

# MATH 3341: Introduction to Scientific Computing Lab

Libao Jin

University of Wyoming

September 02, 2020



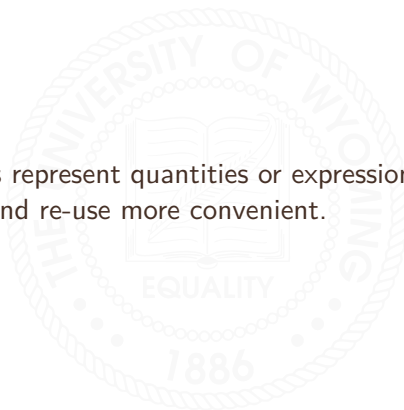
## Lab 02: Variables, Arrays, and Scripts



# Variables



Variables help us represent quantities or expressions in order to make their use and re-use more convenient.



# Naming Variables

- Must start with a letter.
- Followed by letters (a-z, A-Z) or numbers (0-9) or underscores (\_).
- Maximum 65 characters (excluding the .m extension).
- Must not be the same as any MATLAB reserved word.
- Space is not permitted.
- Case sensitive, i.e., `a`  $\neq$  `A`.



# Naming Variables

- Be as descriptive as possible with your variable names.
- Avoid built-in function/variable names (reserved keywords) such as `pi`, `sin`, `exp`, etc.
- Check if a name is already in use: `which variableName` or `exist variableName`.

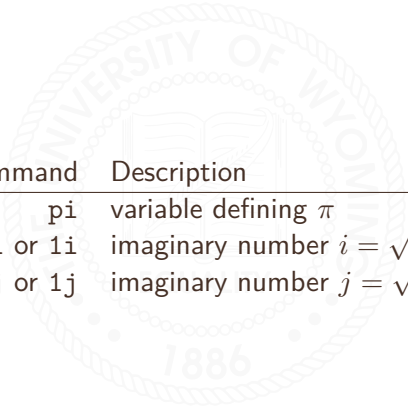


# Naming Conventions

- snake\_case: writing compound words or phrases in which the elements are separated with one underscore character (`_`) and no spaces, e.g. “foo\_bar”.
- camelCase: writing compound words or phrases such that each word or abbreviation in the middle of the phrase begins with a capital letter, with no intervening spaces or punctuation, e.g. “fooBar”
- Other conventions: Hungarian notation, positional notation, etc.
- Reference: [https://en.wikipedia.org/wiki/Naming\\_convention\\_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming))



# Default Variable Definitions



Command	Description
<code>pi</code>	variable defining $\pi$
<code>i</code> or <code>1i</code>	imaginary number $i = \sqrt{-1}$
<code>j</code> or <code>1j</code>	imaginary number $j = \sqrt{-1}$





## Arrays



# Array, Vector, and Matrix

- An array is a data form that can hold several values, all of one type.
- A vector is a 1-D array: we can define row vectors, column vectors.
- A matrix is a 2-D array.
- Also, we can define  $N$ -D array.
- The general notation for a vector or matrix is a list of values enclosed in square brackets `[]` separated by commas (space) or semi-colons (or the combination).



## Vector: []

- Row vector:  $x = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}$   
 $x = [1,2,3,4]$   
 $x = [1 \ 2 \ 3 \ 4]$
- Column vector:  $y = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$  or  $y = \begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix}^T$  or  $y = x^T$ .  
 $y = [1;2;3;4]$   
 $y = \text{transpose}([1 \ 2 \ 3 \ 4])$   
 $y = [1 \ 2 \ 3 \ 4]'$   
 $y = x'$   
 $y = x(:)$

Note: ' and .' are the infix notation for transpose, transpose operation.



## Vector: linspace vs. colon

- `linspace(from, to, n)` generates `n` points between `from` (inclusive) and `to` (inclusive). For example,  
`a = linspace(2, 6, 5) % same as a = [2 3 4 5 6]`
- `colon(from, step, upper_bound)` generates points between `from` (inclusive) and `upper_bound` (may not be inclusive) with spacing `step`. For example,  
`a = colon(2, 1, 6) % same as a = [2 3 4 5 6]`  
`a = colon(2, 2, 6) % same as a = [2 4 6]`  
`a = colon(2, 1, 7) % same as a = [2 3 4 5 6 7]`  
`a = colon(2, 2, 7) % same as a = [2 4 6]`
- `from:step:upper_bound` is same as `colon(from, step, upper_bound)`.



## Vector: linspace vs. colon

- `linspace(from, to, n)` is equivalent to `colon(from, (to - from) / (n - 1), to)`
- `colon(from, step, upper_bound)` is equivalent to `linspace(from, floor((upper_bound - from) / step) * step + from, floor((upper_bound - from) / step))`
- Use `linspace` when the number of points is given.
- Use `colon` when the spacing/step size is given.



## Vector: Slicing

- Define a row vector rowVec:

```
rowVec = [2,4,6,8,10]
```

```
rowVec = linspace(2,10,5)
```

```
rowVec = colon(2,2,10)    % or rowVec = 2:2:10
```

- array(i): the i-th entry of array, where i is called the index:

i	1	2	3	4	5
rowVec(i)	2	4	6	8	10



## Vector: Slicing

i	1	2	3	4	5
rowVec(i)	2	4	6	8	10

- Extract one entry from a vector: For example, to extract 6 from rowVec and assign it to x:  
`x = rowVec(3)`
- Extract multiple entries from a vector: For example, to extract 2, 6, 8 from rowVec and assign it to x:  
`x = rowVec([1,3,4])`
- Extract multiple contiguous entries from a vector: For example, to extract 4, 6, 8 from rowVec and assign it to x:  
`x = rowVec([2,3,4])`  
`x = rowVec(2:4)`



## Vector: Append/Delete Element

```
% 1-D array
rowVec = 1:5
rowVec(end + 1) = 6 % append 6 to rowVec
rowVec = [rowVec,7] % append 7 to rowVec
rowVec(5) = []      % delete 5 from rowVec
rowVec(2:4) = []    % delete 2, 3, 4 from rowVec
```





# Vector Operations

- `sum(vec)/prod(vec)`: sum/product of all elements of `vec`.
- `max(vec)/min(vec)`: maximum/minimum of `vec`.
- `rowVec = rowVec1 .* rowVec2`: elementwise multiplication, where `rowVec(i) = rowVec1(i) * rowVec2(i)`.
- `rowVec .* colVec`: Kronecker product. If `rowVec` has length `m` and `colVec` has length `n`, then the resulting matrix is `m`-by-`n`.
- `dot(vec1, vec2)`: dot product of `vec1` and `vec2`, `vec1` and `vec2` must be of the same length.
- `sum(rowVec1 .* rowVec2)`: `dot(rowVec1, rowVec2)`.
- `rowVec1 * rowVec2'`: `dot(rowVec1, rowVec2)`.
- `indices = find(vec > n)`: find indices of elements greater than `n` in `vec`. Note: `>` can also be `<`, `==`.



## Dimension: size, length, reshape

- `size(array)`: size of array. If array is n-dimensional, size will return a vector of length n.
- `size(array, 1)`: number of rows of array.
- `size(array, 2)`: number of columns of array.
- `length(vec)`: length of vector vec, equivalent to `max(size(vec))`.
- `reshape(array, dim1, dim2, dim3, ...)`.  
`rowVec = 1:8`  
`matrix = reshape(rowVec, 2, 4)`  
`% same as matrix = [1,3,5,7;2,4,6,8]`
- `reshape(array, prod(size(array)), 1)` is same as `array(:)`.



## Matrix: []

Define a  $2 \times 3$  matrix  $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$

```
A = [1,2,3;4,5,6]
```

or

```
row1 = [1,2,3]
```

```
row2 = [4,5,6]
```

```
A = [row1;row2]
```

or

```
col1 = [1;4]
```

```
col2 = [2;5]
```

```
col3 = [3;6]
```

```
A = [col1,col2,col3]
```



## Matrix: zeros, ones, eye, rand, randn, magic

- `zeros(m, n)`: define a  $m$ -by- $n$  matrix with zeros.  
`zeroRowVec = zeros(1, 5)`  
`zeroColVec = zeros(5, 1)`  
`zeroMatrix = zeros(5, 5)`  
`zeroMatrix = zeros(5)`
- `ones(m, n)`: define a  $m$ -by- $n$  matrix with ones.
- `eye(m, n)`: define a  $m$ -by- $n$  matrix with diagonals being ones.
- `rand(m, n)`: define a  $m$ -by- $n$  matrix with uniformly distributed numbers.
- `randn(m, n)`: define a  $m$ -by- $n$  matrix with normally distributed numbers.
- `magic(n)`: define a  $n$ -by- $n$  magic square with row sums, column sums and diagonal sum being equal.



# Matrix: Slicing

- Define a matrix `mat`  
`mat = reshape(1:8, 2, 4)`
- `array(i, j)`: the entry of array at row `i` and column `j`, where `i` is called row index, `j` is called column index:

mat(i, j) \ j		1	2	3	4
i \					
1		1	3	5	7
2		2	4	6	8



# Matrix: Slicing

mat(i, j) \ j		1	2	3	4
i					
1		1	3	5	7
2		2	4	6	8

Extract multiple rows and multiple columns from `mat`: For example, to extract entries at row 1, row 2, and column 2, column 4:

```
A = mat([1,2], [2,4])
```

```
A = mat(1:2, [2,4])
```

```
A = mat(1:end, [2,4])
```

```
A = mat(:, [2,4])
```



## Matrix: Append/Delete Element

```
% 2-D array
matrix = magic(5)
matrix(:, end + 1) = 1:5    % append a column vector
matrix = [matrix,[6:10]']  % append a column vector
matrix(end + 1, :) = 1:7    % append a row vector
matrix = [matrix;8:14]      % append a row vector
matrix(:,6) = []            % delete column 6
matrix(:,3:5) = []          % delete column 3, 4, 5
matrix(2:4,:) = []          % delete row 2, 3, 4
```



# Matrix Operations

- `mat = mat1 .* mat2`: elementwise multiplication, where  $\text{mat}(i, j) = \text{mat1}(i, j) * \text{mat2}(i, j)$ .
- `mat = mat1 * mat2`: matrix multiplication, where `mat1` is  $m$ -by- $p$ , `mat2` is  $p$ -by- $n$ , and `mat` is  $m$ -by- $n$ .
- `sum/prod(mat, 'all')`: sum/product of all elements of `mat`.
- `sum/prod(mat, 1)`: column sums/products.
- `sum/prod(mat, 2)`: row sums/products.
- `max/min(mat, [], 'all')`: maximum/minimum of `mat`.
- `max/min(mat, [], 1)`: column maximums/minimums.
- `max/min(mat, [], 2)`: row maximums/minimums.
- `[row, col] = find(mat > n)`: find indices of elements greater than  $n$  in `mat`, `row/col` stores row/column indices.





# Matrix Operations

- `[V, D] = eig(mat)`: `V(:, i)` and `D(i, i)` are the *i*-th eigenvector and eigenvalue of `mat`.
- `d = diag(mat, k)`: extract *k*-th diagonal elements that is above ( $k > 0$ ) / below ( $k < 0$ ) the main diagonal.
- `mat = diag(d, k)`: construct a matrix with *k*-th diagonal elements being `d`.
- `mat = diag(diag(mat, k), k)`: set elements to zero except the *k*-th diagonal elements.
- `fliplr(mat)`: flip `mat` in left/right direction.
- `flipud(mat)`: flip `mat` in up/down direction.
- `rot90(mat, k)`: rotate `mat`  $k * 90$  degrees.



## *N*-D array: reshape and slicing

Define 3-D array using reshape:

```
rowVec = 1:8  
array = reshape(rowVec, 2, 2, 2);  
length(size(array)) % check the dimension
```

or using slicing:

```
slice1 = [1,2;3,4]  
slice2 = [5,6;7,8]  
C(:,:,1) = slice1  
C(:,:,2) = slice2
```



## Char Array vs. String Array

```
str = "abc"
arrayOfChars1 = 'abc'
arrayOfChars2 = ['a','b','c']
arrayOfChars1 == arrayOfChars2 % return logical 1 (true)
arrayOfChars1 == str           % return logical 1 (true)
class(str)                     % string
class(arrayOfChars1)           % char
[arrayOfChars1,arrayOfChars2] % return 'abcabc'
[arrayOfChars1;arrayOfChars2] % return ['abc';'abc']
[str,str]                      % return ["abc","abc"]
[str;str]                      % return ["abc";"abc"]
```



## Cell Array: array of elements of different types

- `cell(n)`: create 1-D cell array of length `n`
- `cell(m,n)`: create 2-D cell array of size `m` by `n`
- Create a cell array of types `char`, `string`, `double`:

```
cellArray = {[1,2,3], "abc", 'def'}  
cellArray{1}      % return [1,2,3]  
cellArray{2}      % return "abc"  
cellArray{3}      % return 'def'  
cellArray{4} = 'ghi'  
cellArray{4}      % return 'ghi'
```



## Application: Image Processing

- A grayscale image is a 2-D array of pixels, each pixel has a integer value that represent depth of color.
- A colored image is a 3-D array of pixels with RGB channels, each channel is a 2-D array.
- `img = imread(filename)`: read image from graphics file `filename` and assign it `img`.
- `imshow(img)`: display image `img` in handle graphics figure.
- `imwrite(img, filename)`: write image `img` to graphics file named `filename`.

```
uw = imread('UW.png');  
uwFlipud = flipud(uw);  
imshow(uwFlipud);  
imwrite(uwFlipud, 'UW_flipud.png');
```



# Summary

Command	Description
<code>transpose</code> or <code>'</code>	Non-conjugate transpose of a vector
<code>linspace</code>	Linearly spaced vector
<code>logspace</code>	Logarithmically spaced vector
<code>colon</code> or <code>:</code>	Colon
<code>zeros</code>	Zeros array
<code>ones</code>	Ones array
<code>eye</code>	Identity matrix
<code>rand</code>	Uniformly distributed pseudorandom numbers
<code>randn</code>	Normally distributed pseudorandom numbers
<code>magic</code>	Magic square
<code>size</code>	Size of array
<code>length</code>	Length of vector
<code>reshape</code>	Reshape array



# Summary

Command	Description
<code>diag</code>	Diagonal matrices and diagonals of a matrix
<code>cell</code>	Create cell array
<code>sum/prod</code>	Sum/Product of elements
<code>min/max</code>	Minimum/Maximum of elements
<code>dot</code>	Vector dot product
<code>find</code>	Find indices of nonzero elements
<code>eig</code>	Find eigenvalues and eigenvectors
<code>diag</code>	Diagonal matrices and diagonals of a matrix
<code>fliplr/flipud</code>	Flip an array
<code>rot90</code>	Rotate an array 90 degrees
<code>imread/imwrite</code>	Read/Write image from graphics file
<code>imshow</code>	display image in Handle Graphics figure
<code>uint8</code>	Convert to unsigned 8-bit integer



## Additional Commands

Command	Description
<code>iskeyword</code>	Check if input is a keyword
<code>who</code>	List current variables
<code>whos</code>	List current variables, long form
<code>which</code>	Locate functions and files
<code>clear</code>	Clear variables and functions from memory
<code>clc</code>	Clear command window
<code>clf</code>	Clear current figure
<code>close</code>	Close figure
<code>exist</code>	Check existence of variable/script/function/folder/class
<code>disp</code>	Display array





## Script Files



A script file is simply a file that contains a chain of commands that you edit in a separate window, then execute with a single mouse click or command. This is where we can define variables, perform calculations and leave comments to remind us what the file calculates.



# File Naming Conventions

- Start with a letter, followed by letters or numbers or underscore, maximum 63 characters (excluding the .m extension), and must not be the same as any MATLAB reserved word.
- None of the conventions matter to MATLAB itself: they only matter to the people writing the code, and the people maintaining the code (usually a much harder task), and to the people paying for the code (you'd be amazed how much gets written into contract specifications.)
- Reference:  
<https://www.mathworks.com/matlabcentral/answers/30223-what-are-the-rules-for-naming-script-files>



# Put Comments to Your Script File

```
% MATH 3341, Semester Year  
% Lab 02: Variables, Arrays, and Scripts  
% Author: first_name last_name  
% Date: mm/dd/yyyy
```



# Useful MATLAB Shortcuts

- Windows shortcuts

- Press `Ctrl` + `A` to select all
- Press `Ctrl` + `I` to adjust indentation
- Press `Ctrl` + `R` to comment
- Press `Ctrl` + `T` to uncomment

- macOS shortcuts

- Press `command` + `A` to select all
- Press `command` + `I` to adjust indentation
- Press `command` + `/` to comment
- Press `command` + `T` to uncomment





# L<sup>A</sup>T<sub>E</sub>X Primer



# table Environment

```
\begin{table}[!hbtpr]
  \caption{This is a table}
  \begin{tabular}{rcl}
    \toprule
    Column 1 & Column 2 & Column 3 \\
    \midrule
    1          & 1          & 1          \\
    12         & 12         & 12         \\
    123        & 123        & 123        \\
    \bottomrule
  \end{tabular}
\end{table}
```



# table Environment

Table 1: This is a table

Column 1	Column 2	Column 3
1	1	1
12	12	12
123	123	123





# figure Environment

```
\begin{figure}[!hbt]  
  \centering  
  \includegraphics[height=0.3\textheight]{figure.pdf}  
  \caption{Plot of  $\sin x$ }  
  \label{fig:sin}  
\end{figure}
```

generates

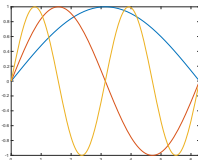


Figure 1: Plot of  $\sin x$



## `\left` and `\right` vs. `\big`, `\Big`, `\Bigg`

```

\begin{align*}
\|x\|_2 &= \big(\sum_{i=1}^n x_i^2 \big)^{1/2}, \\
\|x\|_2 &= \Big(\sum_{i=1}^n x_i^2 \Big)^{1/2}, \\
\|x\|_2 &= \Bigg(\sum_{i=1}^n x_i^2 \Bigg)^{1/2}, \\
\|x\|_2 &= \left(\sum_{i=1}^n x_i^2 \right)^{1/2}. \\
\end{align*}

```

generates

$$\|x\|_2 = \left(\sum_{i=1}^n x_i^2\right)^{1/2}, \|x\|_2 = \left(\sum_{i=1}^n x_i^2\right)^{1/2},$$

$$\|x\|_2 = \left(\sum_{i=1}^n x_i^2\right)^{1/2}, \|x\|_2 = \left(\sum_{i=1}^n x_i^2\right)^{1/2}.$$



# Links

```
\href{https://www.google.com}{Google}
```

Google

Or simply

```
\url{https://www.google.com}
```

```
https://www.google.com
```



## case Environment

```
$$  
f(x) =  
\begin{cases}  
5x + 4 & \text{if } x \leq 1, \\  
3x^2 + 6 & \text{if } x > 1  
\end{cases}  
$$
```

generates

$$f(x) = \begin{cases} 5x + 4 & \text{if } x \leq 1, \\ 3x^2 + 6 & \text{if } x > 1 \end{cases}$$



# Cross-Reference

```
\begin{equation}  
\label{eq:ls}  
A \mathbf{x} = \mathbf{b}.  
\end{equation}
```

The expression `\eqref{eq:ls}` is a linear system.

generates

$$A\mathbf{x} = \mathbf{b}. \tag{1}$$

The expression (1) is a linear system.



## Cross-Reference

```
\begin{table}[!hbt]
\caption{$y = 2x$}
\label{tab:xy}
\begin{tabular}{cc}
\toprule
$x$ & $y$ \\
\midrule
$6$ & $12$ \\
$7$ & $14$ \\
$8$ & $16$ \\
\bottomrule
\end{tabular}
\end{table}
```

Table `\ref{tab:xy}` gives the result of  $y = 2x$ .



# Cross-Reference

Table 2:  $y = 2x$ 

$x$	$y$
6	12
7	14
8	16

Table 2 gives the result of  $y = 2x$ .

