Module 1: Programming in MATLAB

Relational and Logical Operations

FPRINTF: Alternate Displaying Function

Combine literal text with numeric data.

- Number of digits to display

```
fprintf('There are %d days in a year.\n', 365)
```

- Complex number

```
z = exp(1i*pi/4);
fprintf('%f+%fi\n', real(z), imag(z));
```

FPRINTF: Formatting Operator

```
%[field width][precision][conversion character]
```

e.g. `%12.5f`.

- `%`: marks the beginning of a formatting operator
- `[field width]`: maximum number of characters to print; optional
- `[precision]` number of digits to the right of the decimal point; optional
- `[conversion character]`

`%d`	integer
`%f`	fixed-point notation
`%e`	exponential notation
`%g`	the more compact of `%f` or `%e`
`%s`	string array
`%x`	hexadecimal

Relational Operators

How are two numbers X and Y related?

- `[X>Y]` Is X greater than Y?
- `[X<Y]` Is X less than Y?
- `[X>=Y]` Is X greater than or equal to Y?
- `[X<=Y]` Is X less than or equal to Y?
- `[X==Y]` Is X equal to Y?
- `[X~=Y]` Is X not equal to Y?

The symbols used between X and Y are called the **relational operators**.

Logical Variables and Logical Operators

- A relational statement evaluates to either **True(1)** or **False(0)**; these are called **logical variables** or **boolean variables**.
- As arithmetic operators (`+`, `-`, `*`, `/`) put together two numbers and produce other numbers, **logical operators** combine two logical variables to produce other logical variables.
- **Logical Operators**: *and*, *or*, *not*, *xor*

Logical Operator: `&&` (AND)

Let `A` and `B` be two logical variables. The `&&` operation is completely defined by the following truth table:

`A`	`B`	`A && B`
F	F	F
F	T	F
T	F	F
T	T	T

Note that `A && B` is true if and only if both `A` and `B` are true.

Logical Operator: `||` (OR)

Let `A` and `B` be two logical variables. The `||` operation is completely defined by the following truth table:

`A`	`B`	`A || B`
F	F	F
F	T	T
T	F	T
T	T	T

Note that `A || B` is false if and only if both `A` and `B` are false.

Logical Operator: `xor` (exclusive or)

This is a special variant of the `||` operator.

A	B	xor(A,B)
F	F	F
F	T	T
T	F	T
T	T	F

Note that `xorg(A,B)` is true if only one of `A` or `B` is true.

Logical Operator: ~ (NOT)

This is a negation operator.

A	~A
F	T
T	F

Combination of Logical Operations

Let `A` and `B` be logical variables. Then `~(A && B)` and `~A || ~B` are equivalent:

A	B	A && B	~(A && B)
F	F		
F	T		
T	F		
T	T		

A	B	~A	~B	~A || ~B
F	F			
F	T			
T	F			
T	T			

Quadratics Revisited

Consider a monic quadratic function $q(x) = x^2 + bx + c$ on a close interval $[L, R]$.

- Critical point: $x_c = -b/2$
- If $x_c \in (L, R)$, $q(x)$ attains the (global) minimum at x_c; otherwise, the minimum occurs at one of the endpoints $x = L$ or $x = R$.

Question

Write a program which determines whether the critical point of $q(x)$ falls on the interval.

Initialization

```
b = input('Enter b: ');
c = input('Enter c: ');
L = input('Enter L: ');
R = input('Enter R (L<R): ');
clc
fprintf('Function: x^2 + bx + c, b = %5.2f, c = %5.2f\n', b, c)
fprintf('Interval: [L, R], L = %5.2f, R = %5.2f\n', L, R)
xc = -b/2;
```

Main Fragment

```
if L < xc && xc < R
    fprintf('Interior critical point at x_c = %5.2f\n', xc)
else
    disp('Either xc <= L or xc >= R')
end
```

Main Fragment – another way

```
if xc <= L || xc >= R
    disp('Either xc <= L or xc >= R')
else
    fprintf('Interior critical point at x_c = %5.2f\n', xc)
end
```

Main Fragment – yet another way

```
if ~(xc <= L || xc >= R)
    fprintf('Interior critical point at x_c = %5.2f\n', xc)
else
    disp('Either xc <= L or xc >= R')
end
```

The simplest `if` statement?

So far, we have seen

- `if-else` statement
- `if-elseif-else` statement

The simplest `if` statement is of the form

```
if [condition]
  [statements to run]
end
```

Input Errors

If a user mistakenly provides L that is larger than R, fix it silently by swapping L and R.

```
if L > R
    tmp = L;
    L = R;
    R = tmp;
end
```

I will show you how to send an error message and halt a program later.

Exercise 1: Simple Minimization Problem

Question

Write a program which $x_{\min} \in [L, R]$ at which $q(x)$ is minimized and the minimum value $q(x_{\min})$.

- This can be done with `if-elseif-else`

Exercise 2: Leap Year

Question

Write a script which determines whether a given year is a leap year or not. A year is a leap year if

- it is a multiple of 4;
- it is not a multiple of 100;
- it is a multiple of 400.

Useful: `mod` function.

Pseudocode

```
if [YEAR] is not divisible by 4
    it is a common year
elseif [YEAR] is not divisible by 100
    it is a leap year
elseif [YEAR] is not divisible by 400
    it is a common year
else
    it is a leap year
end
```

Exercise 3: Angle Finder

Question

Let x and y be given, not both zero. Determine the angle $\theta \in (-\pi, \pi]$ between the positive x-axis and the line segment connecting the origin to (x, y).

Four quandrants:

- 1st or 4th ($x >= 0$): $\theta = \tan^{-1}(y/x)$
- 2nd ($x < 0, y >= 0$): $\theta = \tan^{-1}(y/x) + \pi$
- 3rd ($x < 0, y < 0$): $\theta = \tan^{-1}(y/x) - \pi$

Useful: `atan` (inverse tangent function)

Extended Inverse Tangent

```
if x > 0
    theta = atan(y/x)
elseif y >= 0
    theta = atan(y/x) + pi
else
    theta = atan(y/x) - pi
end
```

- MATLAB provides a function that exactly does this: `atan2(x,y)`.
- **Further Exploration**: What would you do if you are asked to find the angle $\theta \in [0, 2\pi)$, with `atan` alone or with `atan2`?

FOR-Loops

Approximating π

Suppose the circle $x^2 + y^2 = n^2$, $n \in \mathbb{N}$, is drawn on graph paper.

- The area of the circle can be approximated by counting the number uncut grids, N_{in}.

$$\pi n^2 \approx N_{\text{in}},$$

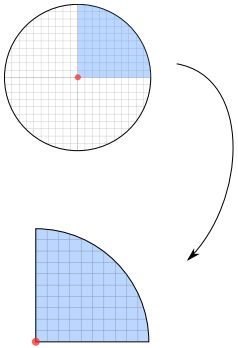
and so

$$\pi \approx \frac{N_{\text{in}}}{n^2}.$$

- Using symmetry, may only count the grids in the first quadrant and modify the formula accordingly:

$$\pi \approx \frac{4N_{\text{in},1}}{n^2},$$

where $N_{\text{in},1}$ is the number of inscribed grids in the first quadrant.



Approximating π

Problem Statement

Write a script that inputs an integer n and displays the approximation of π by

$$\rho_n = \frac{4N_{\text{in},1}}{n^2},$$

along with the (absolute) error $|\rho_n - \pi|$.

Note. The approximation gets enhanced and approaches the true value of π as $n \to \infty$.

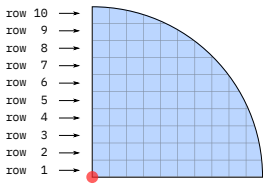
Strategy: Iterate

The key to this problem is to count the number of uncut grids in the first quadrant programmatically.

Set $N_{\text{in},1} = 0$.

> Count the number of uncut grids in `row 1`. Add that to $N_{\text{in},1}$.
>
> Count the number of uncut grids in `row 2`. Add that to $N_{\text{in},1}$.
>
> ⋮
>
> Count the number of uncut grids in `row 10`. Add that to $N_{\text{in},1}$.

Set $\rho_{10} = 4N_{\text{in},1}/10^2$.



MATLAB Way

The repeated counting can be delegated to MATLAB using `for`-loop. The procedure outlined above turns into

Assume `n` is initialized and set $N_{\mathrm{in},1}$ to zero.

`for k = 1:n`

> Count the number of uncut grids in `row k`. Add that to $N_{\mathrm{in},1}$.

`end`

Set $\rho_{10} = 4N_{\mathrm{in},1}/10^2$.

Counting Uncut Tiles

The problem is reduced to counting the number of uncut grids in each row.

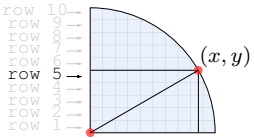
- The x-coordinate of the intersection of the top edge of the kth row and the circle $x^2 + y^2 = n^2$ is

$$x = \sqrt{n^2 - k^2}.$$

- The number of uncut grids in the kth row is the largest integer less than or equal to this value, *i.e.*,

$$\lfloor \sqrt{n^2 - k^2} \rfloor. \qquad \text{(floor function)}$$

- MATLAB provides `floor`.



For $n = 10$ and $k = 5$:

$$\begin{aligned} x &= \sqrt{n^2 - k^2} \\ &= \sqrt{10^2 - 5^2} = 8.6602\ldots \end{aligned}$$

Main Fragment Using FOR-Loop

```
N1 = 0;
for k = 1:n
    m = floor(sqrt(n^2 - k^2));
    N1 = N1 + m;
end
rho_n = 4*N1/n^2;
```

Exercise. Complete the program.

Exercise 1: Overestimation

Question

Note that ρ_n is always less than π. If N_1 denotes the total number of grids, both cut and uncut, within the quarter disk, then $\mu_n = 4N_1/n^2$ is always larger than π. Modify the previous (complete) script so that it prints ρ_n, μ_n, and $\mu_n - \rho_n$.

- `ceil`, an analogue of `floor`, is useful.

Notes on FOR-Loop

- The construct is used when a code fragment needs to be repeatedly run. The number of repetition is known in advance.

```
for <loop variable> = 1:<arithmetic expression>
<code fragment>
end
```

- Examples:

```
for k = 1:3
    fprintf('k = %d\n', k)
end
```

```
nIter = 100;
for k = 1:nIter
    fprintf('k = %d\n', k)
end
```

Caveats

Run the following script and observe the displayed result.

```
for k = 1:3
    disp(k)
    k = 17;
    disp(k)
end
```

- The loop header `k = 1:3` guarantees that `k` takes on the values 1, 2, and 3, one at a time even if `k` is modified within the loop body.
- However, it is a recommended practice that the value of the loop variable is *never* modified in the loop body.

Simulation Using `rand`

`rand` is a built-in function which generate a (uniform) "random" number between 0 and 1. Try:

```
for k = 1:10
    x = rand();
    fprintf('%10.6f\n', x);
end
```

Let's use this function to solve:

Question

A stick with length 1 is split into two parts at a random breakpoint. *On average*, how long is the shorter piece?

Program Development – Single Instance

Consider breaking *one* stick.

- Random breakage can be simulated with `rand`; denote by $x \in (0, 1)$.
- The length of the shorter piece can be determined using `if`-construct; denote by $s \in (0, 1/2)$.

```
x = rand();       % x: the location of breakage
if x <= 0.5       % if x ⩽ 0.5
    s = x;        % shorter part has length x
else              % otherwise
    s = 1-x       % shorter part has length 1 − x
end
```

Program Development – Multiple Instances

- Repeat the previous multiple times using a `for`-loop. Pseudocode: if 1000 breaks are to be simulated:

```
nBreaks = 1000;
for k = 1:nBreaks
<code from previous page>
end
```

- But how are calculating the *average* length of the shorter pieces?

Calculating Average Using Loop

Recall how the total number of uncut grids were calculated using iterations.

Assume `n` is initialized and set $N_{\text{in},1}$ to zero.

`for k = 1:n`

> Count the number of uncut grids in `row k`. Add that to $N_{\text{in},1}$.

`end`

The value of $N_{,1}$ is the total numbers of uncut grids.

Similarly, we can compute an average by:

Assume `n` is initialized and set s to zero.

`for k = 1:n`

> Simulate a break and find the length of the shorter piece. Add that to s.

`end`

Set $s_{\text{avg}} = s/\mathrm{n}$.

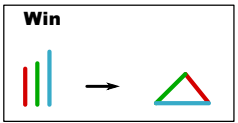
Complete Solution

```
nBreaks = 1000;
s = 0;
for k = 1:nBreaks
    x = rand();
    if x <= 0.5
        s = s + x;
    else
        s = s + (1-x);
    end
end
s_avg = s/nBreaks;
```

Exercise 2: Game of 3-Stick

Game: 3-Stick

Pick three sticks each having a random length between 0 and 1. You win if you can form a triangle using the sticks; otherwise, you lose.



Question

Estimate the probability of winning a game of 3-Stick by simulating one million games and counting the number of wins.

WHILE-Loops

Understanding Loops

Question 1

How many lines of output are produced by the following script?

```
for k = 100:200
    disp(k)
end
```

A 99

B 100

C 101

D 200

Understanding Loops

Question 2

How many lines of output are produced by the following script?

```
for k = 100:200
    if mod(k,2) == 0
        disp(k)
    end
end
```

A 50

B 51

C 100

D 101

FOR-Loop: Tips

- Basic loop header:

```
for <loop var> = 1:<ending value>
```

- To adjust starting value:

```
for <loop var> = <starting value>:<ending value>
```

- To adjust step size:

```
for <loop var> = <starting value>:<step size>:<ending value>
```

Examples

- To iterate over 1, 3, 5, …, 9: [*step size = 2*]

```
for k = 1:2:9
```

or

```
for k = 1:2:10
```

- To iterate over 10, 9, 8, …, 1: [*negative step size*]

```
for k = 10:-1:1
```

Need for Another Loop

- `For`-loops are useful when the number of repetitions is known in advance.

  "Simulate the tossing of a fair coin 100 times and print the number of Heads."

- It is not very suitable in other situations such as

  "Simulate the tossing of a fair coin until the gap between the number of Heads and that of Tails reaches 10."

  We need another loop construct that terminates as soon as $|N_{\mathrm{H}} - N_{\mathrm{T}}| = 10$.

WHILE-Loop Basics

WHILE-loop is used when a code fragment needs to be executed repeatedly *while* a certain condition is true.

```
while <continuation criterion>
<code fragment>
end
```

- The number of repetitions is *not* known in advance.
- The continuation criterion is a boolean expression, which is evaluated at the start of the loop.
  - If it is true, the loop body is executed. Then the boolean expression is evaluated again.
  - If it is false, the flow of control is passed to the end of the loop.

Simple WHILE-Loop Examples

```
k = 1; n = 10;
while k <= n
    fprintf('k = %d\n', k)
    k = k+1;
end
```

```
k = 1;
while 2^k < 5000
    k = k+1;
end
fprintf('k = %d\n', k)
```

FOR-Loop to WHILE-Loop

A `for`-loop can be written as a `while`-loop. For example,

FOR

```
s = 0;
for k = 1:4
    s = s + k;
    fprintf('%2d %2d\n', k, s)
end
```

WHILE

```
k = 0; s = 0;
while k < 4
    k = k + 1; s = s + k;
    fprintf('%2d %2d\n', k, s)
end
```

- Note that `k` needed to be initialized before the `while`-loop.
- The variable `k` needed to be updated inside the `while`-loop body.

Up/Down Sequence

Question

Pick a random integer between 1 and 1,000,000. Call the number n and repeat the following process:

- If n is even, replace n by $n/2$.
- If n is odd, replace n by $3n + 1$.

Does it ever take more than 1000 updates to reach 1?

- To generate a random integer between 1 and k, use `randi`, *e.g.*,

```
randi(k)
```

- To test whether a number n is even or odd, use `mod`, *e.g.*,

```
mod(n, 2) == 0
```

Attempt Using FOR-Loop

```
for step = 1:1000
    if mod(n,2) == 0
        n = n/2;
    else
        n = 3*n + 1;
    end
    fprintf(' %4d %7d\n', step, n)
end
```

- Note that once n becomes 1, the central process yields the following pattern:

$$1, 4, 2, 1, 4, 2, 1, \ldots$$

- This program continues to run even after n becomes 1.

Solution Using WHILE-Loop

```
step = 0;
while n > 1
    if mod(n,2) == 0
        n = n/2;
    else
        n = 3*n + 1;
    end
    step = step + 1;
    fprintf(' %4d %7d\n', step, n)
end
```

- This shuts down when n becomes 1!

Exercise: Gap of 10

Question

Simulate the tossing of a fair coin until the gap between the number of Heads and that of Tails reaches 10.

Summary

- `For`-loop is a programming construct to execute statements repeatedly.

```
for <loop index values>
<code fragment>
end
```

- `While`-loop is another construct to repeatedly execute statements. Repetition is controlled by the termination criterion.

```
while <termination criterion is not met>
<repeat these statements>
end
```

Arrays in MATLAB

Introduction to Arrays

Vectors and matrices are often collectively called **arrays**.

Notation

- $\mathbb{R}^m$ (or $\mathbb{C}^m$): the set of all real (or complex) **column vectors** with m elements.
- $\mathbb{R}^{m\times n}$ (or $\mathbb{C}^{m\times n}$): the set of all real (or complex) $m \times n$ matrices.
- If $\mathbf{v} \in \mathbb{R}^m$ with $\mathbf{v} = (v_1, v_2, \ldots, v_m)^{\mathrm{T}}$, then for $1 \leqslant i \leqslant m$, $v_i \in \mathbb{R}$ is called the ith *element* or the ith *index* of $\mathbf{v}$.
- If $A \in \mathbb{R}^{m\times n}$ with $A = (a_{i,j})$, then for $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$, $a_{i,j} \in \mathbb{R}$ is the element in the ith row and jth column of A.

Creating Arrays

- A *row vector* is created by

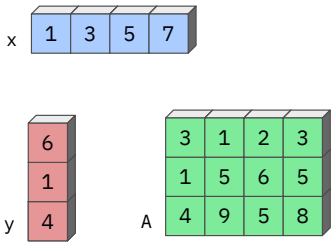
```
x = [1 3 5 7];
x = [1,3,5,7];
```

- A *column vector* is created by

```
y = [6; 1; 4];
y = [6 1 4].';
```

- A matrix is formed by

```
A = [3 1 2 3;
     1 5 6 5;
     4 9 5 8];
```



The MATLAB expression `x.'` means $\mathbf{x}^{\mathrm{T}}$ while `x'` means $\mathbf{x}^{\mathrm{H}} = (\mathbf{x}^*)^{\mathrm{T}}$.

Shape of Arrays

- To find the number of elements of a vector:

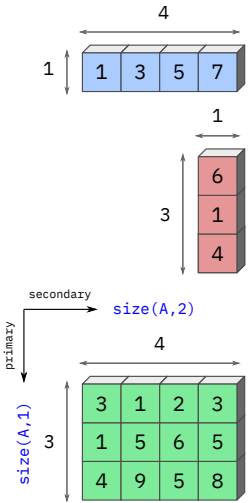
```
length(x)
length(y)
```

- To find the number of rows/columns of an array:

```
size(A,1) % # of rows
size(A,2) % # of cols
size(A)   % both
```

- To find the total number of elements of an array:

```
numel(A)
```



Shape of Arrays (Notes)

- For a matrix `A`, `length(A)` yields the larger of the two dimensions.
- The result of `size(A)` can be stored in two different ways:

```
szA = size(A)
[m, n] = size(A)
```

Q. How are they different?

- All of the following generate *empty arrays*.

```
[]
[1:0]
[1:0].'
```

Q. What are their *sizes*? What are their `numel` values?

Getting/Setting Elements of Arrays

- To access the ith element of a vector:

```
x(2)
y(3)
```

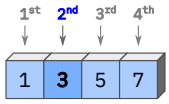
- To access the (i, j)-element of a matrix:

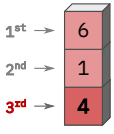
```
A(2,4)
```

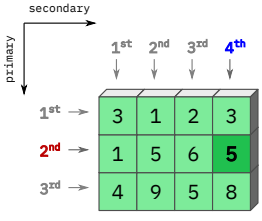
- To assign values to a specific element:

```
x(2) = 2
A(2,4) = 0
```

- Indices start at **1** in MATLAB, not at 0!







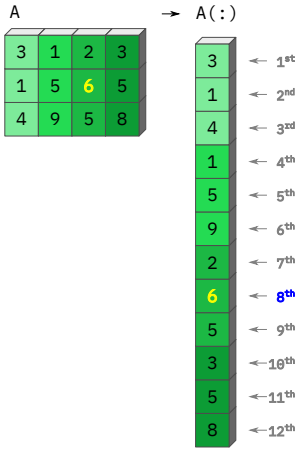
Linear Indexation and Straightening of Matrix

- MATLAB uses *column-major* layout by default, meaning that the elements of the columns are contiguous in memory.
- Consequently, one can get/set an element of a matrix using a single index.

```
A(8)
```

- An array can be put into a column vector using

```
A(:)
```



Two Kinds of Transpose

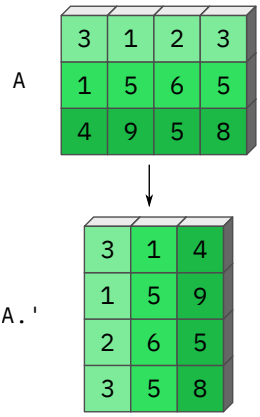
- The transpose of an array: A^{T}

```
A.'
```

- The conjugate transpose of an array: $A^{\mathrm{H}} = A^{*} = \overline{A}^{\mathrm{T}}$

```
A'
```

- If $A \in \mathbb{R}^{m \times n}$, $A^{\mathrm{H}} = A^{\mathrm{T}}$. So, if `A` is a real array, `A.'` and `A'` are equivalent.



Standard Arithmetic Operation

Standard arithmetic operations seen in linear algebra are executed using the familiar symbols.

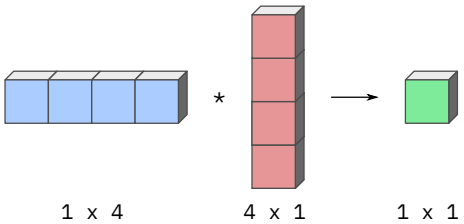
- Let `A`, `B` $\in \mathbb{R}^{m\times n}$ and `c` $\in \mathbb{R}$.
  - `A` $\pm$ `B`: elementwise addition/subtraction ($A \pm B$)
  - `A` $\pm$ `c`: *shifting* all elements of `A` by $\pm$ `c` ($A \pm c$)
- Let `A` $\in \mathbb{R}^{m\times p}$, `B` $\in \mathbb{R}^{p\times n}$, and `c` $\in \mathbb{R}$.
  - `A*B`: the $m \times n$ matrix obtained by the *linear algebraic* multiplication (AB)
  - `c*A`: scalar multiple of `A` (cA)
- Let `A` $\in \mathbb{R}^{m\times m}$ and `n` $\in \mathbb{N}$.
  - `A^n`: the `n`-th power of `A`; the same as `A*A*` $\cdots$ `*A` (`n` times) (A^n)

Standard Arithmetic Operation – Inner Products

Let $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ be column vectors. The *inner product* of $\mathbf{x}$ and $\mathbf{y}$ is calculated by

$$\mathbf{x}^{\mathrm{T}}\mathbf{y} = x_1y_1 + x_2y_2 + \cdots + x_my_m = \sum_{j=1}^{m} x_jy_j \in \mathbb{R}.$$

In MATLAB, simply type `x'*y`.

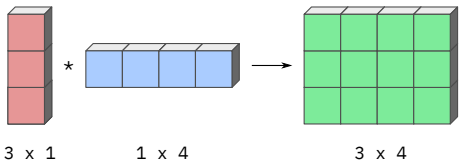


Standard Arithmetic Operation – Outer Products

Let $\mathbf{x} \in \mathbb{R}^m, \mathbf{y} \in \mathbb{R}^n$ be column vectors. The *outer product* of $\mathbf{x}$ and $\mathbf{y}$ is calculated by

$$\mathbf{x}\mathbf{y}^{\mathrm{T}} = \begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{bmatrix} \in \mathbb{R}^{m\times n}.$$

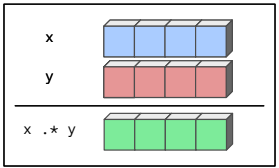
In MATLAB, simply type `x*y'`.



Elementwise Multiplication (`.*`)

- To multiply entries of two arrays of same size, element by element:

```
x .* y
```



Elementwise Division (`./`)

- To divide entries of an array by corresponding entries of another same-sized array:

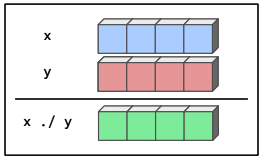
```
x ./ y
```

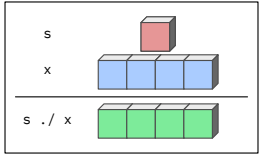
- To divide a number by multiple numbers (specified by entries of an array):

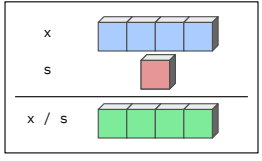
```
s ./ y
```

- To divide all entries of an array by a common number:

```
x / s
```







Elementwise Exponentiation (`.^`)

- To raise all entries of an array to (different) powers:

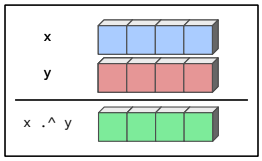
```
x .^ y
```

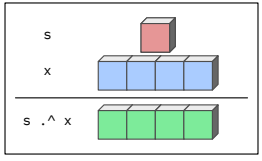
- To raise a number to multiple powers (specified by entries of an array):

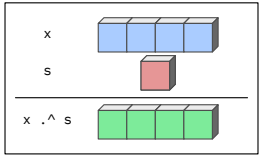
```
s .^ x
```

- To raise all entries of an array to a common power:

```
x .^ s
```



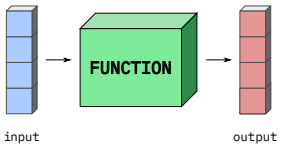




Mathematical Functions

- Built-in mathematical functions accept array inputs and return arrays of function evaluation, *e.g.*,

```
sqrt(A)
sin(A)
mod(A)
...
```



Colon Operator

Suppose `a < b`.

- To create an arithmetic progression from `a` to `b` (increment by 1):

```
a:b
```

The result is a row vector `[a, a+1, a+2, ..., a+m]`, where

$$m = \lfloor b-a \rfloor.$$

- To create an arithmetic progression from `a` to `b` with steps of size `d > 0`:

```
a:d:b
```

The result is a row vector `[a, a+d, a+2*d, ..., a+m*d]`, where

$$m = \lfloor (b-a)/d \rfloor.$$

LINSPACE and LOGSPACE

- To create a row vector of `n` numbers evenly spaced between `a` and `b`:

```
linspace(a, b, n)
```

The result is `[a, a+d, a+2*d, ..., b]`, where

```
d = (b-a)/(n-1).
```

- To create a row vector of `n` numbers that are logarithmically evenly spaced between 10^{a} and 10^{b}:

```
logspace(a, b, n)
```

The result is $[10^{a}, 10^{a+d}, 10^{a+2d}, \ldots, 10^{b}]$, where

```
d = (b-a)/(n-1).
```

ZEROS, ONES, and EYE

- To create an $(m \times n)$ zero matrix:

```
zeros(m, n)
```

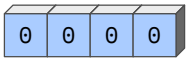
- To create an $(m \times n)$ matrix all whose entries are one:

```
ones(m, n)
```

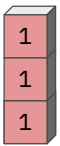
- To create the $(m \times m)$ identity matrix:

```
eye(m)
```

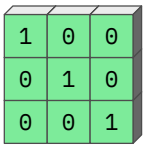
`zeros(1,4)`



`ones(3,1)`



`eye(3)`



Random Arrays

Each of the following generates an $(m \times n)$ array of random numbers:

- `rand(m,n)`: uniform random numbers in $(0, 1)$
- `randi(k,m,n)`: uniform random integers in $[1, k]$
- `randn(m,n)`: Gaussian random numbers with mean 0 and standard deviation 1

Random Arrays (Application)

To generate an $(m \times n)$ array of

- uniform random numbers in (a, b):

```
a + (b - a)*rand(m, n)
```

- uniform random integers in $[k_1, k_2]$:

```
randi([k1, k2], m, n)
```

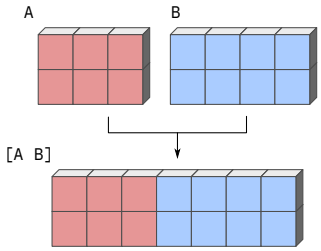
- Gaussian random numbers with mean μ and standard deviation σ:

```
mu + sig*randn(m, n)
```

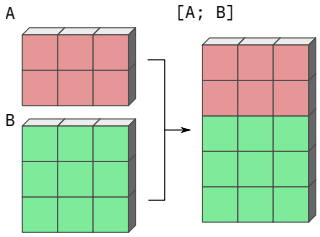
Concatenation

If two arrays `A` and `B` have *comparable* sizes, we can concatenate them.

- horizontally by `[A B]`

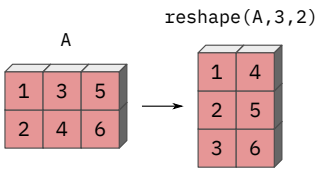


- vertically by `[A; B]`

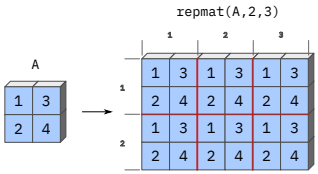


RESHAPE and REPMAT

- `reshape(A, m, n)` reshapes the array `A` into an $m \times n$ matrix whose elements are taken *columnwise* from `A`.



- `repmat(A, m, n)` replicates the array `A`, m times vertically and n times horizontally.



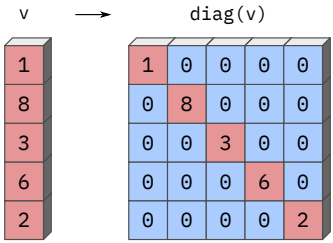
FLIP

- Type `help flip` on the Command Window and learn about `flip` function.
- Do the same with its two variants, `flipud` and `fliplr`

Creating Diagonal Matrices

- To create a diagonal matrix

$$\begin{bmatrix} v_1 & 0 & 0 & \cdots & 0 \\ 0 & v_2 & 0 & \cdots & 0 \\ 0 & 0 & v_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & v_n \end{bmatrix} :$$




```
diag(v)
```

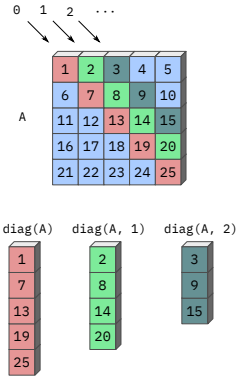
Note.

- `diag(v,k)` puts the elements of `v` on the `k`-th super-diagonal.
- `diag(v,-k)` puts the elements of `v` on the `k`-th sub-diagonal.

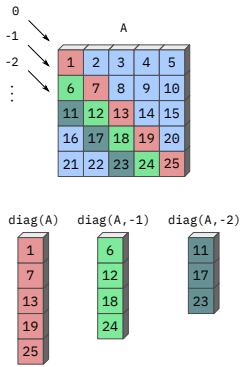
Extracting Diagonal Elements

Use `diag(A,k)` to extract the `k`-th diagonal of `A`. [1]

- `k > 0` for super-diagonals:



- `k < 0` for sub-diagonals:



[1] diag(A) short for diag(A,0) .

Using Vectors as Indices

To get/set multiple elements of an array at once, use vector indices.

- To grab 3rd, 4th, and 5th elements of `x`:

```
x(3:5)   % or x([3 4 5])
```

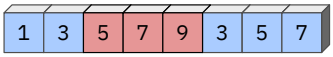
- To grab 3rd to 8th elements of `x`:

```
x(3:8)
x(3:end)
```

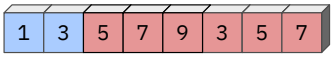
- To grab 3rd to 7th elements of `x`:

```
x(3:7)
x(3:end-1)
```

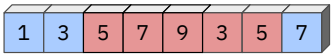
`x(3:5)`



`x(3:8) or x(3:end)`



`x(3:7) or x(3:end-1)`



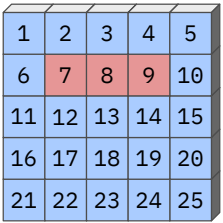
Using Vectors as Indices – Example

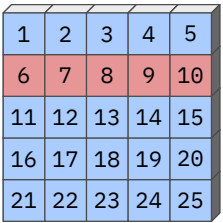
- To extract 2nd, 3rd, and 4th columns of the 2nd row of `A`:

```
A(2,2:4)   % or A(2,[2 3 4])
```

- To extract the entire 2nd row of `A`:

```
A(2,1:5)
A(2,1:end)
A(2,:)
```

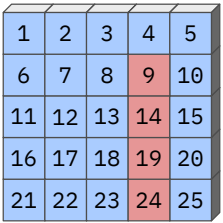




Using Vectors as Indices – Example

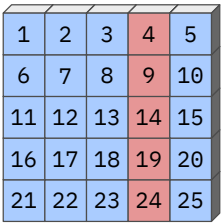
- To extract 2nd through 5th elements of the 4th column of `A`:

```
A([2 3 4 5],4)
A(2:5,4)
A(2:end,4)
```



- To extract the entire 4th column of `A`:

```
A(1:5,4)
A(1:end,4)
A(:,4)
```



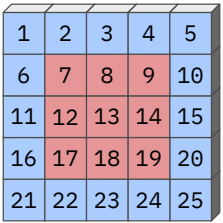
Using Vectors as Indices – Example

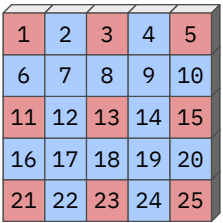
- To grab the *interior block* of `A`:

```
A(2:4,2:4)
A(2:end-1,2:end-1)
```

- To extract every other elements on every other rows as shown:

```
A(1:2:5,1:2:5)
A(1:2:end,1:2:end)
```





More on Arrays

Arithmetic Progressions

Question

Create the following *periodic* arithmetic progressions using ONE MATLAB statement.

$$(1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0).$$

```
m = 5;
n = 15;
mod([1:n], m)
```

Exercise: Arithmetic Progressions

Question

Create each of the following *row* vectors using ONE MATLAB statement.

- $\mathbf{v} = (1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0)$
- $\mathbf{w} = (1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)$

Geometric and Other Progressions

Question

Create each of the following *column* vectors using ONE MATLAB statement.

- $\mathbf{v} = (1, 2, 4, 8, \ldots, 1024)^{\mathrm{T}}$
- $\mathbf{w} = (1, 4, 9, 16, \ldots, 100)^{\mathrm{T}}$

Using the colon operator:

```
v = ( 2.^[0:10] )'
w = ( [1:10].^2 )'
```

Using the `linspace` function:

```
v = ( 2.^linspace(0, 10, 11) )'
w = ( linspace(1, 10, 10).^2 )'
```

Function Evaluation

Recall that mathematical functions such as `sin`, `sind`, `log`, `exp` accept array inputs and return arrays of function evaluation.

Question

Create each of the the following *row* vectors using ONE MATLAB statement.

- $\mathbf{u} = (1!, 2!, 3!, \ldots, n!)$
- $\mathbf{v} = (\sin 0^\circ, \sin 30^\circ, \sin 60^\circ, \ldots, \sin 180^\circ)$
- $\mathbf{w} = (e^1, e^4, e^9, \ldots, e^{64})$

```
v = sind(0:30:180)
w = exp([1:8].^2)
```

Matrices with Patterns

Question

Generate each of the following matrices using ONE MATLAB statement.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 1^2 & 1^3 & \cdots & 1^{10} \\ 2 & 2^2 & 2^3 & \cdots & 2^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 10 & 10^2 & 10^3 & \cdots & 10^{10} \end{bmatrix}.$$

```
A = reshape(1:16, 4, 4)'
B = ((1:10)').^(1:10)
```

Matrices with Patterns

Question

Suppose `n` is already stored in MATLAB. Generate each of the following matrices using ONE MATLAB statement. All the elements not shown are 0's.

$$C = \begin{bmatrix} 2 & & & & \\ & 4 & & & \\ & & 6 & & \\ & & & \ddots & \\ & & & & 2n \end{bmatrix}, D = \begin{bmatrix} \cos 1 & -3 & & & & & \\ & \cos 2 & -3 & & & & \\ & & \cos 3 & -3 & & & \\ & & & \ddots & \ddots & & \\ & & & & \cos(n-1) & -3 & \\ & & & & & \cos n \end{bmatrix}.$$

```
C = diag(2:2:2*n)
D = diag(cos(1:n)) - 3*diag(ones(n-1,1), 1)
```

Data Manipulation Functions

There are a number of MATLAB functions with *spreadsheet functionalities* that are suitable for data manipulation.

- `max` and `min`
- `sum` and `prod`
- `cumsum` and `cumprod` (cumulative sum and product)
- `diff`
- `mean`, `std`, and `var` (simple statistics)
- `sort`

Example 1: Finding the Maximum Value of a Vector

Question

Write a program to find the maximum value of a vector.

- **With loops:**

```
% input: x
% output: m     % DON'T USE max FOR THE VARIABLE NAME
m = x(1);       % CODE ABORTS IF THE VECTOR IS EMPTY
for r = 2:length(x)
    if m < x(r)
        m = x(r);
    end
end
```

- **Vectorized code:**

```
m = max(x)
```

Example 1 (cont')

Question

Now modify the previous program to find both the maximum value of a vector and the corresponding index.

- **With loops:**

```
% input: x
% output: m, index_m
m = x(1);
index_m = 1;
for r = 2:length(x)
    if m < x(r)
        m = x(r);
        index_m = r;
    end
end
```

- **Vectorized code:**

```
[m, index_m] = max(x)
```

Example 2: Summing Elements in a Vector

Question

Sum all elements in a vector.

- **With loops:**

```
% input: x
% output: s     % DON'T USE sum FOR THE VARIABLE NAME
s = 0;          % s begins before the first iteration
for el = 1:length(x)
    s = s + x(el);
end
```

- **Vectorized code:**

```
s = sum(x)
```

FIND Function

Basic Usage of FIND

Let v be an array of numbers (can be a vector or a matrix). Then

```
find(<condition>)
```

returns the (linear) indices of v satisfying <condition>.

- **Some examples of <condition>:**
  - v > k or v >= k
  - v < k or v <= k
  - v == k or v ~= k
- **To combine more than two conditions:**
  - & **(and)**
  - | **(or)**

Example 3: Comparing Elements in Vectors

Question

Compare two real vectors of the same length, say `x` and `y`, elementwise and determine how many elements of the former are larger than the latter.

- **With loops:**

```
% input: x, y
% output: nr_gt
nr_gt = 0;
for k = 1:length(x)
    if x(k) > y(k)
        nr_gt = nr_gt + 1;
    end
end
```

- **Vectorized code:**

```
nr_gt = length(find(x > y))
```

CPU Time

`cputime` reads total CPU time used by MATLAB from the time it was started.

- Single measurement:

```
ct = cputime;   % total cputime as of now
    <statements>
t = cputime - ct;
```

- Average CPU time:

```
ct = cputime;   % total cputime as of now
for i = 1:nr_reps
    <statements>
end
t_avg = (cputime - ct)/nr_reps;
```

Elapsed Time

At the execution of `tic`, MATLAB records the internal time (in seconds); at `toc` command, MATLAB displays the elapsed time.

- Single measurement:

```
tic     % starts a stopwatch timer
    <statements>
toc     % reads the elapsed time from tic
```

- Average elapse time:

```
tic    % starts a stopwatch timer
for i = 1:nr_reps
    <statements>
end
t_avg = toc/nr_reps;
```

What Do You Think It Does?

Below is a modified version of an example code from MATLAB's Help documentation for `tic`. What do you think it's doing?

```
REPS = 1000; minTime = Inf; nSum = 10;
tic;
for i  = 1:REPS
    tStart = tic;
    s = 0;
    for j = 1:nsum
        s = s + besselj(j,REPS);
    end
    tElpased = toc(tStart);
    minTime = min(tElapsed, minTime);
end
t_avg = toc/REPS;
```

Example 4: Timing Elementwise Operations

Question

Generate a $10^7 \times 1$ random vector and measure the internal time and CPU time when computing elementwise squares.

```
n = 1e7;
x = rand(n, 1);
t = cputime;
x1 = x.^2;
time1 = cputime - t;

tic
x2 = x.^2;
time2 = toc();
disp([time1, time2])
```

Pythagorean Triples

Question

Given $n \in \mathbb{N}$, find all triples $(a, b, c)\mathbb{N}^3$, with $a, b \leqslant n$, satisfying

$$a^2 + b^2 = c^2.$$

Notation.

- $\mathbb{N}$: the set of all natural numbers, $1, 2, 3, \ldots$.
- $\mathbb{N}[1, n] = \{1, 2, \ldots, n\}$.
- $\mathbb{N}^3 = \{(a, b, c) \mid a, b, c \in \mathbb{N}\}$.

Pythagorean Triples – Solution Using Loops

```
% input: n
% output: M
iM = 0;
M = [];
for a = 1:n
    for b = 1:n
        c = sqrt(a^2 + b^2);
        if mod(c, 1) == 0
            iM = iM + 1;
            M(iM, :) = [a, b, c];
        end
    end
end
```

Pythagorean Triples – Solution Without Loops

```
% input: x
% output: M
A = repmat([1:n], n, 1);
B = repmat([1:n]', 1, n);
C = sqrt(A.^2 + B.^2);
M = [A(:), B(:), C(:)];
lM = ( mod(M(:, 3), 1) ~= 0 );
M(lM, :) = [];
```

Birthday Problem

Question

In a group of n randomly chosen people, what is the probability that everyone has a different birthday?

1. Find this probability by hand.
2. Let $n = 30$. Write a script that generates a group of n people randomly and determines if there are any matches.
3. Modify the script above to run a number of simulations and numerically calculate the sought-after probability. Try $1000, 10000$, and 100000 simulations. Compare the result with the analytical calculation done in 1.

Birthday Problem (Hints)

- For simplicity, ignore leap years.
- Create a random (column) vector whose elements represent birthdays of individuals (denoted by integers between 1 and 365).
- Line up the birthdays in order and take the difference of successive pairs. What does the resulting vector tell you?
- For 3, to run simulation multiple times, consider creating a random matrix whose rows represent birthdays of individuals and the columns correspond to different simulations.

Graphics in MATLAB

Anonymous Functions

Mathematical functions such as

$$f_1(x) = \cos x \sin\big(\cos(\tan x)\big),$$
$$f_2(\theta) = (\cos 3\theta + 2\cos 2\theta)^2,$$
$$f_3(x, y) = \frac{\sin(x + y)}{1 + x^2 + y^2},$$

can be defined in MATLAB using *anonymous functions*:

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
f2 = @(th) ( cos(3*th) + 2*cos(2*th) ).^2;
f3 = @(x, y) sin(x + y)./(1 + x.^2 + y.^2);
```

Anonymous Functions – Syntax

Take a closer look at one of them.

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
```

- `f1` : the function name or the *function handle*
- `@` : marks the beginning of an anonymous function
- `(x)` : denotes the function (input) argument
- `cos(x).*sin(cos(tan(x)))` : MATLAB expression defining $f_1(x)$

Examples

Expressions in function definitions can get very complicated. For example,

```
h1 = @(x) [2*x, sin(x)];
h2 = @(x) [2*x, sin(x); 5*x, cos(x); 10*x, tan(x)];
r = @(a,b,m,n) a + (b-a)*rand(m,n);
```

Exercise: Different Ways of Defining a Function

The function

$$f_4(\theta; c_1, c_2, k_1, k_2) = (c_1 \cos k_1\theta + c_2 \cos k_2\theta)^2$$

can be defined in two different ways:

```
f4s = @(th,c1,c2,k1,k2) c1*cos(k1*th) + c2*cos(k2*th)
f4v = @(th,c,k) c(1)*cos(k(1)*th) + c(2)*cos(k(2)*th)
```

Question

Use `f4s` and `f4v` to define yet another anonymous functions for

- $g(\theta) = 3\cos(2\theta) - 2\cos(3\theta)$
- $h(\theta) = 3\cos(\theta/7) + \cos(\theta)$

Exercise: Understanding Anonymous Functions

Type in the following statements in MATLAB:

```
f1 = @(x) cos(x).*sin(cos(tan(x)));
f2 = @(th) ( cos(3*th) + 2*cos(2*th) ).^2;
x1 = 5; y1 = f1(x1)
x2 = [5:-2:1]; y2 = f1(x2)
TH = diag(0:pi/2:2*pi); R = f2(TH)
```

Question

1. What are the types of the input and output variables?
   - `x1` and `y1`
   - `x2` and `y2`
   - `TH` and `R`
2. Which of the three outputs will be affected if elementwise operations were not used in the definition of `f1` and `f2`?

The PLOT Function

To draw a curve in MATLAB:

- Construct a pair of n-vectors x and y corresponding to the set of data points $\{(x_i, y_i) \mid i = 1, 2, \ldots, n\}$ which are to appear on the curve in that order.
- Then type `plot(x, y)` .

For example:

```
x = linspace(0, 2*pi, 101);
y = sin(x);
plot(x, y)                          % or simply plot(x, sin(x))
```

or

```
f = @(x) 1 + sin(2*x) + cos(4*x);       % anonymous function
x = linspace(0, 2*pi, 101);
plot(x, f(x))
```

Example: Wiggly Curve

First, run the following script.

```
1 f1 = @(x) cos(x).*sin(cos(tan(x)));
2 x = 2*pi*[0:.0001:1];   % or  x = linspace(0, 2*pi, 10001);
3 plot(x, f1(x))
4 shg
```

Play Around!

Observe what happens after applying the following modifications one by one.

- Change line 3 into `plot(x, f1(x), 'r')`.
- Change line 3 into `plot(x, f1(x), 'r--')`.
- After line 3, add `axis equal, axis tight`.
- Then add `text(4.6, -0.3, 'very wiggly')`.
- Then add
  `xlabel('x axis'), ylabel('y axis'), title('A wiggly curve')`.

Note: Line Properties

- To specify line properties such as colors, markers, and styles:

```
plot(x, y, '--')                    % dashed line
plot(x, y, 'g:')                    % dotted line in green
plot(x, y, '.', 'MarkerSize', 3)    % adjust marker size
plot(x, y, 'b-', 'LineWidth', 5)    % adjust line width
```

Colors

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Markers

.	point
o	circle
x	x-mark
+	plus
*	star
s	square
d	diamond

Line Styles

-	solid
:	dotted
-.	dashdot
--	dashed

Note: Labels and Saving

- To label the axes and the entire plot, add the following after `plot` statement:

```
xlabel('x axis')
ylabel('y axis')
title('my awesome graph')
```

- Save figures using `print` function. Multiple formats are supported.

```
print -dpdf 'wiggly'
% or print('-dpdf', 'wiggly')                    [pdf]

print -djpeg 'wiggly'
% or print('-djpeg', 'wiggly')                  [jpeg]

print -deps 'wiggly'
% or print('-deps', 'wiggly')                    [eps]
```

Note: Drawing Multiple Figures

- To plot multiple curves:

```
plot(x1, y1, x2, y2, x3, y3, ...)
```

- To create a legend, add

```
legend('first graph', 'second graph', 'third graph', ...)
```

Note: Miscellaneous Commands

- `shg`: (show graph) to bring Figure Window to the front
- `figure`: to open a new blank figure window
- `clf`: (clear figure) to clear previously drawn figures
- `axis equal`: to put axes in equal scaling
- `axis tight`: to remove margins around graphs
- `axis image`: same as `axis equal` and `axis tight`
- `grid on`: to put light gray grid lines

Exercise

Question

Do the following:

- Define $f(x) = x^3 + x$ as an anonymous function.
- Find f' and f'' and define them as anonymous functions.
- Plot all three functions in one figure in the interval $[-1, 1]$.
- Include labels and title in your plot.
- Add legend to the graph.
- Save the graph as a pdf file.

Multiple Figures – Stacking

To draw multiple curves in one plot window as in Figure 1:

- One liner:

```
plot(x1, y1, x2, y2, x3, y3)
```

- Or, add curves one at a time using `hold` command.

```
plot(x1, y1)
hold on
plot(x2, y2)
plot(x3, y3)
```

  - `hold on`: holds the current plot for further overlaying
  - `hold off`: turns the *hold* off

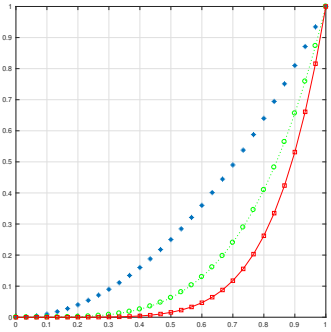


Figure 1: Multiple curves in one plot window

Multiple Figures – Subplots

To plot multiple curves separately and arrange them as in Figure 2:

```
subplot(1,3,1)
plot(x1, y1)
subplot(1,3,2)
plot(x2, y2)
subplot(1,3,3)
plot(x3, y3)
```

`subplot(m,n,p)`:

- `m`, `n`: determine grid dimensions
- `p`: determines grid is to be used

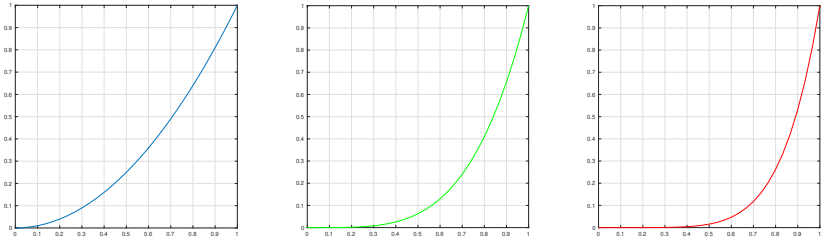


Figure 2: Multiple plots in 1×3 grids

Exercise: Multiple Figures

Do It Yourself

Generate Figures 1 and 2.

- Common: Generating sample points

```
x = linspace(0, 1, 101);
y1 = x.^2; y2 = x.^4; y3 = x.^6;
```

- Figure 1:

```
hold off
plot(x, y1, '*')
hold on
plot(x, y2, 'g:o')
plot(x, y3, 'r-s')
```

- Figure 2:

```
subplot(1, 3, 1)
plot(x, y1)
subplot(1, 3, 2)
plot(x, y2, 'g')
subplot(1, 3, 3)
plot(x, y3, 'r')
```

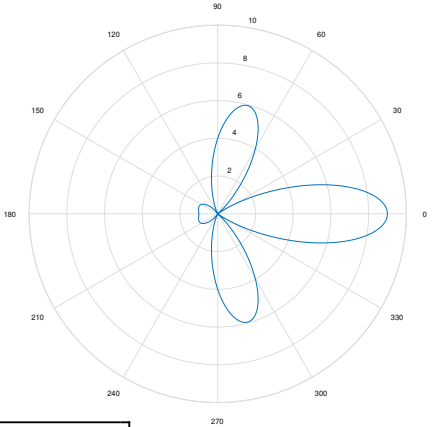
The `POLAR` Function

To draw the polar curve $r = f(\theta)$, for $\theta \in [a, b]$:

- Grab n sample points $\{(\theta_i, r_i) \mid r_i = f(\theta_i),\ 1 \leqslant i \leqslant n\}$ on the curve and form vectors `th` and `r`.
- Then type `polar(th, r)` .
- For example, to plot

$$r = f_2(\theta) = (\cos 3\theta + 2\cos 2\theta)^2,$$

for $\theta \in [0, 2\pi]$:




```
th = linspace(0, 2*pi, 361);
f2 = @(th) (cos(3*th) + 2*cos(2*th)).^2;
polar(th, f2(th));
```

Exercise: Drawing Polar Curves

Question

1. Draw the graph of two-petal leaf given by

$$r = f(\theta) = 1 + \sin(2\theta)\,, \quad \theta \in [0, 2\pi]\,.$$

2. Draw the graphs of

$$r = f(\theta - \pi/4)\,, \quad r = f(\theta - \pi/2)\,, \quad r = f(\theta - 3\pi/4)$$

on the same plotting window.

3. Does your figure make sense?

Curves and the `PLOT3` Function

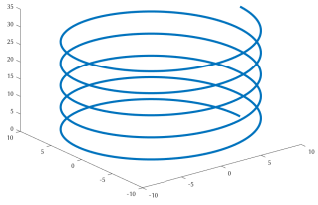
Curves in $\mathbb{R}^3$ are plotted in an analogous fashion.

- Grab n sample points $\{(x_i, y_i, z_i) \mid i = 1, 2, \ldots, n\}$ on the curve and form vectors `x`, `y`, and `z`.
- Then type `plot3(x, y, z)` .
- For example, to plot the helix given by the parametrized equation

$$\mathbf{r}(t) = \langle 10\cos(t), 10\sin(t), t \rangle,$$

for $t \in [0, 10\pi]$:

```
t = linspace(0, 10*pi, 1000);
plot3(10*cos(t), 10*sin(t), t);
```



Exercise: Corkscrew

Question

Modify the code to generate a corkscrew by putting the helix outside of an upside down cone.

Hint: Use $\mathbf{r}(t) = \langle t\cos(t), t\sin(t), t\rangle$.

Surfaces and the `SURF` Function

To plot the surface of $z = f(x, y)$ on $R = [a, b] \times [c, d]$:

- Collect samples points on the intervals $[a, b]$ and $[c, d]$ and form vectors `x` and `y`.
- Based on `x` and `y`, generate grid points $\{(x_i, y_j) \mid i = 1, 2, \ldots, m, j = 1, 2, \ldots, n\}$ on the domain R and separate coordinates into matrices `X` and `Y` using `meshgrid`.
- Type `surf(X, Y, f(X,Y))`

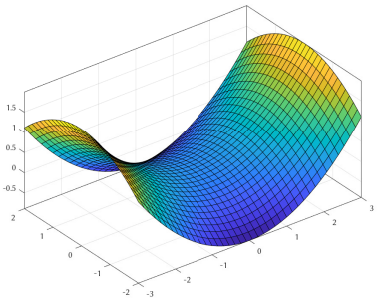
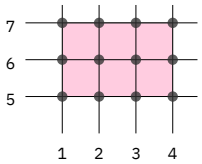
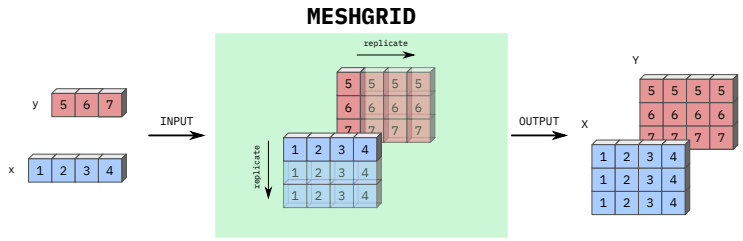


Figure 3: Graph of $z = \frac{2}{9}(x^2 - y^2)$ on $[-3, 3] \times [-2, 2]$

Note: How MESHGRID Work

```
>> x = [1 2 3 4]; y = [5 6 7];
>> [X, Y] = meshgrid(x,y)
X =
     1     2     3     4
     1     2     3     4
     1     2     3     4
Y =
     5     5     5     5
     6     6     6     6
     7     7     7     7
```





Example: Saddle

Question

Plot the saddle parametrized by

$$\frac{z}{c} = \frac{x^2}{a^2} - \frac{y^2}{b^2}$$

for your choice of a, b, and c.

```
x = linspace(-3, 3, 13);
y = linspace(-2, 2, 9);
[X, Y] = meshgrid(x, y);
a = 1.5; b = 1.5; c = .5;
g2 = @(x,y) c*( x.^2 /a^2 - y.^2 /b^2);
surf(X, Y, g2(X,Y))
axis equal, box on
```

Figure 3 was generated using this code.

Example: Oblate Spheroid

The figure for Problem 5 of Homework 1 was generated by the following code.[2]

```
a = 1; b = 1.35; c = 1;
nr_th = 41; nr_ph = 31;
x = @(th, ph) a*cos(th).*sin(ph);
y = @(th, ph) b*sin(th).*sin(ph);
z = @(th, ph) c*cos(ph);
th = linspace(0, 2*pi, nr_th);
ph = linspace(0, pi, nr_ph);
[T, P] = meshgrid(th, ph);
surf(x(T,P), y(T,P), z(T,P))
colormap(winter)
axis equal, axis off, box off
```

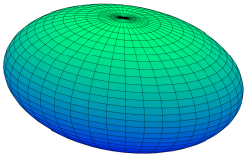


Figure 4: Oblate spheroid.

Function M-File

Basics

- A *function* is a piece of code which
  - performs a specific task;
  - has specific input and output arguments;
  - is encapsulated to be independent of the rest of the program.
- In MATLAB, functions are defined in .m files just as scripts.
- The name of the file and that of the function must coincide.

Why Functions?

Two Important Principles

- A piece of code should only occur once in a program. If it tries to appear more times, put it into its own function.
- Each function should do precisely **one** task well. If it has to carry out a number of tasks, separate them into their own functions.

Benefits of writing functions:

- elevated reasoning by hiding details
- facilitates top-down design
- ease of software management

Writing a Function

A function m-file must be written in a specific manner.

- When there is no input/output argument

```
function myfun()
  ....
end    % <-- optional
```

- Where there are multiple input and output arguments

```
function [out1, out2, out3] = myfun(in1, in2, in3, in4)
  ....
end    % <-- optional
```

Calling a Function

- If the function m-file `myfun.m` is saved in your current working directory[3], you can use it as you would use any other built-in functions:

```
% when no input/output argument is required
myfun
```

or

```
% multiple inputs/outputs
[out1, out2, out3] = myfun(in1, in2, in3, in4)
```

- When not all output arguments are needed:

```
out1 = myfun(in1, in2, in3, in4)          % only 1st output
[~, ~, out3] = myfun(in1, in2, in3, in4) % only 3rd output
```

Note that tilde (~) is used as a placeholder.

[3] The `path` function gives more flexibility in this regard.

Practical Matters: Specification

- The comments written below `function` header statement

```
function ... = myfun( ... )
% MYFUN: this awesome function calculates ...
  ....
end
```

  can be easily accessed from the command line using `help myfun`.

- Use this feature to write the function specification such as its purpose, usage, and syntax.
- Write a clear, complete, yet concise specification.

Properties of Function M-Files

- The variable names in a function are completely isolated from the calling code. Variables, other than outputs, used inside a function are unknown outside of the function; they are *local variables*.
- The input arguments can be modified within the body of the function; no change is made on any variables in the calling code. (“ *pass by value* ” as opposed to “ *pass by reference* ”)
- Unlike script m-files, you **CANNOT** execute a function which has input arguments using the RUN icon.

Example: Understanding Local Variables

- The function on the right finds the maximum value of a vector and an index of the maximal element.
- Run the following on the Command Window. Pay attention to `m`.

```
>> m = 33;
>> x = [1 9 2 8 3 7];
>> [M, iM] = mymax(x)
>> disp(m)
```

```
function [m, el] = mymax(x)
    if isempty(x)
        el = [];
        m = [];
        return
    end
    el = 1;
    m = x(el);
    for i = 2:length(x)
        if m < x(i)
            el = i;
            m = x(el);
        end
    end
end
```

Example: Understanding Pass-By-Value

Consider the function

$$f(x) = \sin|x| \, \frac{e^{-|x|}}{1+|x|^2} + \frac{\ln(1+|x|)}{1+2|x|}$$

written as a MATLAB function:

```
function y = funky(x)
    x = abs(x);  % x is redefined to be abs(x)
    y = sin(x).*exp(-x)./(1 + x.^2)...
        + log(1 + x)./(1 + 2*x);
end
```

Confirm that the function does not affect `x` in the calling routine:

```
>> x = [-3:3]';
>> y = funky(x);
>> disp([x, y])
```

Quiz

Question

Script.

```
x = 2;
x = myfun(x);
y = 2*x
```

Function.

```
function y = myfun(x)
    x = 2*x;
    y = 2*x;
end
```

What is the output when the script on the left is run?

1. `y = 2`
2. `y = 4`
3. `y = 8`
4. `y = 16`

Description of Problem

Problem

Given m and θ (in degrees), draw m equilateral triangles so that for any two successive triangles,

- the vertices of the smaller triangle are lying on the sides of the larger;
- the angle between the two triangles is θ (degrees)

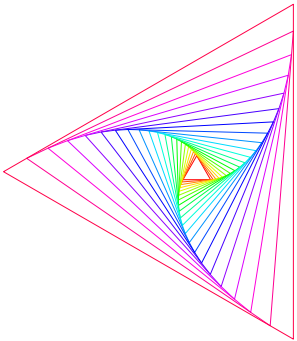


Figure 5: A spiral triangle with $m = 21$ and $\theta = 4.5^\circ$.

Generation and Transformation of Polygons

Let $(x_j, y_j), j = 1, 2, \ldots, n$ be the coordinates of vertices of an n-gon.

- To plot the polygon:

```
x = [x1 x2 ... xn x1]; % note: x1 and y1 are repeated at
y = [y1 y2 ... yn y1]; % end to enclose the polygon.
plot(x, y)
```

- To plot the polygon obtained by scaling the original by a factor of `s`:

```
plot(s*x, s*y)
```

- To plot the polygon obtained by rotating the original by an angle `theta` (in degrees) about the origin:

```
V = [x; y];                          % all vertices
R = [cosd(theta) -sind(theta);       % 2-D rotation matrix
     sind(theta)  cosd(theta)];
Vnew = R*V;                          % rotated vertices
plot(Vnew(1,:), Vnew(2,:))
```

Special Case: Inscribed Regular Polygons

- The vertices of the *regular* n-gon inscribed in the unit circle can be found easily using trigonometry, *e.g.*,

$$(x_j, y_j) = \left(\cos \frac{2\pi(j-1)}{n}, \sin \frac{2\pi(j-1)}{n} \right), \quad j = 1, \ldots, n.$$

- Thus we can plot it by

```
theta = linspace(0, 360, n+1);
V = [cosd(theta);      % 2-by-(n+1) matrix whose cols are
     sind(theta)];     % coordinates of the vertices
plot(V(1,:), V(2,:)), axis equal
```

- To stretch and rotate the polygon:

```
Vnew = s*R*V;
plot(Vnew(1,:), Vnew(2,:)), axis equal
```

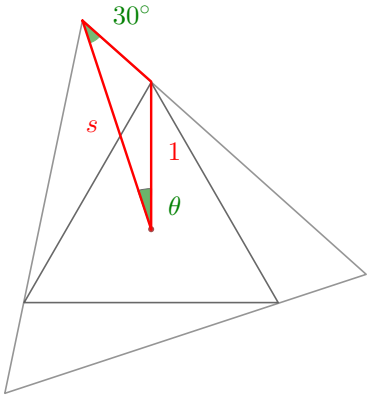
Scale to Spiral

To create the desired spiraling effect, the scaling factor must be calculated carefully.

- Useful:

$$\frac{\sin\alpha}{a} = \frac{\sin\beta}{b} = \frac{\sin\gamma}{c}$$

- Compute the scaling factor s:



Put All Together

```
m = 21; d_angle = 4.5;
th = linspace(0, 360, 4) + 90;
V = [cosd(th);
     sind(th)];
C = colormap(hsv(m));
s = sind(150 - abs(d_angle))/sind(30);
R = [cosd(d_angle) -sind(d_angle);
     sind(d_angle) cosd(d_angle)];
hold off
for i = 1:m
    if i > 1
        V = s*R*V;
    end
    plot(V(1,:), V(2,:), 'Color', C(i,:))
    hold on
end
set(gcf, 'Color', 'w')
axis equal, axis off
```

Exercise: Script to Function

Question

Modify the script so that it can create a spiral n-gon consisting of m regular n-gons successively rotated by θ degrees. Then turn the script into a function m-file `spiralgon.m`.

```
function V = spiralgon(n, m, d_angle)
% SPRIALGON plots spiraling regular n-gons
% input:   n = the number of vertices
%          m = the number of regular n-gons
%          d_angle = the degree angle between successive n-gons
%          (can be positive or negative)
% output:  V = the vertices of the outermost n-gon

% Fill in the rest.
....
```

Local Functions

There are “general purpose” functions which may be used by a number of other functions. In this case, we put it into a separate file. However, many functions are quite specific and will only be used in one program. In such a case, we keep them in whatever file calls them.

- The first function is called the *primary* function and any function which follows the primary function is called a *local function* or a *subfunction*.

```
function <primary function>
  ....
end
function <local function #1>
  ....
end
<any number of following local functions>
```

- The primary function interact with local functions through the input and output arguments.
- Only the primary function is accessible from outside the file.

Defining/Evaluating Mathematical Functions in MATLAB

- using an anonymous function
- using a local function
- using a primary funciton
- using a nested function
- putting it *in situ*, i.e., write it out completely in place – not recommended

See `time_anon.m`.

Passing Function to Another Function

- A function can be used as an input argument to another function.
- The function must be stored as a function-handle variable in order to be passed as an argument.
- This is done by prepending " `@` " to the function names. For instance, try

```
>> class(mymax)    % WRONG
>> class(@mymax)   % CORRECT
```

When a Function Is Called

Below is what happens internally when a function is called:

- MATLAB creates a new, local workspace for the function.
- MATLAB evaluates each of the input arguments in the calling statements (if there are any). These values are then assigned to the input arguments of the function. These are now local variables in the new workspace.
- MATLAB executes the body of the function. If there are output arguments, they are assigned values before the execution is complete.
- MATLAB deletes the local workspace, saving only the values in the output arguments. These values are then assigned to the output arguments in the calling statement.
- The code continues following the calling statement.