Outline

• More plotting
  – A few other 3-D plotting functions
  – Basic volume visualization
• Creating functions
• Vectorized code
• Logical indexing
• Performance tips
2-D Plotting

- \( h = \text{scatter}(X,Y,S,C) \)
  - Will produce a scatter plot with marker size \( S \) and color \( C \)
  - \( C \) can be a vector of values
  - MATLAB uses current colormap and \( C \) to determine color of each marker
  - \( C \) needs to be the same length as \( X, Y \) if it is a vector
3-D Plotting

• \( h = \text{plot3}(X_1,Y_1,Z_1,...) \)
  – \text{plot3} is the 3-D analog of \text{plot} 
  – Will plot a 3-D plot of the set of data points 
  – Change linespec properties in the same fashion as \text{plot} 
  – Can plot matrices of data 
    • Plots lines obtained from columns of the \( x, y, z \) matrices passed to \text{plot3}
3-D Plotting

```matlab
>> t = 0:pi/50:10*pi;
plot3(sin(t),cos(t),t)
axis square; grid on
```
3-D Plotting

```matlab
>> [X,Y] = meshgrid([-2:0.1:2]);
Z = X.*exp(-X.^2-Y.^2);
plot3(X,Y,Z)
grid on
```

![3-D Plot](image.png)
3-D Plotting

- \( h = \text{scatter3}(X,Y,Z,S,C) \)
  - Same as \texttt{scatter}, except it will make a 3-D scatter plot

```matlab
>> [x, y, z] = sphere(16);
X = [x(:)*.5 x(:)*.75 x(:)];
Y = [y(:)*.5 y(:)*.75 y(:)];
Z = [z(:)*.5 z(:)*.75 z(:)];
S = repmat([1 .75 .5]*10,prod(size(x)),1);
C = repmat([1 2 3],prod(size(x)),1);
scatter3(X(:),Y(:),Z(:),S(:),C(:),'filled');
view(-60,60);
```
3-D Plotting

• Scaled mapping
  – Mentioned this with `imagesc` command
  – Used direct mapping of colors to colormap with `contour, surf` functions
  – `surf(Z,C)`
    • `C` is a matrix used to specify colors
    • You can set this to anything you want to highlight regions of the plot
3-D Plotting

```matlab
>> P = peaks(40);
C = del2(P);
surf(P,C)
colormap hot
```
3-D Plotting

• There are other properties to tinker with when creating a 3-D plot
  – View, lighting and transparency are three important ones
  – The next few slides are just some examples changing some of these properties
3-D Examples

• The Camera toolbar enables you to perform a number of viewing operations interactively. To use the Camera toolbar,
• Display the toolbar by selecting Camera Toolbar from the figure window's View menu or by typing CameraToolbar in the Command Window.
• Select the type of camera motion control you want to use by either clicking on the buttons or changing the CameraToolbar mode in the Command Window.
• Position the cursor over the figure window and click, hold down the right mouse button, then move the cursor in the desired direction.
• The display updates immediately as you move the mouse.
• The toolbar contains the following parts:
• Camera Motion Controls — These tools select which camera motion function to enable. You can also access the camera motion controls from the Tools menu.
• Principal Axis Selector — Some camera controls operate with respect to a particular axis. These selectors enable you to select the principal axis or to select nonaxis constrained motion. The selectors are grayed out when not applicable to the currently selected function. You can also access the principal axis selector from the Tools menu.
• Scene Light — The scene light button toggles a light source on or off in the scene (one light per axes).
• Projection Type — You can select orthographic or perspective projection types.
• Reset and Stop — Reset returns the scene to the standard 3-D view. Stop causes the camera to stop moving (this can be useful if you apply too much cursor movement). You can also access an expanded set of reset functions from the Tools menu.
3-D Examples

```
sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)
```

Camera toolbar
3-D Examples

>> ezsurf('sin(sqrt(x^2+y^2))/sqrt(x^2+y^2)',[-6*pi,6*pi])
>> view(0,75)
shading interp
lightangle(-45,30)
set(gcf,'Renderer','zbuffer')
set(findobj(gca,'type','surface'),...
    'FaceLighting','phong',...
    'AmbientStrength',.3,'DiffuseStrength',.8,....
    'SpecularStrength',.9,'SpecularExponent',25,....
    'BackFaceLighting','unlit')


Figure 1

\( \frac{\sin(\sqrt{x^2+y^2})}{\sqrt{x^2+y^2}} \)
3-D Examples

```matlab
>> vert = [1 1 1; 1 2 1; 2 2 1; 2 1 1; 1 1 2; 1 2 2; 2 2 2; 2 1 2;];
>> fac = [1 2 3 4; 2 6 7 3; 4 3 7 8; 1 5 8 4; 1 2 6 5; 5 6 7 8];
>> sphere(36);
>> h = findobj('Type','surface');
set(h,'FaceLighting','phong',...'
   'FaceColor','interp',...'
   'EdgeColor',[.4 .4 .4],...'
   'BackFaceLighting','lit')
hold on
patch('faces',fac,'vertices',vert,'FaceColor','y');
light('Position',[1 3 2]);
light('Position',[-3 -1 3]);
material shiny
axis vis3d off
hold off
>>
```
3-D Examples
3-D Examples

• findobj
  – h = findobj
    h = findobj('PropertyName',PropertyValue,...)
    h = findobj('PropertyName',PropertyValue,'-logicaloperator',
    'PropertyName',PropertyValue,...)
    h = findobj('-regexp','PropertyName','regexp',...)
    h = findobj('-property','PropertyName')
    h = findobj(objhandles,...)
    h = findobj(objhandles,'-depth',d,...)
    h = findobj(objhandles,'flat','PropertyName',PropertyValue,...)

• Logical operators
  – ‘-and’, ‘-or’, ‘-xor’, ‘-not’
3-D Examples

• Find all line objects in the current axes:
  – h = findobj(gca,'Type','line')

• Find all objects having a Label set to 'foo' and a String set to 'bar':
  – h = findobj('Label','foo','-and','String','bar');

• Find all objects whose String is not 'foo' and is not 'bar':
  – h = findobj('-not','String','foo','-not','String','bar');

• Find all objects having a String set to 'foo' and a Tag set to 'button one' and
  whose Color is not 'red' or 'blue':
  – h = findobj('String','foo','-and','Tag','button one',... '-and','-not',{Color,'red','-or','Color','blue'}))

• Find all objects for which you have assigned a value to the Tag property (that
  is, the value is not the empty string ''):
  – h = findobj('-regexp','Tag','[^\"]')

• Find all children of the current figure that have their BackgroundColor property
  set to a certain shade of gray ([.7 .7 .7]). This statement also searches the
  current figure for the matching property value pair.
  – h = findobj(gcf,'-depth',1,'BackgroundColor',[.7 .7 .7])
Volumes

• MATLAB can plot volume data in a variety of ways

• Volume data are datasets defined on a 3-D grid

• MATLAB defines two types of volume data
  – Scalar: one value at each point in the 3-D grid
  – Vector: two or three values at each grid point defining direction and magnitude of vector
Volumes

• contourslice(X,Y,Z,V,Sx,Sy,Sz)
• contourslice(X,Y,Z,V,Xi,Yi,Zi)
• contourslice(V,Sx,Sy,Sz)
• contourslice(V,Xi,Yi,Zi)

  – X, Y, Z are coordinates for volume V
  – Sx, Sy, Sz or Xi, Yi, Zi are vectors or points used to determine the contour slice location(s)
Volumes

- `slice(V, sx, sy, sz)`
- `slice(X, Y, Z, V, sx, sy, sz)`
- `slice(V, XI, YI, ZI)`
- `slice(X, Y, Z, V, XI, YI, ZI)`

- Create an orthogonal slice through volume data
- Variables passed to slice are the same as contour slice
Volumes

- \( \text{fv} = \text{isosurface}(X,Y,Z,V,\text{isovalue}) \)
- \( \text{fv} = \text{isosurface}(V,\text{isovalue}) \)
- \( \text{fvc} = \text{isosurface}(...,\text{colors}) \)
  - Again, \( X, Y, Z \) are 3-D arrays containing the coordinates for array \( V \)
  - \text{isovalue} is the value to create the isosurface at
  - can specify an array of colors to color-code isosurface
Volumes

- \texttt{streamline(X,Y,Z,U,V,W,startx,starty,startz)}
- \texttt{streamline(U,V,W,startx,starty,startz)}
  - \textbf{X, Y, Z} are the same as before
  - \textbf{U, V, W} are 3-D arrays specifying components of direction
  - \textbf{Startx, starty, startz}, can be scalars or vectors that determine the start locations for the streamlines

- Can also use a combination of \texttt{streamline} and \texttt{stream3}
  - \texttt{streamline(stream3(x,y,z,u,v,w,sx,sy,sz))}
Volumes

- \texttt{streamribbon}(X,Y,Z,U,V,W,startx,starty,startz)
- \texttt{streamribbon}(U,V,W,startx,starty,startz)

- Arguments are the same as streamline
- Function makes ribbons instead of lines
- Just a style option
Volumes

- Curl and Divergence can be calculated on volume data

- \([\text{curl}_x, \text{curl}_y, \text{curl}_z, \text{cav}] = \text{curl}(X,Y,Z,U,V,W)\)
  \([\text{curl}_x, \text{curl}_y, \text{curl}_z, \text{cav}] = \text{curl}(U,V,W)\)
  \([\text{curl}_z, \text{cav}] = \text{curl}(X,Y,U,V)\)
  \([\text{curl}_z, \text{cav}] = \text{curl}(U,V)\)

- \(\text{div} = \text{divergence}(X,Y,Z,U,V,W)\)
  \(\text{div} = \text{divergence}(U,V,W)\)
  \(\text{div} = \text{divergence}(X,Y,U,V)\)
  \(\text{div} = \text{divergence}(U,V)\)
Saving Figures

• Figures can be saved in a few ways
  – File -> Save or Save As
  – Save button on figure toolbar or Ctrl+S
  – Save As gives many options for output format
Saving Figures

- Two functions to save figures
  - `saveas(h,'filename.ext')` or `saveas(h,'filename','format')`
    - Where `h` is a figure handle (figure number) and `.ext` or `format`
- `print -dformat filename`
  - `print('-dpng','-r150',[outfile '.png'])`
    - Format, resolution, uses a variable to set filename
Functions

• Functions are M-files that accept input and output arguments
• The M-file needs to have the same name as the function
• Functions create their own local workspace
  – You won’t see variables used in function call in main workspace
Functions

• Look at example function: rank

```matlab
function r = rank(A, tol)
% RANK Matrix rank.
% RANK(A) provides an estimate of the number of linearly
% independent rows or columns of a matrix A.
% RANK(A,tol) is the number of singular values of A
% that are larger than tol.
% RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

s = svd(A);
if nargin==1
tol = max(size(A)') * max(s) * eps;
end
r = sum(s > tol);
```

• First line contains function keyword, output arguments, function name and input arguments
Functions

• Comment section appears when issuing command `help rank`

```matlab
function r = rank(A, tol)

% RANK Matrix rank.
% RANK(A) provides an estimate of the number of linearly
% independent rows or columns of a matrix A.
% RANK(A,tol) is the number of singular values of A
% that are larger than tol.
% RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

s = svd(A);
if nargin==1
    tol = max(size(A))' * max(s) * eps;
end
r = sum(s > tol);
```
Functions

- Actual function code follows
  - `r` is the return variable so that needs to be set somewhere in the function
- **rank** is an example of a primary function
Functions

• Another type of function you may use is the anonymous function

• Can be defined in a script M-file or at the command line
  – Does not need its own function M-file

• \( f = @(\text{arglist})\text{expression} \)

• Example:

```
>> sqr = @(x) x.^2;
>> a = sqr(5)

a =

   25
```
Functions

• Can create private functions, nest functions within functions, overloaded functions
• Global variables are also available
  – `global var_name`
  – This will declare `var_name` as a global variable
Vectorizing Code

• We’ve already hit on the basics
• Don’t use loops everywhere
  – Use them sparingly
    ```matlab
    >> i = 0;
    for t = 0:.01:10
        i = i + 1;
        y(i) = sin(t);
    end
    ```
  – Use vectorized version instead
    ```matlab
    >> t = 0:.01:10;
    y = sin(t);
    ```
Vectorizing Code

• Another way to vectorize your code is to take advantage of built-in MATLAB functions
  – Essentially all MATLAB functions can work on arrays of any dimension
  – Functions that work with manipulating arrays are much faster than doing things in loops
Vectorizing Code

• Example
  – Say you wanted to take an array and replicate and tile it to a larger size
  – Use the `repmat` function

```
>> A = [1 2 3; 4 5 6];
B = repmat(A,2,3);
>> B = repmat(A,2,3)

B =

1 2 3 1 2 3 1 2 3
4 5 6 4 5 6 4 5 6
1 2 3 1 2 3 1 2 3
4 5 6 4 5 6 4 5 6
```

– `repmat` creates a new array with m by n tiles of original array
Vectorizing Code

• Vectorizing code takes practice
• It is quite different from programming in C or FORTRAN
• Getting an algorithm to work first before vectorizing it seems to work for me
Logical Indexing

• Logical array
  – Array of 1’s or 0’s with 1 being true and 0 being false

• Can use a logical array to index another array

• Comes in handy when trying to find elements of arrays that meet certain criteria

• Can use find command as discussed before, but logical indexing is faster

• It is MUCH faster than using loops to find elements
Logical Indexing

• Example:

```matlab
>> rand('state',0);
R=rand(5,7);
q = R>0.3 & R<0.9
q =
0 1 1 1 1 0 0 0
0 1 1 0 1 0 1
1 0 0 0 1 1 1
1 1 1 1 1 0 0 0
1 1 0 1 0 0 1
```

• Create a matrix of random numbers
• Find numbers greater than 0.3 and less than 0.9
• \( q \) is a logical array that indicates which elements meet our criteria
Logical Indexing

• You can then use \( q \) to access the correct elements in \( R \)

\[
\begin{array}{cccccccccccc}
\text{Columns 1 through 11} \\
0.6068 & 0.4860 & 0.8913 & 0.7621 & 0.4565 & 0.8214 & 0.4447 & 0.6154 & 0.7919 & 0.7382 & 0.4057 \\
\text{Columns 12 through 19} \\
0.4103 & 0.8936 & 0.3529 & 0.8132 & 0.6038 & 0.7468 & 0.4451 & 0.4660
\end{array}
\]

• \( R(q) \) is a vector of the elements meeting our criteria
Short-Circuit Operators

• Logical operators && and || are short-circuit operators
• They evaluate the first argument and will skip the second if it is not needed
  – If the first argument of an and comparison is false, the second is not evaluated
  – Cannot use these in `find` because `find` works with arrays
Performance Tips

• In general, use vector version of algorithm if possible
• Use built-in functions when searching arrays, manipulating their size and shape
• Function files execute slightly faster than script M-files
  – You can make any script M-file into a function M-file by adding the `function` keyword as a wrapper
Performance Tips

• Overloaded functions are slower
• Don’t max out your system memory
• **load** and **save** are faster than lower level read/write functions
• Use short-circuit logical operators when appropriate
• Don’t have a bunch of other processes running when performing complex MATLAB scripts
Next Week

• Data manipulation functions
  – Regression, integration, etc

• Toolboxes
  – Mapping
  – Signal processing
  – Others?

• Other topics?
Questions?