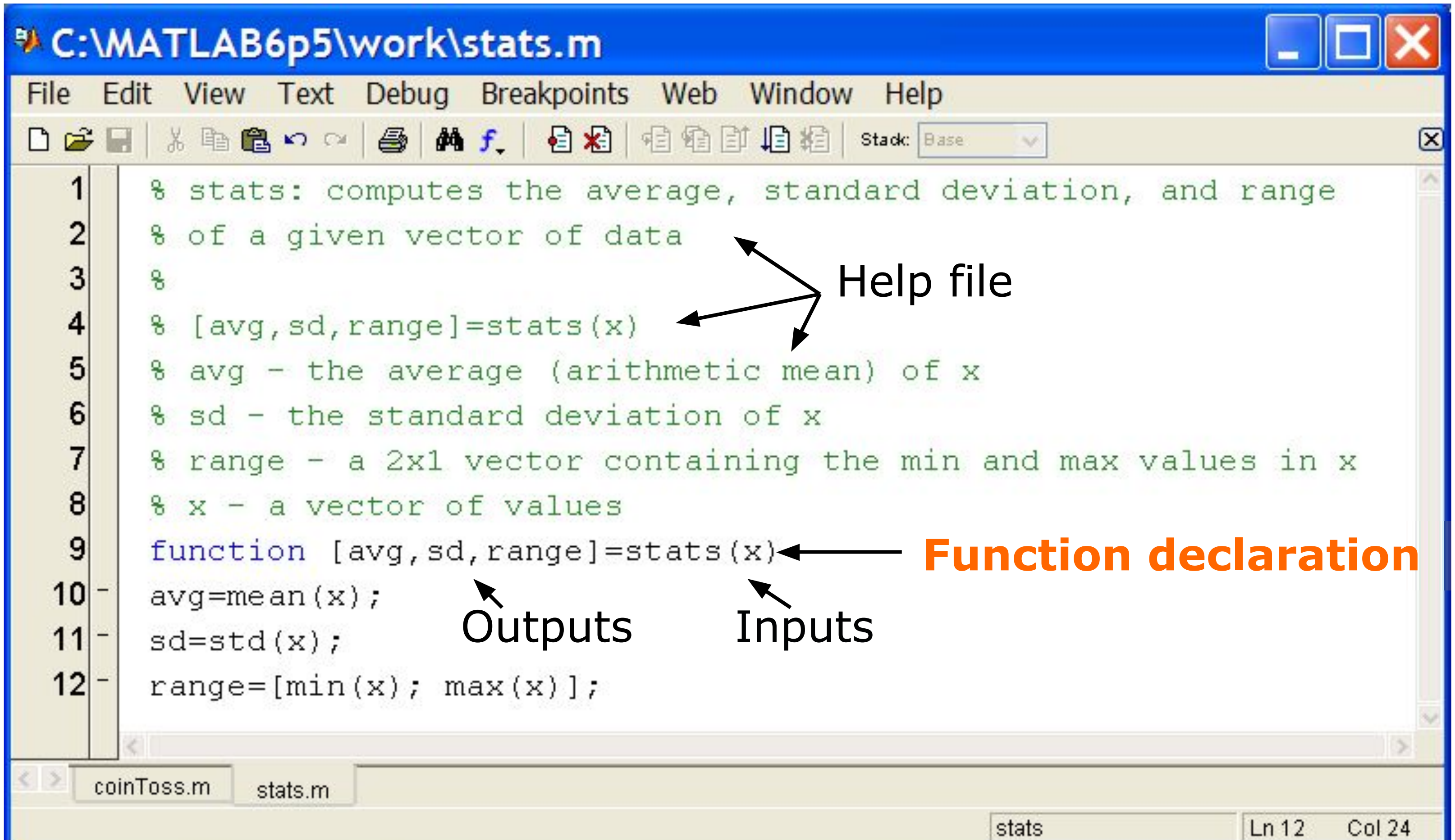
- In helloWorld.m, open a new figure (use `figure`)
- Plot knowledge trajectory using `tVec` and `knowledgeVec`
- When plotting, convert `tVec` to days by using `secPerDay`
- Zoom in on the plot to verify that `halfTime` was calculated correctly

Outline for Lec 2

(1) **Functions**
(2) Flow Control
(3) Line Plots
(4) Image/Surface Plots
(5) Efficient Codes
(6) Debugging

User-defined Functions

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



User-defined Functions

- Some comments about the function declaration

Inputs

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

Must have the reserved word: function

Function name should match m-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 (though, you can use **`return`** to break and go back to invoking function)
- Variable scope: Any variable created within the function but not returned disappears after the function stops running (They're called "local variables")

Functions: overloading

- We're familiar with
  - » `zeros`
  - » `size`
  - » `length`
  - » `sum`

- Look at the help file for size by typing
  - » `help size`

- The help file describes several ways to invoke the function
  - ➢ D = SIZE(X)
  - ➢ [M,N] = SIZE(X)
  - ➢ [M1,M2,M3,...,MN] = SIZE(X)
  - ➢ M = SIZE(X,DIM)

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)
```

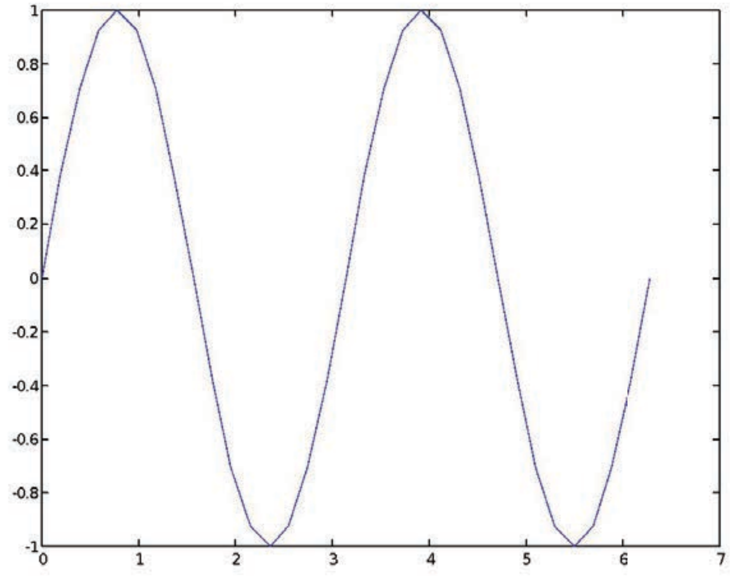
- You can overload your own functions by having variable number of input and output arguments (see `varargin`, `nargin`, `varargout`, `nargout`)

Functions: Exercise

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

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

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



Outline

(1) Functions

(2) Flow Control

(3) Line Plots

(4) Image/Surface Plots

(5) Efficient Codes

(6) Debugging

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

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
- Lots of `elseif`'s? consider using `switch`

for

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

Loop variable

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

Command block

- 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**

while

- The while is like a more general for loop:
  - No 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! CTRL+C?!
- You can use `break` to exit a loop

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 is stored in the built-in variable `nargin`

Outline

(1) **Functions**
(2) **Flow Control**
(3) **Line Plots**
(4) **Image/Surface Plots**
(5) **Efficient Codes**
(6) **Debugging**

Plot Options

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

```
» plot(x,y,'k.-');
```

color → k, marker → ., line-style → -

- 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 line styles

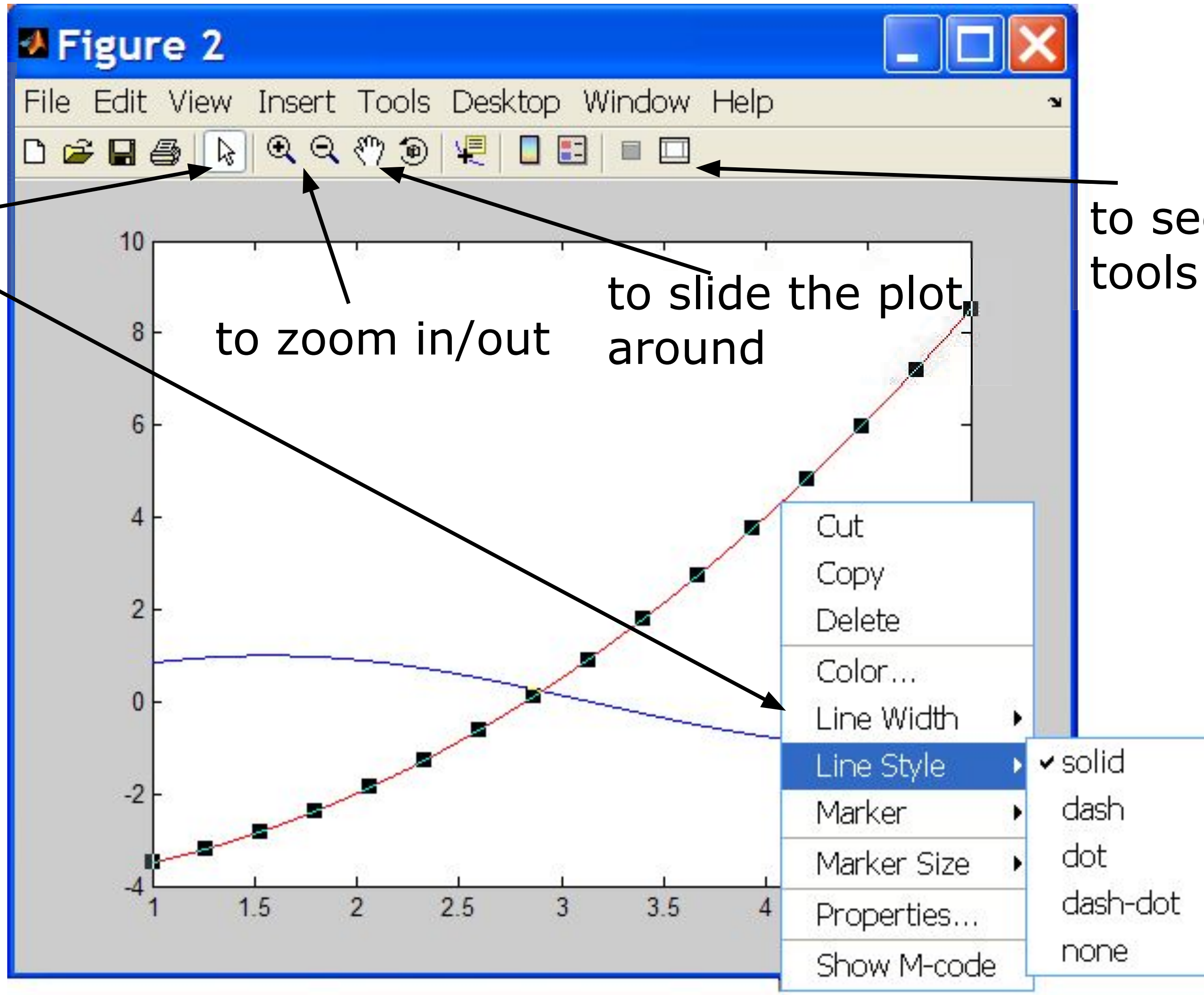
Playing with the Plot

to select lines and delete or change properties

to zoom in/out

to slide the plot around

to see all plot tools at once

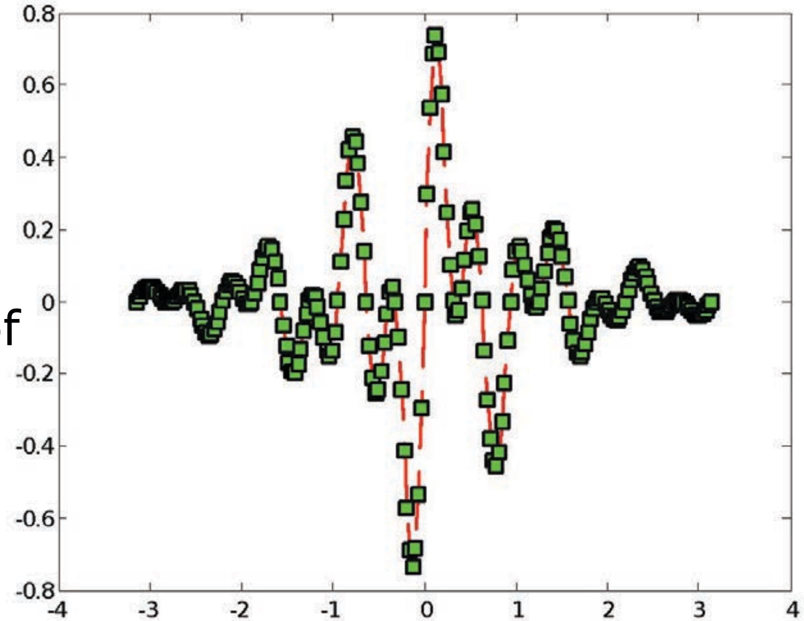


Line and Marker Options

- 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

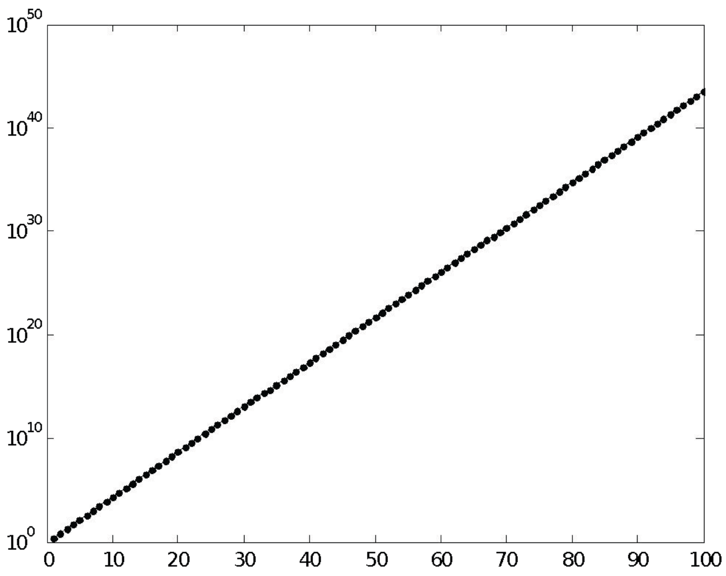
```
» x=-pi:pi/100:pi;
» y=cos(4*x).*sin(10*x).*exp(-abs(x));
» plot(x,y,'k-');
```

- 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

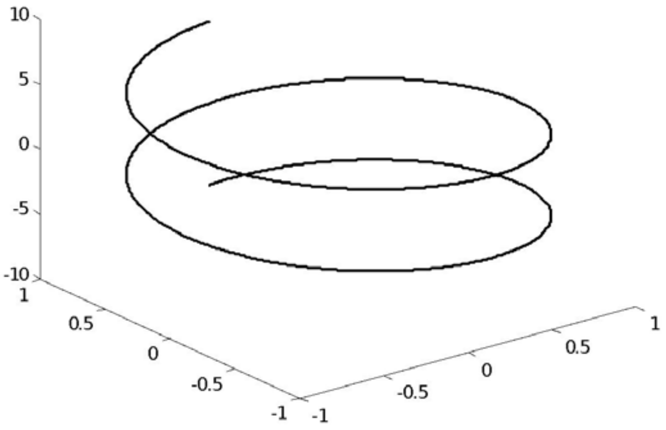
- We can plot in 3 dimensions just as easily as in 2D

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

3D Line Plots

- We can plot in 3 dimensions just as easily as in 2D
  - » `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 (see `doc axis` for more modes)
  - » `axis square`
    - ➢ makes the current axis look like a square box
  - » `axis tight`
    - ➢ fits axes to data
  - » `axis equal`
    - ➢ makes x and y scales the same
  - » `axis xy`
    - ➢ puts the origin in the lower left corner (default for plots)
  - » `axis ij`
    - ➢ puts the origin in the upper left corner (default for matrices/images)

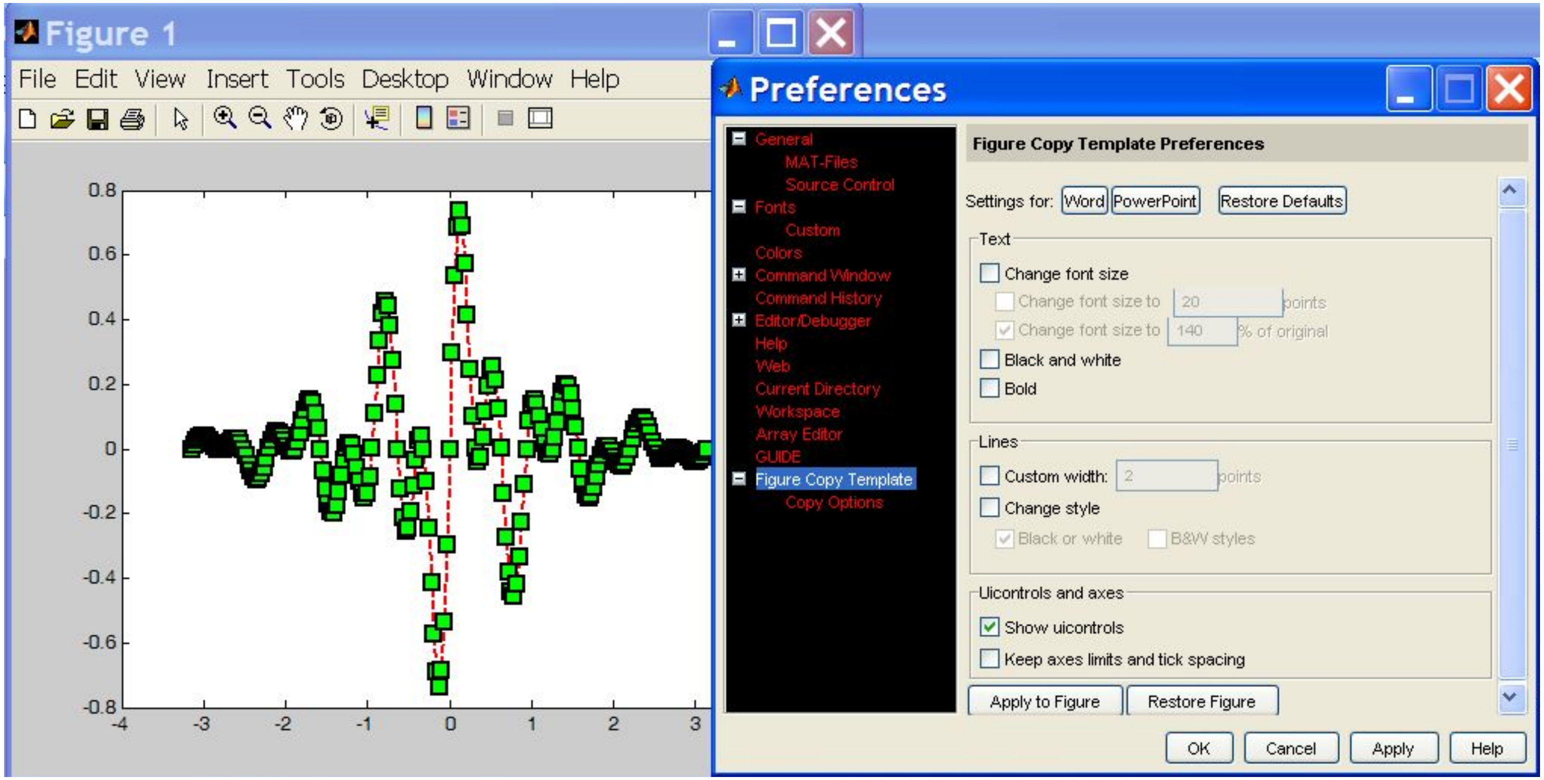
Multiple Plots in one Figure

- To have multiple axes in one figure
  - » `subplot(2,3,1)`
    - ➢ makes a figure with 2 rows and 3 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)`
    - ➢ activates a range of axes and fuses them into one

- To close existing figures
  - » `close([1 3])`
    - ➢ closes figures 1 and 3
  - » `close all`
    - ➢ closes all figures (useful in scripts)

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



Saving Figures

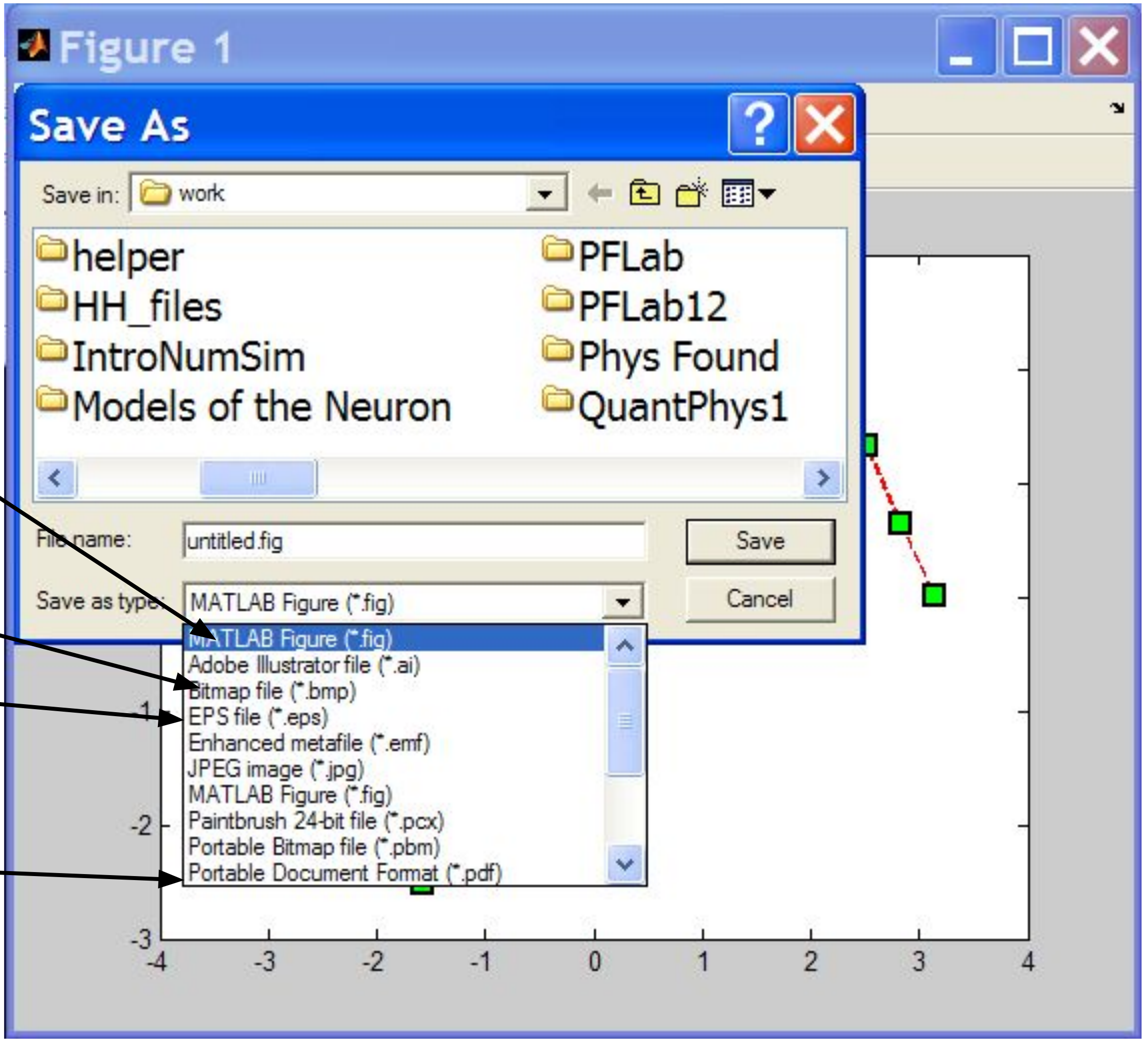
- Figures can be saved in many formats. The common ones are:

.fig preserves all information

.bmp uncompressed image

.eps high-quality scaleable format

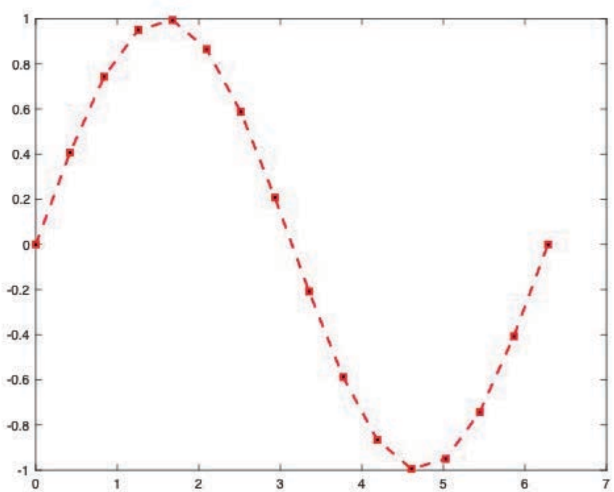
.pdf compressed image



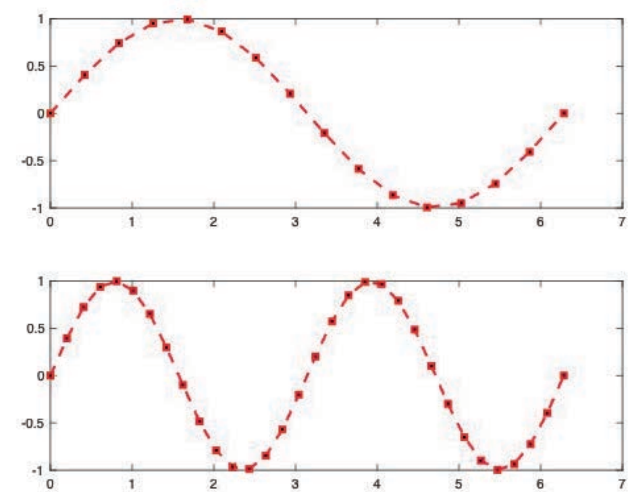
Advanced Plotting: Exercise

- 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 plot both frequencies (`subplot`)

`plotSin(6)`



`plotSin(1,2)`

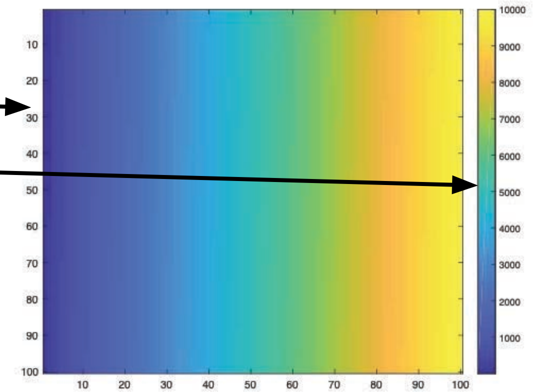


Outline

(1) Functions
(2) Flow Control
(3) Line Plots
(4) Image/Surface Plots
(5) Efficient Codes
(6) Debugging

Visualizing matrices

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

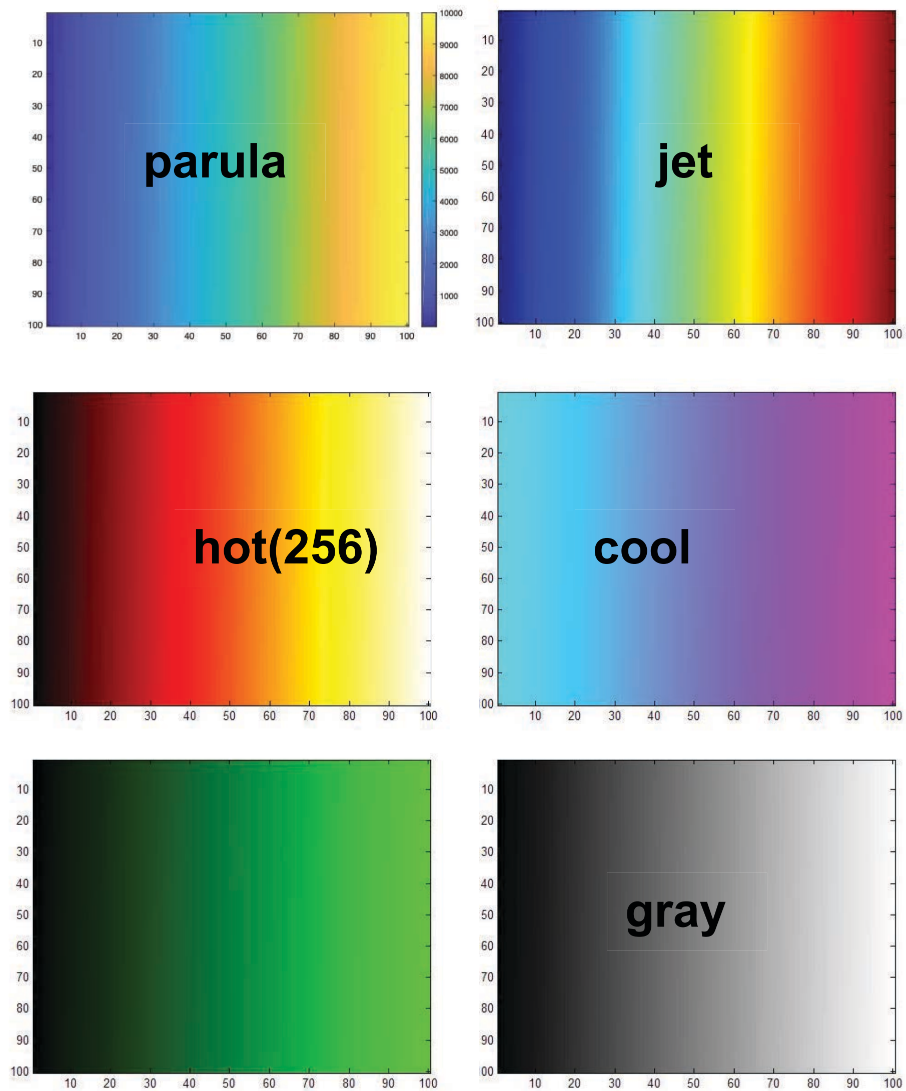


- **imagesc** automatically scales the values to span the entire colormap

- 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 parula
  - » `colormap(gray)`
  - » `colormap(cool)`
  - » `colormap(hot(256))`
- See `help hot` for a list
- Can define custom color-map
  - » `map=zeros(256,3);`
  - » `map(:,2)=(0:255)/255;`
  - » `colormap(map);`

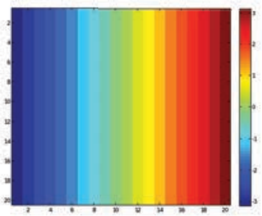


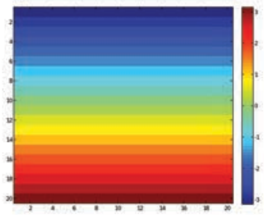
Surface Plots

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

$$f(x,y) = sin(x)cos(y)$$
$$x \in [-\pi,\pi]; y \in [-\pi,\pi]$$

- **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
  - ➢ built-in function: **meshgrid**





surf

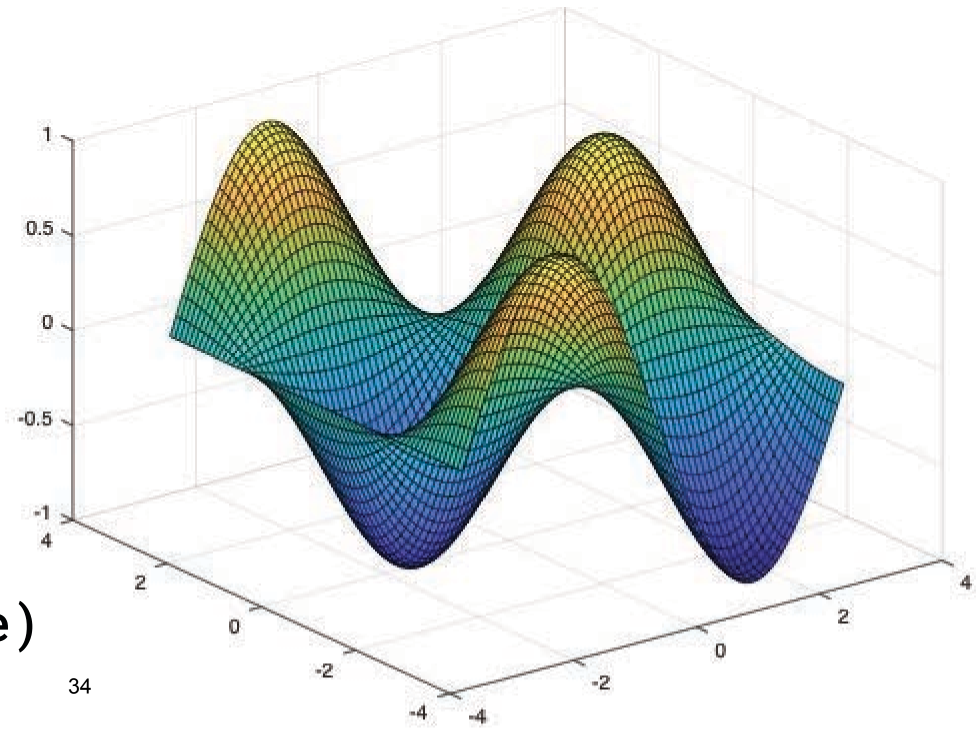
- Make the x and y vectors
  » `x=-pi:0.1:pi;`
  » `y=-pi:0.1:pi;`

- Use meshgrid to make matrices
  » `[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);`

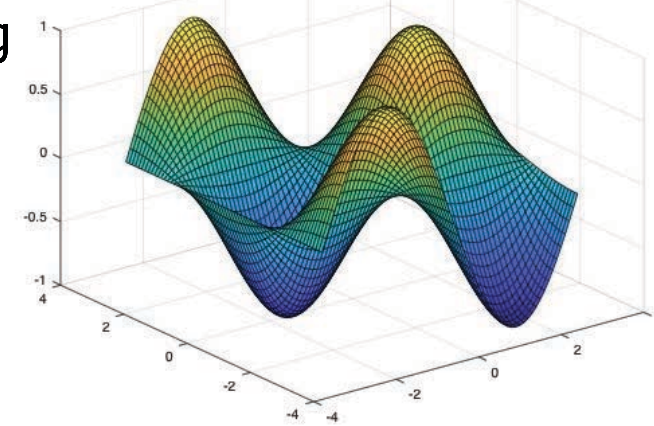
*Try typing **`surf(membrane)`**

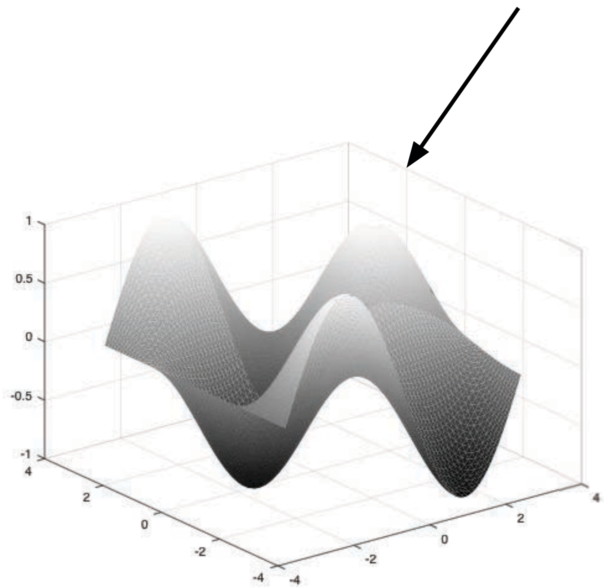


surf Options

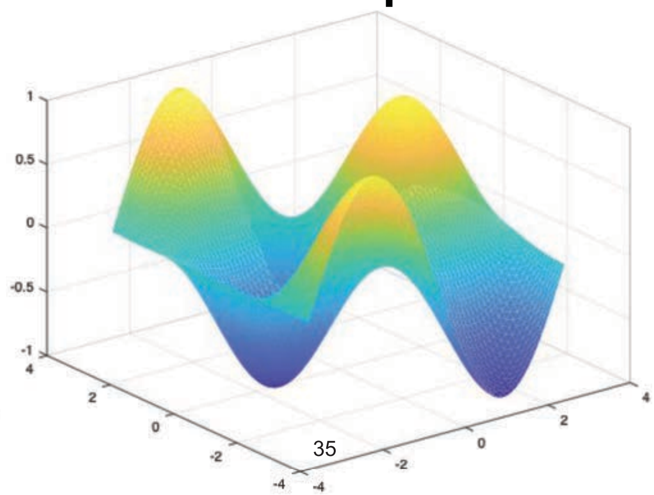
- See **help surf** for more options
- There are three types of surface shading
  - » `shading faceted`
  - » `shading flat`
  - » `shading interp`
- You can also change the colormap
  - » `colormap(gray)`

faceted

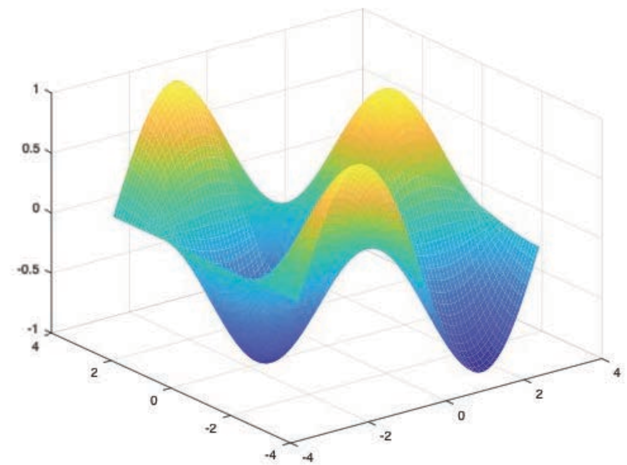




interp

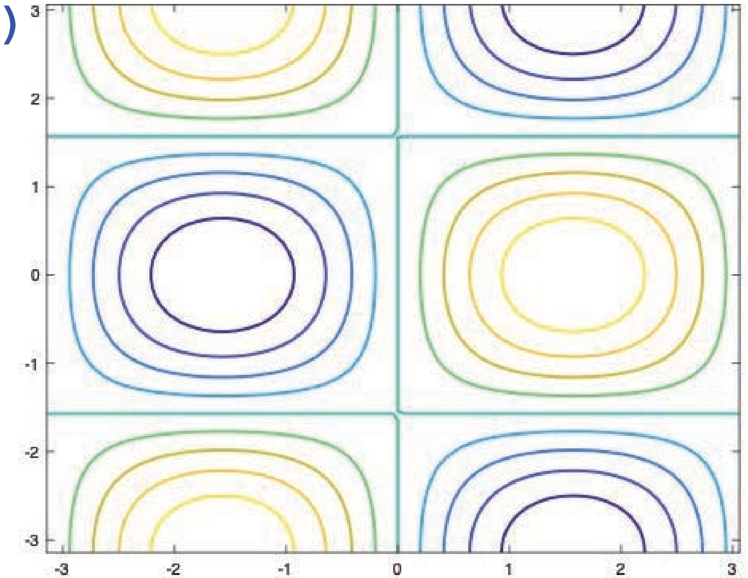


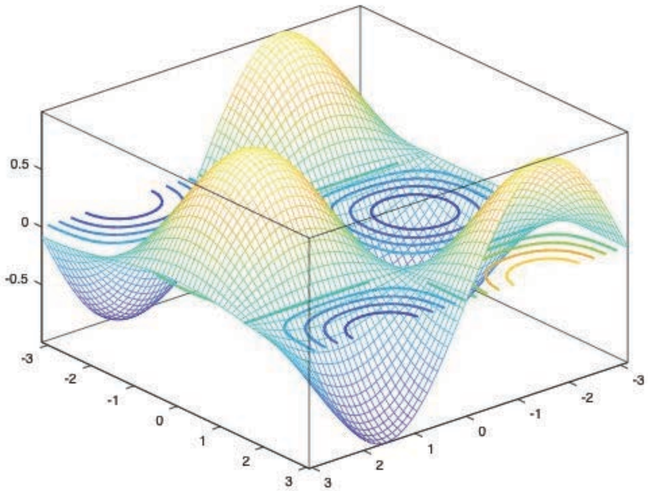
flat



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

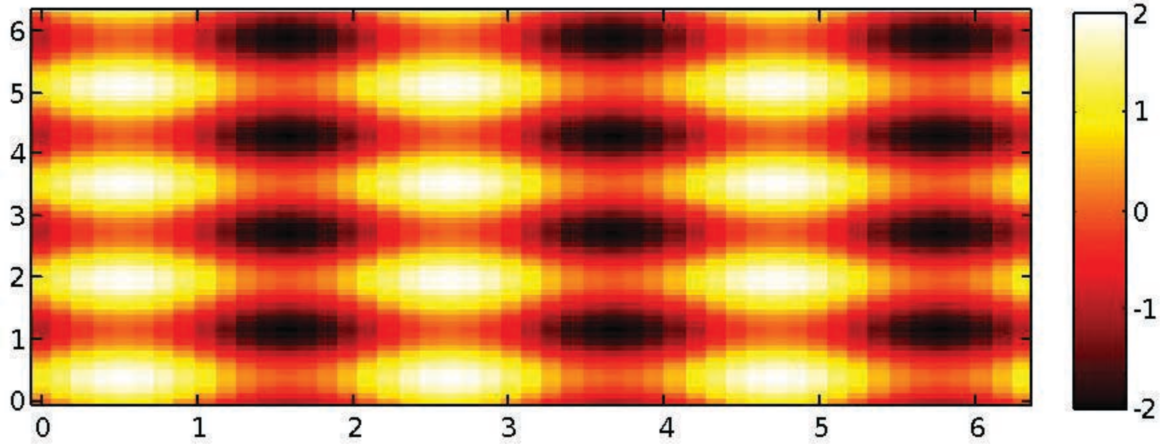
- Modify `plotSin` to do the following:
- If two inputs are given, evaluate the following function:

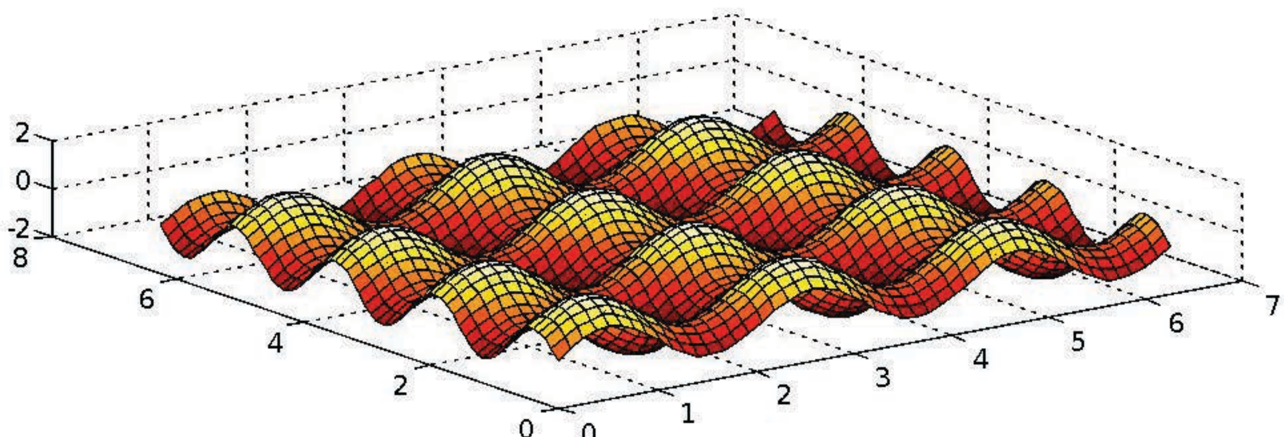
$$Z = \sin(f_1 x) + \sin(f_2 y)$$

- 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

`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

Outline

(1) Functions
(2) Flow Control
(3) Line Plots
(4) Image/Surface Plots
(5) **Efficient codes**
(6) Debugging

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` contains 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

& combines the two vectors using logical **and** operator

find returns the indices of the 1's

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));
```

Is there a better way?!

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

Vectorization

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

Vectorization

- Avoid loops
  - ➢ This is referred to as vectorization
- Vectorized code is more efficient for MATLAB
- Use indexing and matrix operations to avoid loops
- For instance, to add 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

Vectorization

- Avoid loops
  - ➢ This is referred to as vectorization
- Vectorized code is more efficient for MATLAB
- Use indexing and matrix operations to avoid loops
- For instance, to add 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`

Preallocation

- Avoid variables growing inside a loop
- Re-allocation of memory is time consuming
- Preallocate the required memory by initializing the array to a default value
- For example:

```
» for n=1:100
»     res = % Very complex calculation %
»     a(n) = res;
» end
```

  ➢ Variable `a` needs to be resized at every loop iteration

Preallocation

- Avoid variables growing inside a loop
- Re-allocation of memory is time consuming
- Preallocate the required memory by initializing the array to a default value
- For example:

```
» a = zeros(1, 100);
» for n=1:100
»     res = % Very complex calculation %
»     a(n) = res;
» end
```

  - ➢ Variable `a` is only assigned new values. No new memory is allocated

Outline

(1) Functions
(2) Flow Control
(3) Line Plots
(4) Image/Surface Plots
(5) Efficient codes
(6) **Debugging**

Display

- When debugging functions, use **disp** to print messages
  - » `disp('starting loop')`
  - » `disp('loop is over')`
    - ➢ `disp` prints the given string to the command window

- It's also helpful to show variable values
  - » `disp(['loop iteration ' num2str(n)]);`
    - ➢ Sometimes it's easier to just remove some semicolons

Debugging

- To use the debugger, set breakpoints
  - ➢ Click on – next to line numbers in m-files
  - ➢ Each red dot that appears is a breakpoint
  - ➢ Run the program
  - ➢ The program pauses when it reaches a breakpoint
  - ➢ Use the command window to probe variables
  - ➢ Use the debugging buttons to control debugger

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
» Mystery1;
» a=toc;
» Mystery2;
» b=toc;
```

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

Performance Measures

- Example: Sparse matrices

```
» A=zeros(10000); A(1,3)=10; A(21,5)=pi;
» B=sparse(A);
» inv(A);   % what happens?
» inv(B);   % what about now?
```

- If system is sparse, can lead to large memory/time savings

```
» A=zeros(1000); A(1,3)=10; A(21,5)=pi;
» B=sparse(A);
» C=rand(1000,1);
» tic; A\C; toc;   % slow!
» tic; B\C; toc;   % much faster!
```

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	

End of Lecture 2

(1) Functions
(2) Flow Control
(3) Line Plots
(4) Image/Surface Plots
(5) Efficient codes
(6) Debugging

Vectorization makes coding fun!

6.057
Introduction to MATLAB

Lecture 3 : Solving Equations, Curve Fitting, and Numerical Techniques

Orhan Celiker

IAP 2019

Outline

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

Systems of Linear Equations

MATLAB makes linear algebra fun!

- 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.

Worked Example: 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;
» x=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

```
» x=A\b;
```

  - gives least squares solution

```
» error=abs(A*x1-b)
```

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; matrix invertible if det nonzero
- 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 (mostly) the same as x=inv(A)*b
- Get the condition number
  - » `c=cond(mat);` (or its reciprocal: `c = rcond(mat);`)
    - ➢ if condition number is large, when solving A*x=b, small errors in b can lead to large errors in x (optimal c==1)

Matrix Decompositions

- MATLAB has many built-in matrix decomposition methods

- The most common ones are
  - » `[V,D]=eig(X)`
    - ➢ Eigenvalue decomposition
  - » `[U,S,V]=svd(X)`
    - ➢ Singular value decomposition
  - » `[Q,R]=qr(X)`
    - ➢ QR decomposition
  - » `[L,U]=lu(X)`
    - ➢ LU decomposition
  - » `R=chol(X)`
    - ➢ Cholesky decomposition (R must be positive definite)

Exercise: Fitting Polynomials

- Find the best second-order polynomial that fits the points: (-1,0), (0,-1), (2,3).

$$a(-1)^2 + b(-1) + c = 0$$
$$a(0)^2 + b(0) + c = -1$$
$$a(2)^2 + b(2) + c = 3$$

Outline

(1) Linear Algebra

(2) Polynomials

(3) Optimization

(4) Differentiation/Integration

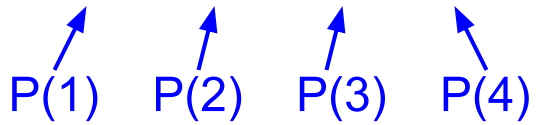
(5) Differential Equations

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

$$ax^3+bx^2+cx+d$$



- 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 (least-squares sense) 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

Outline

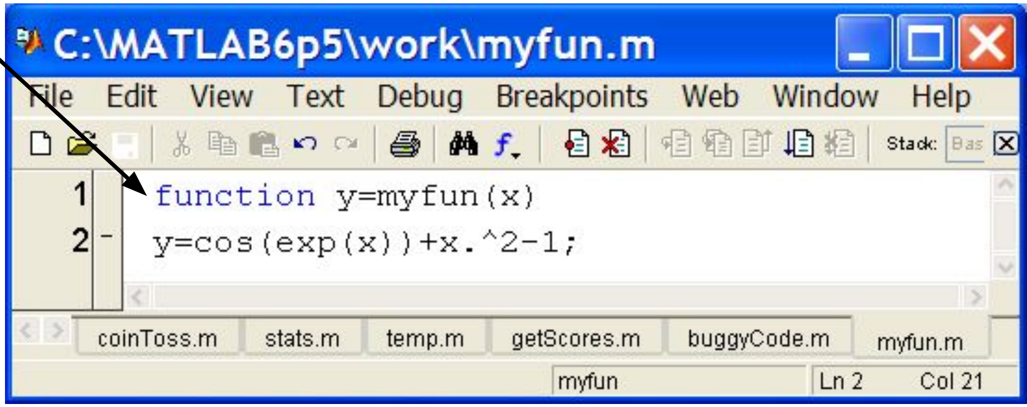
(1) Linear Algebra
(2) Polynomials
(3) Optimization
(4) Differentiation/Integration
(5) Differential Equations

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



Minimizing a Function

- **fminbnd**: minimizing a function over a bounded interval
  - » `x=fminbnd('myfun',-1,2);`
    - ➢ myfun takes a scalar input and returns a scalar output
    - ➢ myfun(x) will be the minimum of myfun for $-1 \leq x \leq 2$
- **fminsearch**: unconstrained interval
  - » `x=fminsearch('myfun',.5)`
    - ➢ finds the local minimum of myfun starting at x=0.5
- Maximize g(x) by minimizing f(x)=-g(x)
- Solutions may be local!

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 );
```

input — function to evaluate

```
» x=fminbnd(@(x) (cos(exp(x))+x.^2-1),-1,2);
```

- Can also store the function handle

```
» func=@(x) (cos(exp(x))+x.^2-1);
» func(1:10);
```

Optimization Toolbox

- If you are familiar with optimization methods, use the optimization toolbox
- Useful for larger, more structured optimization problems
- Sample functions (see **`help`** for more info)
  - » **`linprog`**
    - ➢ linear programming using interior point methods
  - » **`quadprog`**
    - ➢ quadratic programming solver
  - » **`fmincon`**
    - ➢ constrained nonlinear optimization

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.

Digression: Numerical Issues

- Many techniques in this lecture use floating point numbers
- **This is an approximation!**

- Examples:
  - » `sin(pi) = ?`
  - » `sin(2 * pi) = ?`
  - » `sin(10e16 * pi) = ?`
    - ➢ Both sin and pi are approximations!

  - » `A = (10e13)*ones(10) + rand(10)`
    - ➢ A is nearly singular, poorly conditioned (see `cond(A)`)
  - » `inv(A)*A = ?`

A Word of Caution

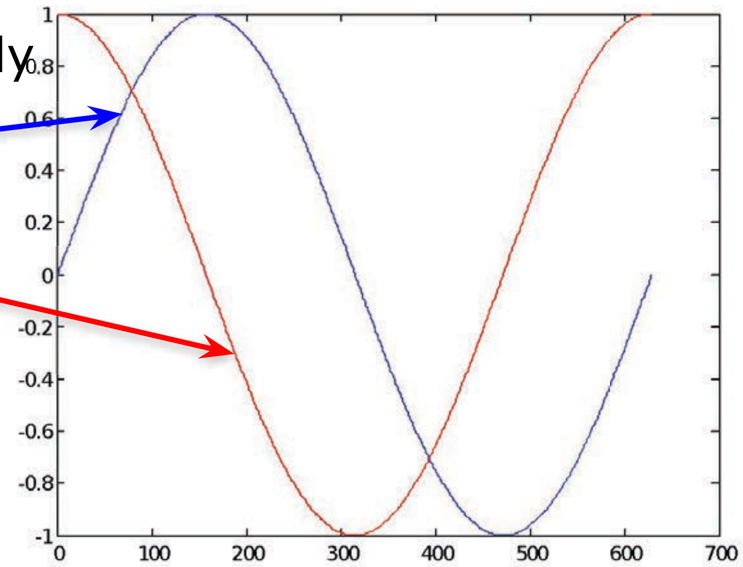
- MATLAB knows no fear!
- Give it a function, it optimizes / differentiates / integrates
  - ➢ That's great! It's so powerful!
- Numerical techniques are powerful **but** not magic
- **Beware of overtrusting the solution!**
  - ➢ You will get an answer, but it may not be what you want
- Analytical forms may give more intuition
  - ➢ Symbolic Math Toolbox

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 2^{nd} dimension, dm=[2 2;4 -2]
  - ➢ The opposite of **`diff`** is the cumulative sum **`cumsum`**

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

- Higher derivatives / complicated problems: Fit spline (see **help**)

Numerical Integration

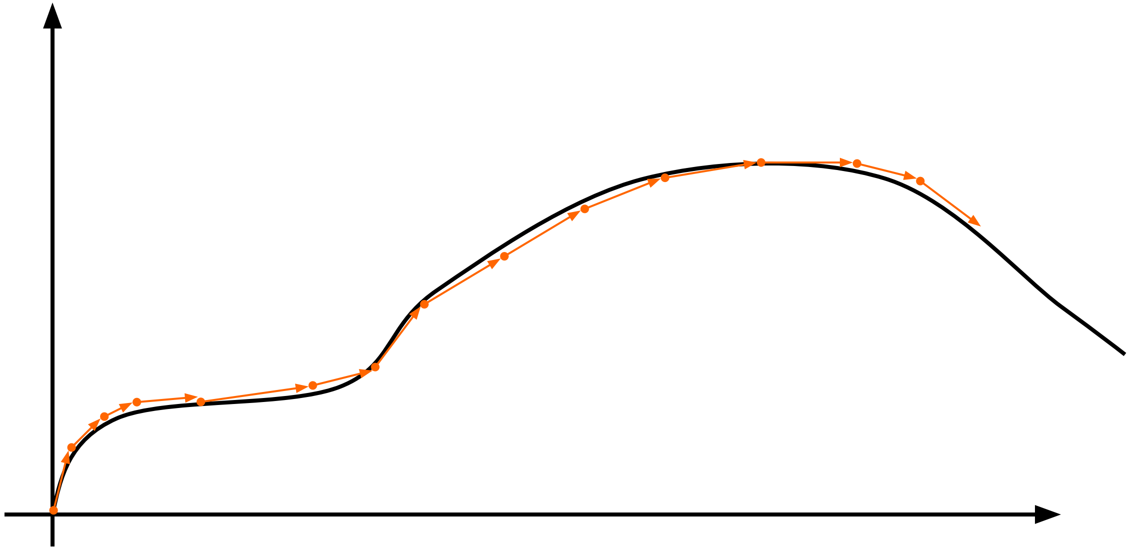
- MATLAB contains common integration methods
- Adaptive Simpson's quadrature (input is a function)
  - » **`q=quad('myFun',0,10)`**
    - ➢ q is the integral of the function **`myFun`** from 0 to 10
  - » **`q2=quad(@(x) sin(x).*x,0,pi)`**
    - ➢ q2 is the integral of **`sin(x).*x`** from 0 to pi
- 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

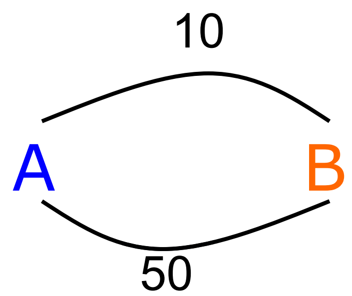
Initial conditions

- Inputs:
  - ➢ ODE function name (or anonymous function). This function should take inputs (t,y), and returns dy/dt
  - ➢ Time interval: 2-element vector with initial and final time
  - ➢ Initial conditions: column vector with an initial condition for each ODE. This is the first input to the ODE function
  - ➢ Make sure all inputs are in the same (variable) order
- Outputs:
  - ➢ t contains the time points
  - ➢ y contains the corresponding values of the variables

ODE Function

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

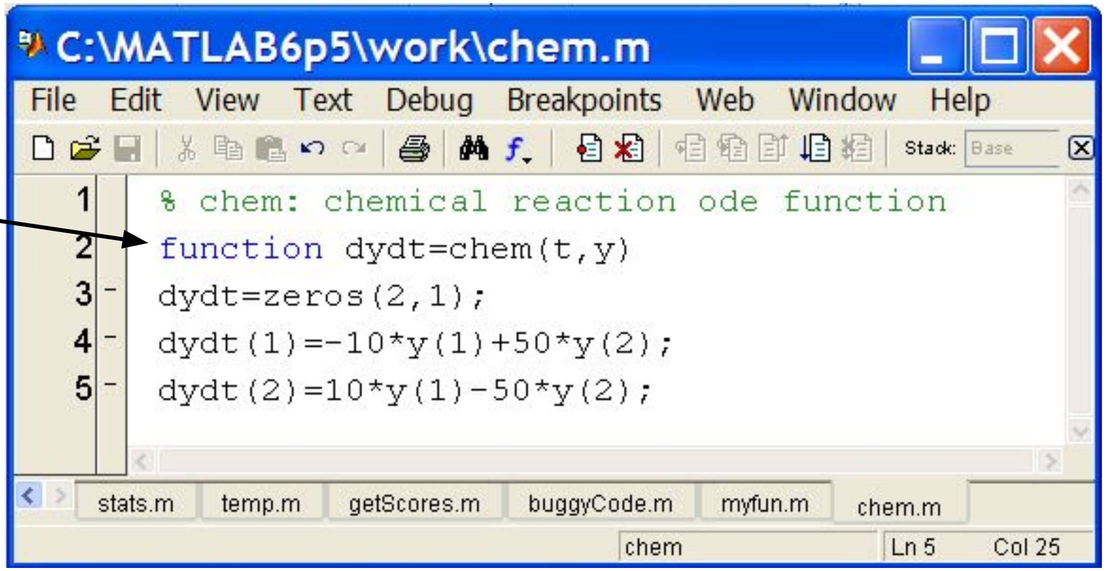
- Example: chemical reaction
  - ➢ Two equations



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

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

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

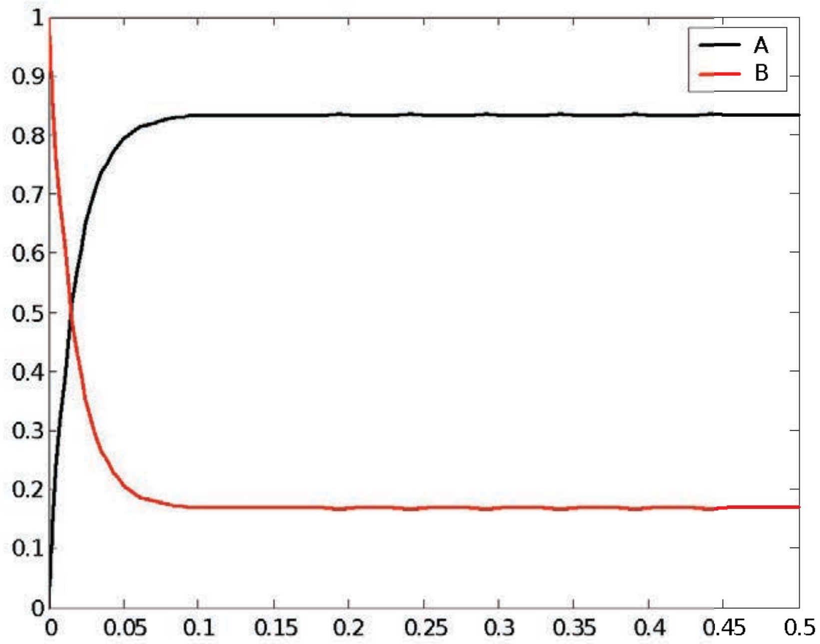


ODE Function: viewing results

- 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:

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

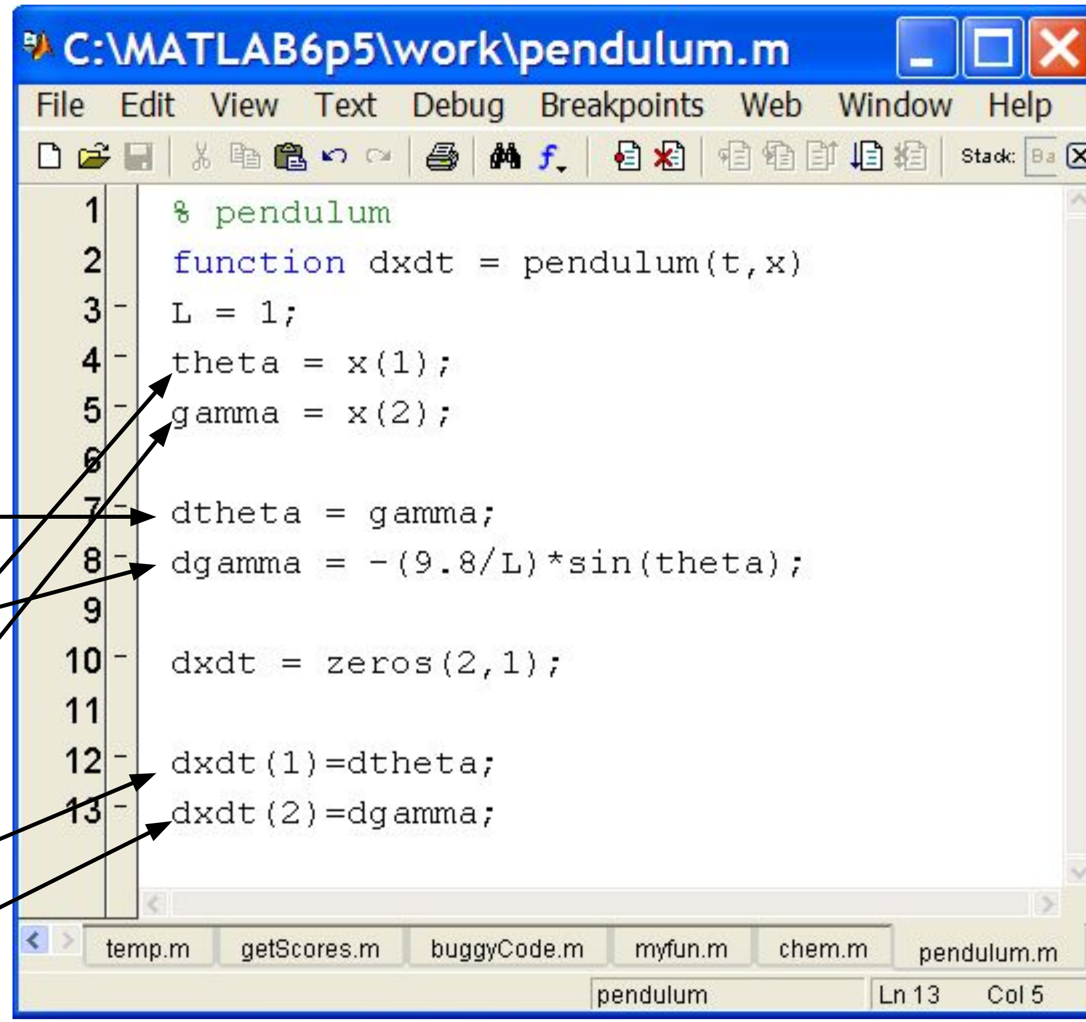
$$\ddot{\theta} = -\frac{g}{L}\sin(\theta)$$

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

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

$$\vec{x} = \begin{bmatrix} \theta \\ \gamma \end{bmatrix}$$

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



Plotting the Output

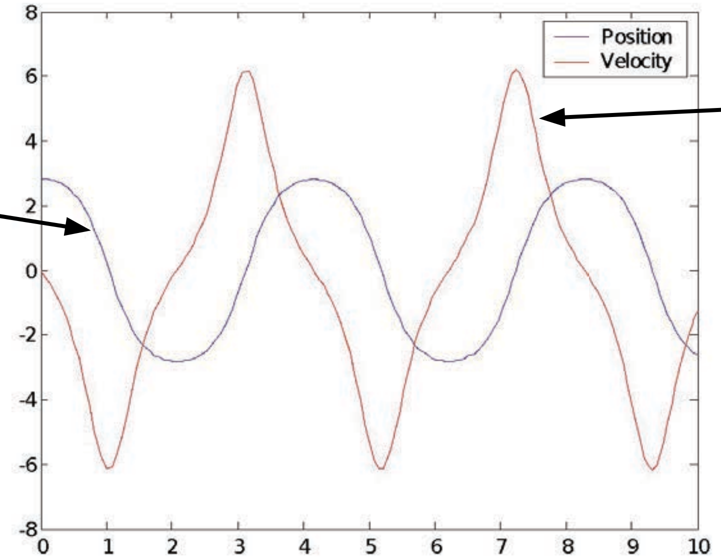
- We can solve for the position and velocity of the pendulum:

```
» [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)



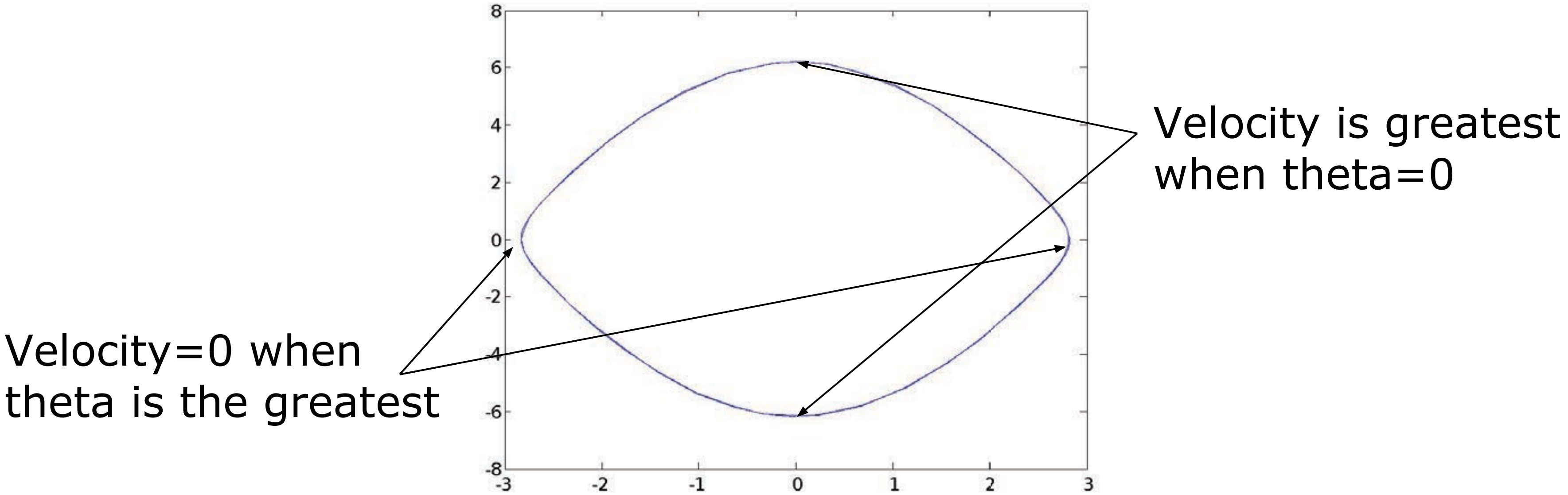
Velocity (m/s)

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 timestep by giving a vector of (increasing) times
    - ➢ The function value will be returned at the specified points

- 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: