

Image Processing

Introduction to MATLAB

MATLAB Basics

- MATLAB can be thought of as a super-powerful graphing calculator
- In addition it is a programming language
 - ⌚ MATLAB is an interpreted language, like Java
 - ⌚ Commands executed line by line

Help/Docs

- `help`
 - 🕒 **The most** important function 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 doc
- To get a nicer version of help with examples and easy-to-read descriptions:
 - » `doc sin`
- To search for a function by specifying keywords:
 - » `doc` + Search tab

Variable Types

- MATLAB is a weakly typed language
 - ⌚ No need to initialize variables!
- MATLAB supports various types, the most often used are
 - » 3.84
 - ⌚ 64-bit double (default)
 - » 'a'
 - ⌚ 16-bit char
- Most variables you'll deal with will be vectors or matrices of doubles or chars
- Other types are also supported: complex, symbolic, 16-bit and 8 bit integers, etc. You will be exposed to all these types through the homework

Naming variables

- To create a variable, simply assign a value to a name:
 - » `var1=3.14`
 - » `myString='hello world'`
- Variable names
 - 🕒 first character must be a LETTER
 - 🕒 after that, any combination of letters, numbers and `_`
 - 🕒 CASE SENSITIVE! (`var1` is different from `Var1`)
- Built-in variables. Don't use these names!
 - 🕒 `i` and `j` can be used to indicate complex numbers
 - 🕒 `pi` has the value 3.1415926...
 - 🕒 `ans` stores the last unassigned value (like on a calculator)
 - 🕒 `Inf` and `-Inf` are positive and negative infinity
 - 🕒 `NaN` represents 'Not a Number'

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
 - » `d = 13/3;`

Arrays

- Like other programming languages, arrays are an important part of MATLAB
- Two types of arrays

(1) matrix of numbers (either double or complex)

(2) cell array of objects (more advanced data structure)

**MATLAB makes vectors easy!
That's its power!**



Row Vectors

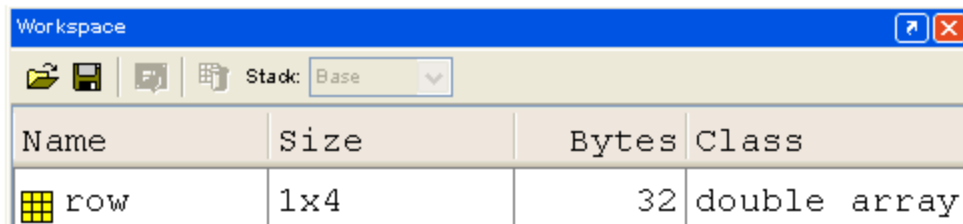
- Row vector: comma or space separated values between brackets

```
» row = [1 2 5.4 -6.6]
```

```
» row = [1, 2, 5.4, -6.6];
```

- Command window: `>> row=[1 2 5.4 -6.6]`

- Workspace:



The screenshot shows the MATLAB Workspace window. At the top, there are icons for Home, Save, and Help, and a dropdown menu for the stack (Base). Below this is a table with the following data:

Name	Size	Bytes	Class
row	1x4	32	double array

Column Vectors

- Column vector: semicolon separated values between brackets

» `column = [4;2;7;4]`

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

Command window:

```
column =
```

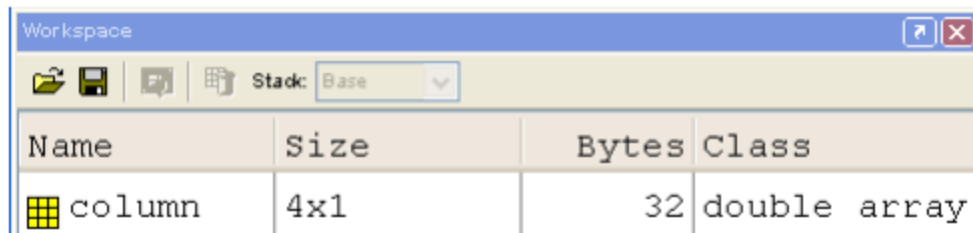
```
4
```


```
2
```

```
7
```

```
4
```

- Workspace:



Name	Size	Bytes	Class
 column	4x1	32	double array

size & length

- You can tell the difference between a row and a column vector by:

- 🕒 Looking in the workspace
- 🕒 Displaying the variable in the command window
- 🕒 Using the size function

```
>> size(row)           >> size(column)

ans =

     1     4           ans =

     4     1
```

- To get a vector's length, use the length function

```
>> length(row)        >> length(column)

ans =

     4                 ans =

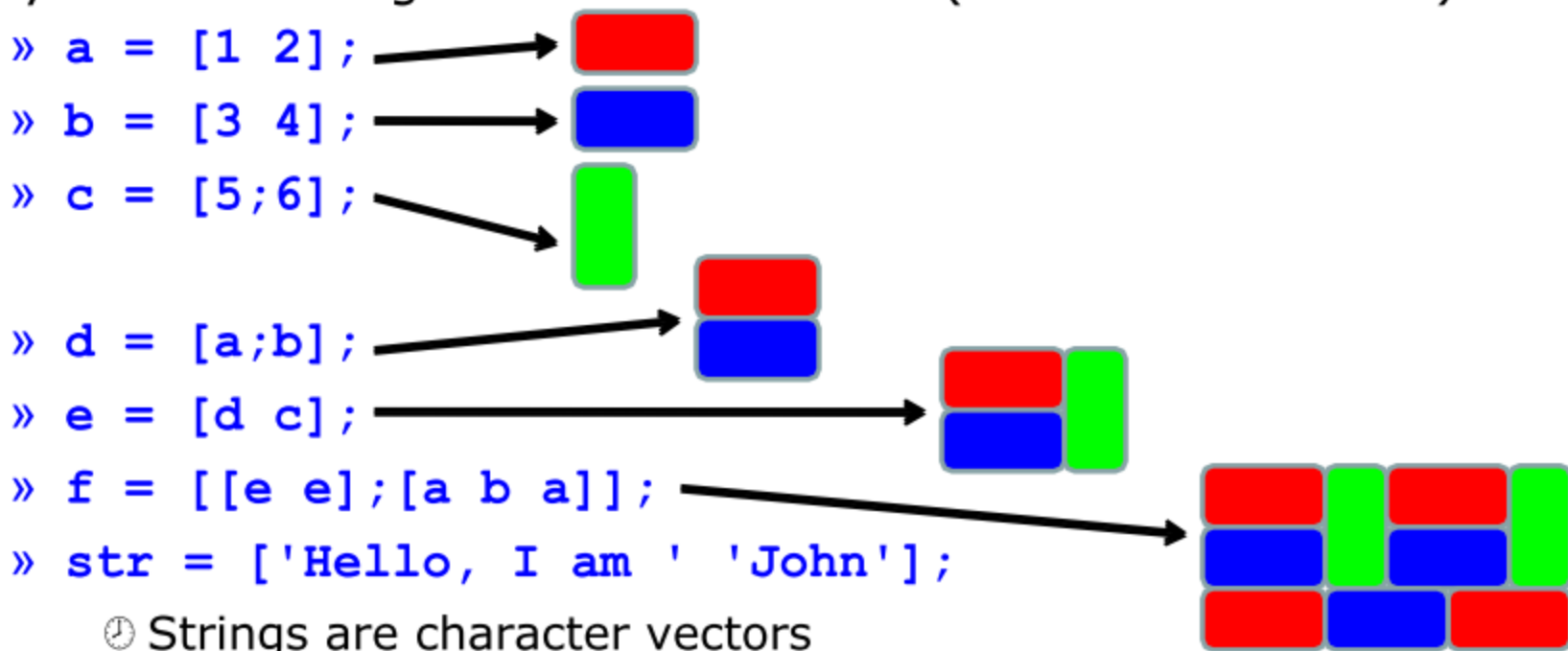
     4
```

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)



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
 - 🕒 myfile.mat file is saved in the current directory
 - 🕒 Default working directory is
 - » `\MATLAB`
 - 🕒 Make sure you're in the desired folder when saving files. Right now, we should be in:
 - » `MATLAB\IAPMATLAB\day1`
- Use **clear** to remove variables from environment
 - » `clear a b`
 - 🕒 look at workspace, the variables a and b are gone
- Use **load** to load variable bindings into the environment
 - » `load myFile`
 - 🕒 look at workspace, the variables a and b are back
- Can do the same for entire environment
 - » `save myenv; clear all; load myenv;`

Basic Scalar Operations

- Arithmetic operations (+, -, *, /)
 - » 7/45
 - » (1+i)*(2+i)
 - » 1 / 0
 - » 0 / 0
- Exponentiation (^)
 - » 4^2
 - » (3+4*j)^2
- Complicated expressions, use parentheses
 - » ((2+3)*3)^0.1
- Multiplication is NOT implicit given parentheses
 - » 3(1+0.7) gives an error
- To clear command window
 - » `clc`

Built-in Functions

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

Transpose

- The transpose operators 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, i.e. transposes and conjugates all complex numbers
- For vectors of real numbers `.'` and `'` give same result

Addition and Subtraction

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

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

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

- The following would give an error
 - » `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

Operators: element-wise

- 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

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} .* \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} = \text{ERROR}$$
$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} .* \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 4 \\ 4 \\ 3 \end{bmatrix}$$
$$3 \times 1 .* 3 \times 1 = 3 \times 1$$

$$\begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix} .* \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 6 \\ 3 & 6 & 9 \end{bmatrix}$$
$$3 \times 3 .* 3 \times 3 = 3 \times 3$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} .^2 = \begin{bmatrix} 1^2 & 2^2 \\ 3^2 & 4^2 \end{bmatrix}$$

Can be any dimension

Operators: standard

- Multiplication can be done in a standard way or element-wise
- Standard multiplication ($*$) is either a dot-product or an outer-product
 - ⌚ Remember from linear algebra: inner dimensions must MATCH!!
- Standard exponentiation ($^$) can only be done on square matrices or scalars
- Left and right division ($/$ \backslash) is same as multiplying by inverse
 - ⌚ Our recommendation: just multiply by inverse (more on this later)

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} * \begin{bmatrix} 4 \\ 2 \\ 1 \end{bmatrix} = 11$$

$1 \times 3 * 3 \times 1 = 1 \times 1$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} ^2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

Must be square to do powers

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

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

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 (useful for representing uninitialized variables)

The general function call is:

```
var=zeros(M,N);
```

Number of rows

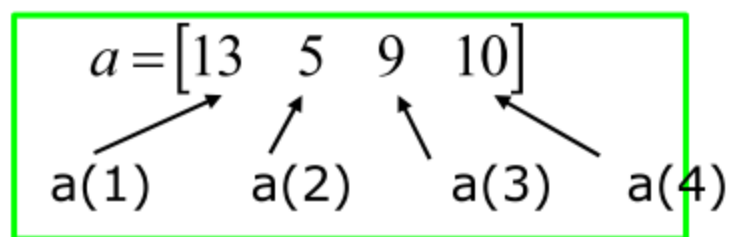
Number of columns

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 isn't specified, default is 1
- To initialize logarithmically spaced values use **logspace**
 - 🕒 similar to **linspace**, but see **help**

Vector Indexing


- MATLAB indexing starts with **1**, not **0**
- $a(n)$ returns the n^{th} 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.
 - » $x = [12 \ 13 \ 5 \ 8];$
 - » $a = x(2:3);$ \longrightarrow $a = [13 \ 5];$
 - » $b = x(1:end-1);$ \longrightarrow $b = [12 \ 13 \ 5];$

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**


$$\begin{array}{l} b(1,1) \longrightarrow \begin{bmatrix} 14 & 33 \end{bmatrix} \longleftarrow b(1,2) \\ b(2,1) \longrightarrow \begin{bmatrix} 9 & 8 \end{bmatrix} \longleftarrow b(2,2) \end{array}$$


$$\begin{array}{l} b(1) \longrightarrow \begin{bmatrix} 14 & 33 \end{bmatrix} \longleftarrow b(3) \\ b(2) \longrightarrow \begin{bmatrix} 9 & 8 \end{bmatrix} \longleftarrow b(4) \end{array}$$

- Picking submatrices
 - » `A = rand(5)` % shorthand for 5x5 matrix
 - » `A(1:3,1:2)` % specify contiguous submatrix
 - » `A([1 5 3], [1 4])` % specify rows and columns

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];`

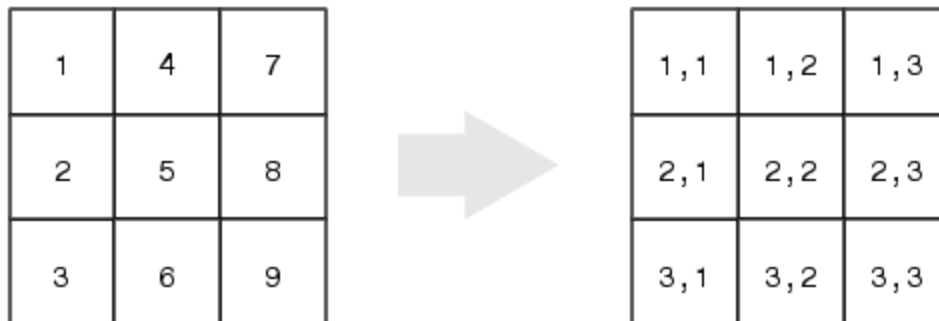
» `c(2,:)=[3 6];` `%replaces second row of c`

Advanced Indexing 2

- MATLAB contains functions to help you find desired values within a vector or matrix
 - » `vec = [5 3 1 9 7]`
- To get the minimum value and its index:
 - » `[minVal,minInd] = min(vec);`
 - 🕒 `max` works the same way
- To find any the indices of specific values or ranges
 - » `ind = find(vec == 9);`
 - » `ind = find(vec > 2 & vec < 6);`

In Matrices

- To convert between subscripts and indices, use **ind2sub**, and **sub2ind**. Look up **help** to see how to use them.



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
⌚ 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

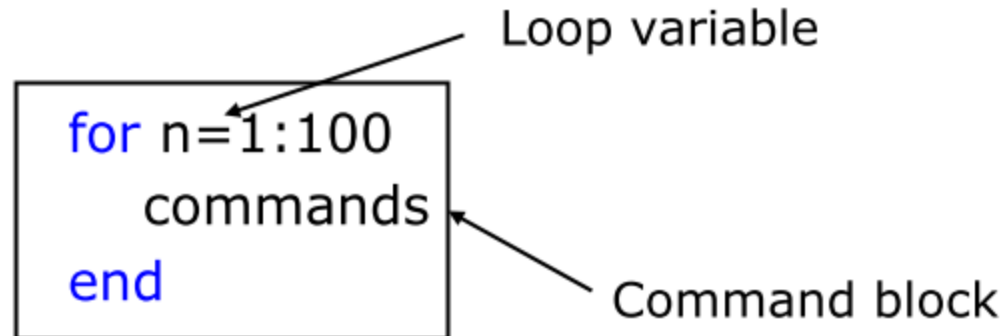
```
ELSE
if cond
  commands1
else
  commands2
end
```

```
ELSEIF
if cond1
  commands1
elseif cond2
  commands2
else
  commands3
end
```

- **No need for parentheses:** command blocks are between reserved words

for

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



- 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:
 - 🕒 Don't 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!

Working with Images

- Use `imread` to open an image
 - `Im1=imread('sample.bmp');`
- Images are stored as matrices. Try:
 - `size(Im1)`
- Display images using `imshow`
 - `imshow(Im1)`

Example Demo

- Reading an image
- Changing the values of some pixels
- Display original and modified images
- Draw histogram