Lecture 2: Visualization and Programming

Orhan Celiker

IAP 2019
Homework 1 Recap

Some things that came up:

- Plotting a straight line
  ```matlab
  >> 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:
  ```matlab
  >> x=1:10; y=(x-5).^2; z = x.*y;
  ```
Plotting

• Example
  » \texttt{x=linspace(0,4*pi,10);} \\
  » \texttt{y=sin(x);} \\

• Plot values against their index
  » \texttt{plot(y);} \\

• Usually we want to plot \( y \) versus \( x \)
  » \texttt{plot(x,y);} \\

\textbf{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: ![Graph with 10 x values](image1)
1000 x values: ![Graph with 1000 x values](image2)
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

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

**Inputs**
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`)
Write a function with the following declaration:

```python
function plotSin(f1)
```

In the function, plot a sine wave with frequency f1, on the interval [0, 2π]: $\sin(f_1x)$

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 \textit{mostly} standard relational operators
  \begin{itemize}
  \item equal \hspace{1em} ==
  \item \textbf{not} equal \hspace{1em} \sim =
  \item greater than \hspace{1em} >
  \item less than \hspace{1em} <
  \item greater or equal \hspace{1em} \geq
  \item less or equal \hspace{1em} \leq
  \end{itemize}

• Logical operators \hspace{1em} elementwise \hspace{1em} short-circuit (scalars)
  \begin{itemize}
  \item And \hspace{1em} & \hspace{1em} &&
  \item Or \hspace{1em} | \hspace{1em} ||
  \item \textbf{Not} \hspace{1em} \sim
  \item Xor \hspace{1em} xor
  \item All true \hspace{1em} all
  \item Any true \hspace{1em} any
  \end{itemize}

• Boolean values: zero is false, nonzero is true
• See \texttt{help} \hspace{1em} 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

• No need for parentheses: command blocks are between reserved words
• Lots of elseif’s? consider using switch

ELSEIF
if cond1
commands1
elseif cond2
commands2
else
commands3
end
• **for** loops: use for a known number of iterations

• MATLAB syntax:

```matlab
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**
while

• The while is like a more general for loop:
  ➢ No need to know number of iterations

• 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.-');`

• 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

MATLAB version 6.5. Courtesy of The MathWorks, Inc. Used with permission. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.
Line and Marker Options

- Everything on a line can be customized:
  ```matlab
  » 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
  
  ```matlab
  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
  
  ```matlab
  semilogx(x,y,'k');
semilogy(y,'r.-');
loglog(x,y);
  ```

- For example:
  
  ```matlab
  x=0:100;
semilogy(x,exp(x),'k.-');
  ```
3D Line Plots

- We can plot in 3 dimensions just as easily as in 2D
  
  ```matlab
  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
  ```matlab
  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
  ```matlab
  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**)

\[
\text{plotSin(6)} \quad \text{plotSin(1,2)}
\]
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
  - \([X, Y] = \text{meshgrid}(x, y);\)

- To get function values, evaluate the matrices
  - \( Z = \sin(X) \cdot \cos(Y); \)

- Plot the surface
  - \( \text{surf}(X, Y, Z) \)
  - \( \text{surf}(x, y, Z); \)

*Try typing \( \text{surf}(\text{membrane}) \)
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)
You can make surfaces two-dimensional by using `contour`.

```matlab
contour(X,Y,Z,'LineWidth',2)
```
- Takes same arguments as `surf`
- Color indicates height
- Can modify linestyle properties
- Can set colormap

```matlab
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:
  \[ Z = \sin(f_1x) + \sin(f_2y) \]
- `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

---

The figure above shows a 3-D plot generated by `plotSin(3,4)`.
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
(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 \( x = \sin(\text{linspace}(0, 10\pi, 100)) \), how many of the entries are positive?

<table>
<thead>
<tr>
<th>Using a loop and if/else</th>
</tr>
</thead>
<tbody>
<tr>
<td>count = 0;</td>
</tr>
<tr>
<td>for n = 1:length(x)</td>
</tr>
<tr>
<td>if x(n) &gt; 0</td>
</tr>
<tr>
<td>count = count + 1;</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Being more clever</th>
</tr>
</thead>
<tbody>
<tr>
<td>count = length(find(x &gt; 0));</td>
</tr>
<tr>
<td>Is there a better way?!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>length(x)</th>
<th>Loop time</th>
<th>Find time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.01</td>
<td>0</td>
</tr>
<tr>
<td>10,000</td>
<td>0.1</td>
<td>0</td>
</tr>
<tr>
<td>100,000</td>
<td>0.22</td>
<td>0</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1.5</td>
<td>0.04</td>
</tr>
</tbody>
</table>

- 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:
  ```matlab
  » 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:
  ```matlab
  » a=rand(1,100);  » a=rand(1,100);
  » b=zeros(1,100);  » b=[0 a(1:end-1)]+a;
  » for n=1:100
  »   if n==1
  »     b(n)=a(n);
  »   else
  »     b(n)=a(n-1)+a(n);
  »   end
  » end
  ```
  - Efficient and clean. Can also do this using `conv`
  - Slow and complicated
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:

```matlab
» 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:

```matlab
» 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
• When debugging functions, use \texttt{disp} to print messages
  » \texttt{disp('starting loop')}  
  » \texttt{disp('loop is over')}  
     ➢ \texttt{disp} prints the given string to the command window

• It's also helpful to show variable values
  » \texttt{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 \texttt{tic/toc}:

  » \texttt{tic}
  » \texttt{Mystery1;}
  » \texttt{a=toc;}
  » \texttt{Mystery2;}
  » \texttt{b=toc;}
  ➢ \texttt{tic} resets the timer
  ➢ Each \texttt{toc} returns the current value in seconds
  ➢ Can have multiple \texttt{toc}s per \texttt{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

<table>
<thead>
<tr>
<th>Filename</th>
<th>File Type</th>
<th>Calls</th>
<th>Total Time</th>
<th>Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>newplot</td>
<td>M-function</td>
<td>1</td>
<td>0.802 s</td>
<td></td>
</tr>
<tr>
<td>gcf</td>
<td>M-function</td>
<td>1</td>
<td>0.460 s</td>
<td></td>
</tr>
<tr>
<td>newplot/ObserveAxesNextPlot</td>
<td>M-subfunction</td>
<td>1</td>
<td>0.291 s</td>
<td></td>
</tr>
<tr>
<td>... matlab/graphics/private/clo</td>
<td>M-function</td>
<td>1</td>
<td>0.251 s</td>
<td></td>
</tr>
<tr>
<td>allchild</td>
<td>M-function</td>
<td>1</td>
<td>0.100 s</td>
<td></td>
</tr>
<tr>
<td>setdiff</td>
<td>M-function</td>
<td>1</td>
<td>0.050 s</td>
<td></td>
</tr>
</tbody>
</table>
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!