Administrivia
Midterm #2 graded
Homework #11 will be due Wed Jan. 4.
Midterm #2 graded
Homework #11 will be due Wed Jan. 4.
Homework #12 will be released over the break, due Friday, Jan 13.
Warmup Questions
Question #1

How can we produce this array?

A \( \text{ones}(3,3) - 2 \times \text{eye}(3,3) \)
B \( \text{ones}(3,3) + 2 \times \text{eye}(3,3) \)
C \( 2 \times \text{ones}(3,3) + \text{eye}(3,3) \)
D \( 2 \times \text{ones}(3,3) - \text{eye}(3,3) \)
How can we produce this array?

A  \texttt{ones(3,3) - 2*eye(3,3)}
B  \texttt{ones(3,3) + 2*eye(3,3)}
C  \texttt{2*ones(3,3) + eye(3,3)}
D  \texttt{2*ones(3,3) - eye(3,3)} \star
How do we access 6 in this array?

A  A(2,1)
B  A(1,2)
C  A(3,2)
D  A(2,3)
Question #2

How do we access 6 in this array?

A  A(2,1)
B  A(1,2)
C  A(3,2) ⚫
D  A(2,3)
MATLAB
a = [ 1 2 3 ]; %row vector
b = [ 1 2 3 ]'; %column vector
A = [ 1 2 3 ; 4 5 6 ]; %matrix
B = [ a ; b ]; % matrix composition
If $A$ is an $m \times n$ matrix (i.e., with $n$ columns), then the product $A \times x$ is defined for $n \times 1$ column vectors $x$. If we let $A \times x = b$, then $b$ is an $m \times 1$ column vector. In other words, the number of rows in $A$ (which can be anything) determines the number of rows in the product $b$. 
http://mathinsight.org/matrix_vector_multiplication
Matrix v. elementwise operations:

- Matrix operations are matrix–vector operations:

\[
\begin{pmatrix}
1 & 0 \\
0 & 1 \\
\end{pmatrix}
\begin{pmatrix}
2 \\
3 \\
\end{pmatrix}
= 
\begin{pmatrix}
2 \\
3 \\
\end{pmatrix}
\]
Matrix v. elementwise operations:
- Matrix operations are matrix–vector operations:

\[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
3
\end{pmatrix} =
\begin{pmatrix}
2 \\
3
\end{pmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
2 \\
3
\end{bmatrix}
\]

Array operations
Matrix v. elementwise operations:
- Matrix operations are matrix–vector operations:

\[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
3
\end{pmatrix} =
\begin{pmatrix}
2 \\
3
\end{pmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix} \times \begin{bmatrix}
2 & 3
\end{bmatrix}'
\]

\[
\begin{pmatrix}
1 & 2 \\
1 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
3
\end{pmatrix} =
\begin{pmatrix}
2 + 6 \\
3 + 2
\end{pmatrix} =
\begin{pmatrix}
8 \\
5
\end{pmatrix}
\]
Matrix v. elementwise operations:

Matrix operations are matrix–vector operations:

\[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
3
\end{pmatrix} =
\begin{pmatrix}
2 \\
3
\end{pmatrix}
\]

\[
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
2 & 3
\end{bmatrix}'
\]

\[
\begin{pmatrix}
1 & 2 \\
1 & 1
\end{pmatrix}
\begin{pmatrix}
2 \\
3
\end{pmatrix} =
\begin{pmatrix}
2 + 6 \\
3 + 2
\end{pmatrix} =
\begin{pmatrix}
8 \\
5
\end{pmatrix}
\]

\[
\begin{bmatrix}
1 & 2 \\
1 & 1
\end{bmatrix}
\begin{bmatrix}
2 & 3
\end{bmatrix}'
\]
Array operations

- Matrix v. elementwise operations:
  - Elementwise operations are spreadsheet-like operations:

\[
\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix}
\]
Matrix v. elementwise operations:

Elementwise operations are spreadsheet-like operations:

\[
\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \times \begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & 5 \end{pmatrix}
\]

\[
\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 & 4 \\ 3 & 5 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 5 \end{bmatrix}
\]
We can index arrays with arrays.

```matlab
A = 0:10:100;
B = A( [ 5, 9, 2, 2 ] );
```
Indexing arrays

- We can index arrays with arrays.

\[
A = 0:10:100;
B = A( [ 5,9,2,2 ] );
\]

- This permits slicing.

\[
A = 0:10:100;
B = A( 4:7 );
\]
In more dimensions:

\[
A = \begin{bmatrix} 1,2,3 ; 4,5,6 ; 7,8,9 \end{bmatrix}; \\
B = A( 1:2,1:2 ); \\
C = A(:,1:2);
\]
Functions can return several values.
Functions can return several values.

```matlab
function [ a, b ] = nonsense( x, y )
    a = x ^ 2;
    b = y ^ 3;
end

[ q, r ] = nonsense( 3, 4 )
```
plot works identically to plt.plot.
Plotting

- plot works identically to plt.plot.
- figure creates a new figure (window for plots).
Plotting

- plot works identically to plt.plot.
- figure creates a new figure (window for plots).

```matlab
x = 0:.1:2*pi;
y = sin(x);
figure
plot(x, y, 'o');
title('sin(x)');
xlabel('x values');
ylabel('y values');
```
Plotting

- plot works identically to plt.plot.
- figure creates a new figure (window for plots).

```matlab
x = 0:.1:2*pi;
y = sin(x);
figure
plot(x, y, 'o');
title('sin(x)');
xlabel('x values');
ylabel('y values');
```

- MATLAB also supplies an excellent plot editor.
Here's what we have now:

- functions
- array definitions, operations, slicing
- plotting
Here’s what we have now:
- functions
- array definitions, operations, slicing
- plotting

We’ve seen these parts—what about the rest of our “control structures”? 
%%% set parameters
alpha = 0.1;
tmax = 0.5;    % maximum time (s)
length = 3.0;  % length of material
dx = 0.2;      % mesh spacing
dt = 0.01;     % time step (s)

%%% data storage initialization
t = 0:dt:tmax;    % (s)
x = 0:dx:length;  % (m)
u = zeros(numel(t), numel(x));  % Kelvin
%% set initial condition
u(1,x>=1&x<=2) = 353.15; % Kelvin (= 80 deg C)

r = alpha * dt / (dx^2);
s = 1 - 2*r;

%% loop through time steps
for i = 2:1:numel(t)
    for j = 2:1:(numel(x)-1)
        u(i,j) = r*u(i-1,j-1) + s*u(i-1,j) + r*u(i-1,j+1);
    end
end
The `for` loop ranges over a set of possible values.
The `for` loop ranges over a set of possible values. This is *not* as flexible as Python’s `in` syntax—think of always having to loop over the *index* rather than the item.
We create a `for` loop as follows:
- statement `for var in range`, where you create `var` and provide `range`
- one or more statements
- closing statement `end`
We create a for loop as follows:
- statement for var in range, where you create var and provide range
- one or more statements
- closing statement end

Also have continue and break available.
function [ y ] = absolute( x )
    y = 0;
    if x >= 0
        y = x;
    else
        y = -x;
    end
We create an if/else statement as follows:
- the keyword if
- a logical comparison *(more on these!)*
- a **block** of code
We create an if/else statement as follows:

- the keyword `if`
- a logical comparison *(more on these!)*
- a **block** of code
- the keyword `elseif` *(note this!)*
- a new logical comparison
- a different **block** of code
We create an if/else statement as follows:

- the keyword \textit{if}
- a logical comparison (more on these!)
- a \textbf{block} of code
- the keyword \textit{elseif} (note this!)
- a new logical comparison
- a different \textbf{block} of code
- the keyword \textit{else}
- a different \textbf{block} of code
We create an if/else statement as follows:

- the keyword if
- a logical comparison (more on these!)
- a block of code
- the keyword elseif (note this!)
- a new logical comparison
- a different block of code
- the keyword else
- a different block of code
- the keyword end
MATLAB does not have a bool data type.
MATLAB does not have a bool data type.
Instead of True/False, MATLAB uses integers:
- 0 means False
- 1 means True
Logical statements

- MATLAB does not have a bool data type.
- Instead of True/False, MATLAB uses integers:
  - 0 means False
  - 1 means True
- Available logical operators include:
  - `<`, `>`, `<=`, `>=`, `==`, `~=`
  - `&&` for ‘and’, `||` for ‘or’
  - `ismember` checks equality of elements in arrays.
  - Also, logical operators as indices!
MATLAB does *not* have a bool data type. Instead of True/False, MATLAB uses integers:
- 0 means False
- 1 means True

Available logical operators include:
- $<$, $>$, $<=$, $>=$, $==$,
- `&&` for ‘and’, `||` for ‘or’
- `ismember` checks equality of elements in arrays.
- Also, logical operators as indices!
- `A( A<0 )`
Saving data uses save:

```matlab
A = [ 1 2 3 ; 4 5 6 ];
save( 'test', 'A' );
```
Saving data uses `save`:

```matlab
A = [ 1 2 3 ; 4 5 6 ];
save( 'test', 'A' );
```

Note that the `string` version of the variable name is required!

`load` also useful:

```matlab
A = load( 'test', 'A' );
```
A more advanced tool: importdata

data = importdata( 'rainfall.txt' );
A more advanced tool: `importdata`

```matlab
data = importdata( 'rainfall.txt' );
```

Can be used to process CSVs.
A more advanced tool: `importdata`

```matlab
data = importdata('rainfall.txt');```

- Can be used to process CSVs.
- Old process using `fopen`, `fscanf`, `fclose`, `fprintf` also common.
Images can also be opened as files.

A = importdata( 'rabbit-bw.jpg' );
image( A );
Images can also be opened as files.

```matlab
A = importdata('rabbit-bw.jpg');
image(A);
```

- Black and white images are arrays of 0s and 1s.
- Greyscale images are values from 0 and 1.
- Color images are three-dimensional arrays. (Why?)
- Variations exist depending on the underlying data.