FINAL PARTICIPATION PORTFOLIO

Coding Notebook

Razaq Aribidesi

University of Pennsylvania

Table of Contents

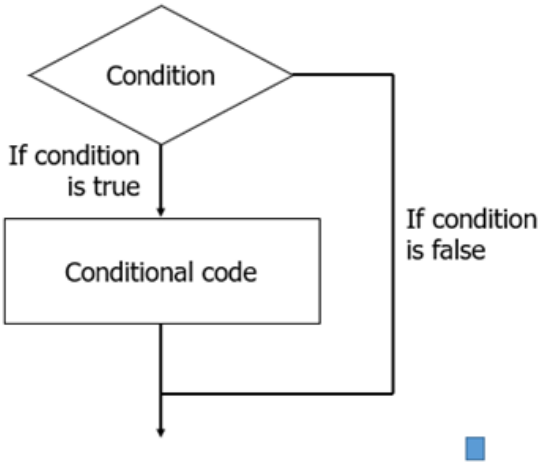
Origins of MATLAB 8
MATLAB Interface 8
Working at the command line 9
Using LiveScripts 9
General notes on MATLAB syntax and peculiarites 10
Statements and Expressions 10
Naming conventions for variables 11
Comments, suppression, and separation 11
Procedural programming order 11
Errors and Warnings 12
Using the Editor 12
Functions 12
Scripts 13
Clearing the workspace and figures 13
Working directory and paths 14
Interrupting code execution 14
MATLAB documentation and help 14
Built-in MATLAB functions 15
Basics of vectors: 1- D arrays 16
Creating vectors with slice notation 16
Creating vectors with `linspace()` and `logspace()` 16
Other vectors 17
Accessing vector indices 17
Replacing indices and adding indices to vectors 18
Removing vector indices 20
Concatenating vectors 21
Operations on vectors 21
TOPIC 2: Introduction to arrays: vectors, matrices, and more 22
Revisiting vectors 23
Example 23
Column vectors 24
Determining the number of entries in a vector 25
Precedence 26
Arrays 28
Dimensions of an array 29
Matrices 30
Constructing matrices 30
Accessing array indices 33
Overwriting indices 35
Array mathematics 36
Addition and Subtraction 37
Multiplication 37
Division 40
Application: solving a set of linear equations 43
One approach: substitution 46
Another approach: matrix representation 47
More general representation 48

TOPIC 3: Systems of equations, functions, and more....49
Array mathematics (continued) 49
Addition and Subtraction 49
Multiplication....49
Division....53
Application: solving a set of linear equations 55
One approach: substitution 58
Another approach: matrix representation....59
More general representation 60
Example application: fluid flow in a Y-channel 61
Functions....64
Functions vs. scripts....65
Examples of function declarations....65
Function rules....66
Function headers....67
Function naming conventions....68
Example 68
Example: systems of nonlinear systems of equations....68
TOPIC 4: Loops and logic 70
Introductory remarks 70
Example - Fibonacci sequence 71
FOR loops 72
Anatomy of a `for` loop 73
Additional (pecular) notes....76
Nested `for` loops 78
Example - back to the Fibonacci sequence....78
WHILE loops 79
Relational operators 79

Relational operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
~=	Not equal to
>	Larger than
>=	Larger than or equal to

....79
Logical operators....80
Examples....80
Anatomy of a `while` loop 81
Example: simulating the roll of two dice 82
Example: back again to the Fibonacci sequence....82
Vector preallocation....83
For loops vs. vectorization....85
TOPIC 5: Conditionals and Boolean Operators....88

Loops (continued): for and while loops....88
Vector preallocation....88
For loops vs. vectorization....91
Recall: while loops iterate as long as a logical condition is true....93
Conditional statements....94



....94
Anatomy of a simple `if` statement....94
Nested `if` statements....95
Anatomy of an `if-elseif-else` statement....96
Boolean operators....98
Examples: scalars....100
Examples: element-wise on vectors....100
Precedence revisited....101
Boolean operators and conditional statements....104
Logical statements and `while` loops....105
TOPIC 6: Building solutions to problems, part 1....106
Recap of topics covered so far:....106
Using scientific computing to approach problems....110
PRO: a roadmap for building computational solutions....110
Troubleshooting and debugging techniques....111
Getting started with building computational solutions to problems....113
Practice problem 1: modeling the motion of a projectile....113
Practice problem 2: numerical methods and root-finding....118
Visual depiction of the Newton-Raphson algorithm....118
Derivation of the Newton-Raphson method....120
Question 1....120
Question 2....125
TOPIC 7: Building solutions to problems, part 2....126
Tinkering with fundamental MATLAB operations....126
1) Functions....126
2) Conditional statements....127
3) `for` loops....128
4) `while` loops....128
Practice problem 2: numerical methods and root-finding....129
Visual depiction of the Newton-Raphson algorithm....129
Derivation of the Newton-Raphson method....131
Question 1....131

TOPIC 8: Revisiting arrays, pt. 1: useful math and logical arrays 133
Revisiting arrays: useful structures for storing and processing numerical information 133
Reminder: Vector indexing 133
Matrix (and array) indexing 134
Adding, subtracting, multiplying and dividing arrays and their elements 135
Other useful mathematical operations on arrays 136
Useful built-in arrays 139
Logical Arrays 140
Logical numbers 140

Relational operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
~=	Not equal to
>	Larger than
>=	Larger than or equal to

..... 141
Logical arrays: vectors, matrices, etc. with logical numbers 141
Practice Problem: tic tac toe 143
TOPIC 9: Revisiting arrays, pt. 2: logical arrays and more 144
Logical Arrays (cont'd) 144
Indexing with logical arrays 145
Logical array indexing rules 146
Examples with vectors 146
Examples with matrices 147
Mathematical operations using logical arrays 149
Built-in logical functions 151
Changing the shape of arrays 153
Rotating and flipping arrays 153
Reshaping arrays 155
TOPIC 10: Revisiting arrays, pt. 3: improving computational efficiency 155
Logical arrays 155
Reminder: logical indexing and reshaping arrays 155
Logical arrays vs. `if-else` statements 162
Always preallocate arrays 166
Sparse arrays 168
TOPIC 11: Revisiting arrays, pt. 4; function handles 171
Practice Problem: process engineering in a medical environment 171
Increasing computation efficiency with arrays (continued from last time): 179
Always preallocate arrays 179
Sparse arrays 181
Revisiting functions 184
Function handles 186
Example: force on a spring 188

TOPIC 12: Function handles and recursion 190
Reminder: function handles....190
Practice problem: a root-finding numerical solver 191
Anonymous functions....193
Recursion 196
Example: Revisiting the fibonacci sequence....197
Example: calculate the factorial of a number 198
TOPIC 13: Plotting in MATLAB 199
Plotting in 2D 202
`plot()` - the "workhorse" plotting function....202
Overlaying plots....205
Specifying plot attributes inline....208
Colors 209
Basic axis formatting 210
Important note regarding LiveScript files....216
Plots as objects 217
Getting axes or figure properties....219
Setting axes and figure properties....222
Graphics handles 224
Linear vs log axes 237
Error bars 245
Additional 2D plotting functions....251
Closing graphics objects -- SURPRISINGLY USEFUL....267
Multiple axes in a single figure window 267
Plotting in 3D 272
`plot3` 272
Practice Problem: adventure summary 278
TOPIC 14: Data types in MATLAB 286
Practice problem: Legendre polynomials and recursion 286
Recall: different data types....289
Data types (classes) in MATLAB....289
Bits, bytes, and memory....289
Binary vs. base 10....290
Bit size and numbers....290
MATLAB defaults to `double` precision 292
Characters arrays and strings 293
Cell arrays 294
Back to the practice problem: making more efficient....297
TOPIC 15: Working with text 299
Practice problem: reactor cooling....299
Reminder: characters and strings 308
Reminder: cell arrays 309
Working with text 310
Loading text....310
Character arrays....315
String arrays....317
Character and string functions 318
Converting numbers to characters 319
Practice Problem: analyzing song lyrics....320
TOPIC 16: Working with text, pt 2 320
Reminder: working with text 320

Practice Problem: analyzing song lyrics 323
Practice Problem: the central dogma of molecular biology 324
TOPIC 17: Text and structure arrays 337
Practice Problem: the central dogma of molecular biology 337
Writing text data to a file 358
Structure arrays 358

```
% Load the data we just collected
rawData = load('sensorData5.mat')
```

```
rawData = struct with fields:
    Position: [42×6 timetable]
```

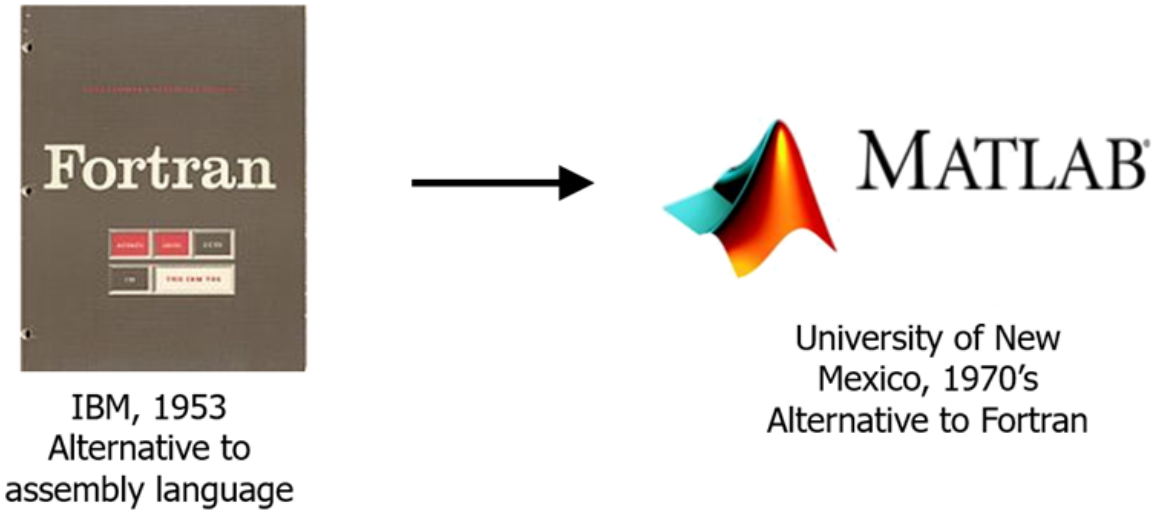
..... 358
Creating a structure array 359
Relation to graphics objects 361
Example: visualizing "particle" movement 364
TOPIC 18: Intro to audio and image processing 371
Numeric data, audio, and images 371
Importing data and reading files 371
Exporting data and writing files 372
What is a "signal?" 372
Introduction to audio processing 372
Stereo vs. mono sound 376
Filtering a 1-D signal 377
Creating audio signals 382
Introduction to image processing 383
Color images 384
Loading multiple signal files 386
Example: removing the cat from the image 387
TOPIC 19: Revisiting images and graphics 388
Reminder: audio processing 388
Reminder: image processing 389
Basic image filtering 396
Binary (logical) images 399
Revisiting plotting and graphics 416
Animating objects 420
Keyboard callbacks 423
TOPIC 20: Simulations and Callbacks (cont'd) 426
Revisiting simulations 426
Practice problem: simulating an infectious disease 426
TOPIC 21: Callbacks and simple interfaces 440
Miscellaneous information: changing the directory and accessing all files in the current folder 440
Reminder: keyboard callbacks (key press) 441
Keyboard callbacks (key release) 443
Mouse callbacks 445
Storing data in an interface with `guidata` 451
Example: Revisiting Tic Tac Toe 454
Creating figure UI objects using `uicontrol` 458
TOPIC 22: Calculus in MATLAB 461
Practice problem: diagnosing sickle cell disease 461
Calculus in MATLAB 463
1-D Differentation 463

Example: Usain Bolt and the 100 m 464
Visualizing and creating multi-dimensional derivatives 467
Visualizing velocity fields: `quiver` plots 468
Visualizing velocity fields: `surface` plots 469
Multi-dimensional differentiatial calculus 472
Integration 477
Engineering example: controllers 480
TOPIC 23: Differential equation solvers and other numerical methods 485
Solving ordinary differential equations (ODEs) 485
Taylor series approximations 486
Euler method: simple approximation of an ODE solution 488
Predictor-corrector method: better approximation of a solution 490
Runge-Kutta solutions to differential equations 492
Differential equation solvers in MATLAB 494
Example: chemical reaction kinetics 497
ODEs of higher orders 499
Supplying additional arguments to `ode45` 502
Example problem: creating an interactive physics simulator 503
TOPIC 24: Symbolic variables, equations, and functions 508
Symbolic math in MATLAB 509
Declaring symbolic numbers and variables 509
Symbolic expressions and functions 511
Plotting symbolic functions 513
Solving equations of one variable 518
Solving sets of linear equations 519
Solving sets of nonlinear equations 521
Calculus 525
Differential equations 527
Example problem: Harmonic oscillator 529
TOPIC 25: Animations 534
Reminder: animating simple graphical objects 534
Reminder: the flow of information in an interface 536
Animations 538
Revisiting image processing 539
Images as graphical handles objects 540
Image transparency 542
Creating your own `.png` images with transparency data 546
Object animation 547
Animation with timer objects 551
Example problem: rock simulator 553
Example problem: drag and drop 559
Functions 560
Topic 1 Functions: 560
Topic 3 Functions: 561
Topic 7 Functions 562
Topic 8 Functions 562
Topic 11 Functions 563
Topic 12 Functions 565
Topic 14 (Legrange Function) 567
Topic 15 Functions 567
Topic 19 Functions 569

Topic 20 Functions 570
Topic 21 Functions 571
Topic 23 Functions 583
Functions for Topic 23 stored here: 583
Topic 24 Functions: 586
Topic 25 Functions 587

Origins of MATLAB

MATLAB (Matrix Laboratory) is descended in part from the programming language FORTRAN, which was designed to support formula translation.



MATLAB is a high-level language that automatically interprets your instructions (code) into machine-level computation.Computation, ultimately performed at the transistor scale, is a physical process that takes time. For this class, our focus is on computational solutions to problems. We want code that is:

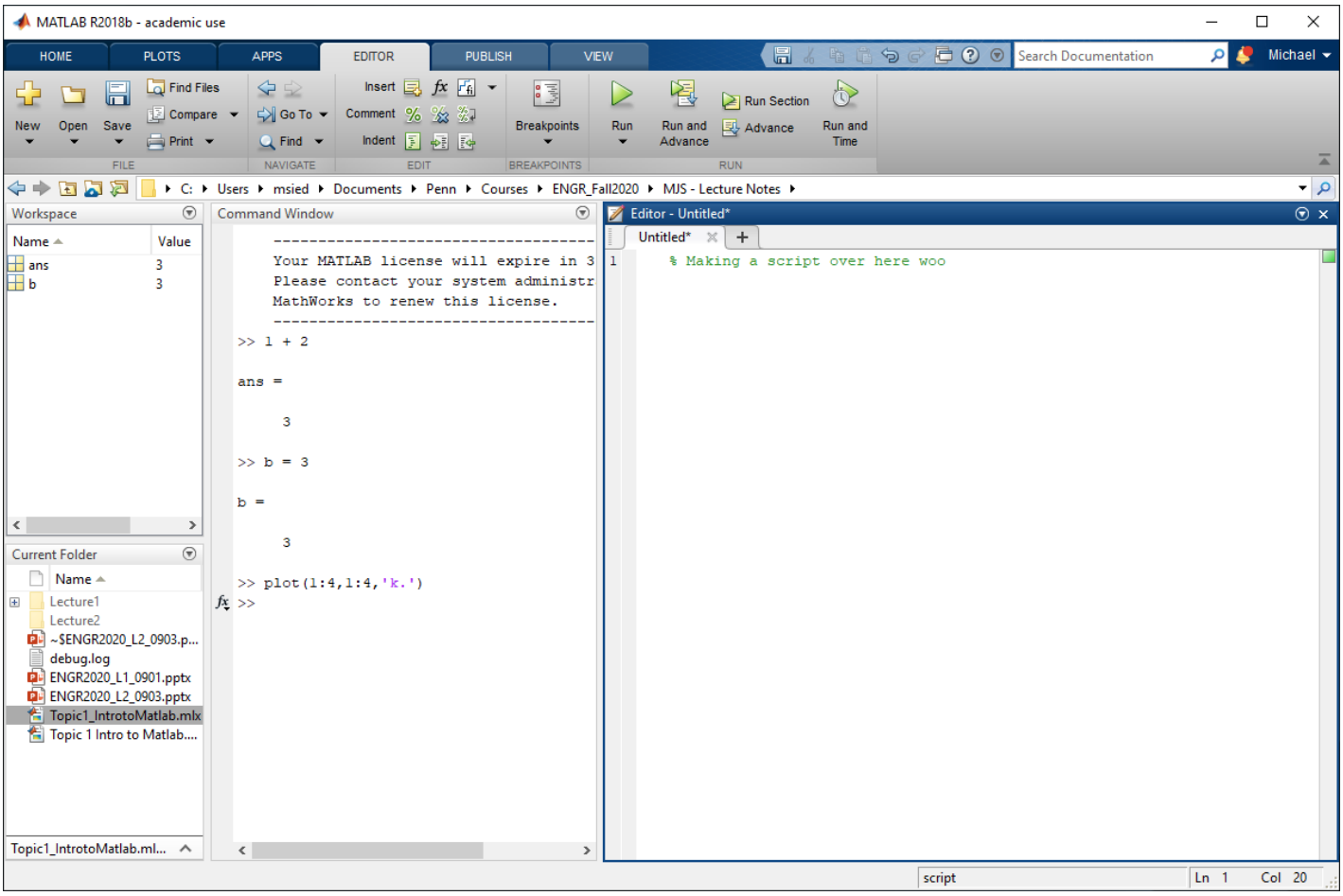
1. **Correct**: it robustly does what we want
2. **Computationally efficient**: it does what we want quickly
3. **Clear**: it is easy to understand and adapt what we are doing

MATLAB Interface

The MATLAB interface is fully customizable, in that you can organize the following modules. The organization of these is a user preference.

- **Workspace:** this shows the variables stored and able to be called
- **Command Window:** depicts errors, notifications, and previously called lines of code. Also contains the command line (>>) in which bits of code can be tested. Scripts and functions may also be called using the command line.
- **Editor:** compose and view livescripts, scripts, and functions.

- **Command history:** lists previously called commands and functions.
- **Current Folder:** depicts the files and folders in the current directory.
- **Directory:** current directory (if you contain a script or function if you are going to call it).



Working at the command line

The command line is useful for testing snippets of code or calling scripts or functions. Note that using a single "code block" in the LiveScript lecture notes *effectively* accomplishes the same thing. Note where the output is visible.

```
disp('hello world')
```

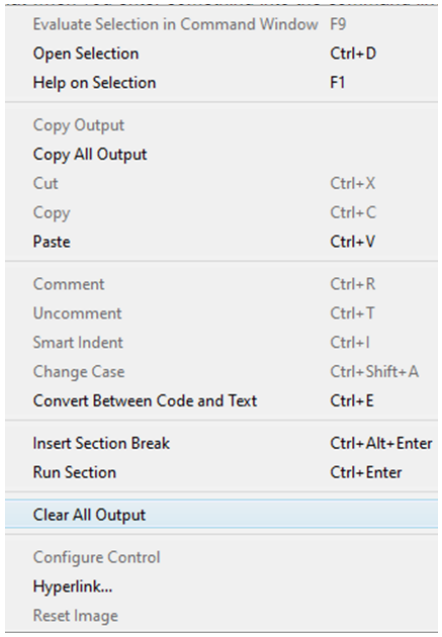
```
hello world
```

- Try inputting some values
- Note that when you enter something into the command line, the variable is stored in the workspace.
- `ans` contains the last output. **Avoid calling `ans`**, as it is very easy to lose track of what the last value called was. **Instead, use a statement to define a variable such that the value of that variable can be called.**

Using LiveScripts

For lecture notes, we will use LiveScripts to organize material and interactively run through code.

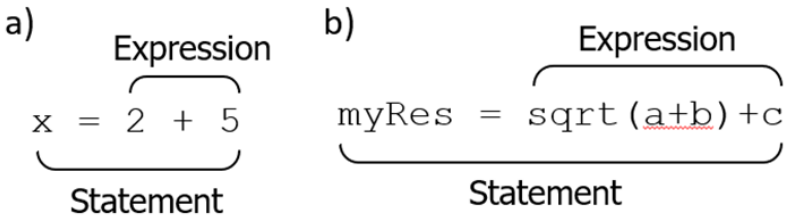
- LiveScripts are organized into sections. To "run" the active section, press `cntl + Enter`
- To clear outputs from code, right-click and select "clear all outputs."



General notes on MATLAB syntax and peculiarites

Statements and Expressions

Expressions and statements are the fundamental building blocks of code. Expressions are located on the right side of a statement.



The above reads as:

- a) "2 and 5 are added and stored in variable x"
- b) "the square root of the sum of variables `a` and `b` is added to `c` and stored in varilable `myRes`"

These statements would execute:

```
b = 2
x = b + 5
y = sqrt(x)
```

These statements would not execute:

```
2 = b
b + 5 = x
sqrt(x) = y
```

Naming conventions for variables

In the above examples, a **scalar** is stored in a **variable** that becomes visible in the **Workspace**. When creating variables, the following naming conventions should be followed:

1. Must consist of only the letters a-z, the digits 0-9, and underscores (_)
2. Must start with a letter
3. Should not overwrite built-in functions
4. Should not overwrite built-in variables

Comments, suppression, and separation

Comments are initiated by percent signs (%). Comments are vary important for helping **you** and others understand your code.

```
%This is a comment, they are useful
```

Some resources quote "one comment for every three lines of code" - this is dependent on the code. Some code contains more characters than productive syntax. When in doubt, add comments for clarity. When using MATLAB in Windows, large swaths of highlighted code can be *commented* using "`ctrl`" + "`r`" and *uncommented* using "`ctrl`" + "`t`".

A semicolon (`;`) placed at the end of a statement suppresses output to the command line

```
x = 2+5; %This comment is suppresed
y = 2+3 %This comment is not suppresed
```

```
y = 5
```

It is good practice when starting to have one statement per line, for ease of visualization. However, commas can be used to combine statements on a single line without suppression

```
x = 2 + 5, y = x+3
```

```
x = 7
y = 10
```

Procedural programming order

We will primarily use MATLAB as a procedural programming interface. Lines of contiguous code are executed in the order written and the results of expressions are placed into the statement variable.

```
% First, the scalar 3 is placed in variable x
x = 3;
% Second, variable x (equal to 3) is squared, 2 is added, and the result
```

```
% is placed back into variable x (now equal to 11)
x = x^2 + 2;
% Third, variable x (equal to 11) and 5 are added and placed in variable y
y = x + 5
```

```
y = 16
```

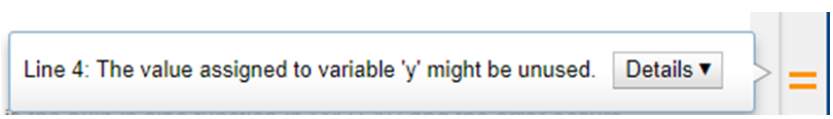
Errors and Warnings

An extremely useful feature of MATLAB is that it alerts you (and stops running) when your code has an **error**. For example, typing `sin` into the command line and running the code produces the following error.

```
>> sin
Error using sin
Not enough input arguments.
```

The nature of the error text provides insight into what went wrong. In this case, `sin` is the built-in sine function in MATLAB and the error occurs because we are not providing enough input arguments to that function.

Warnings, on the other hand, do not interrupt the code but alert that user of a potential oversight or computational inefficiency. Errors and warnings correspond to red and yellow lines in the right bar of a MATLAB script file, as well as annotations in the text. In this LiveScript file, the following warning is observed by hovering the cursor over the warning alert.



Using the Editor

Assemblages of code can be composed in the editor window and stored as an .m file for more permanence. These assemblages of code are referred to as either functions or scripts.Functions start with the `function` declaration whereas scripts are just assemblages of statements.

<u>Function</u>

```
function c = Pythagorean(a,b)
% Compute c for the relation
% a^2+b^2=c^2.
% a, b, and c are scalars.

% Compute c.
c = sqrt(a^2+b^2);
end
```

<u>Script</u>

```
% Pythagorean_script.m

clear, clc, close all

% Define values a and b.
a = 5;
b = 17;

% Compute c.
c = sqrt(a^2+b^2)
```

Functions

Functions usually take inputs and return outputs. In the above function:

- The function is called by the name `Pythagorean`
- Inputs `a` and `b` are provided as scalar numbers
- The value of `c` is returned

NOTE: when incorporating functions into LiveScripts, all functions are stored at the end of the LiveScript document.

Functions can be invoked (i.e. "called") from the command line or another function.

```
Pythagorean(1,1)
```

```
ans = 1.4142
```

```
result = Pythagorean(1,1)
```

```
result = 1.4142
```

Note: when you assign variables in a LiveScript, the variable is stored in the Workspace.

- Variables created within a function stay within the function - only function outputs are returned
- For example, the command line call `result = Pythagorean(2,5)` yields the variable `result` in the user workspace, but the variables in the function (`a`, `b`, and `c`) are nowhere to be found

Scripts

Scripts and command line calls operate in the user workspace and the variables generated can be seen by the command line and other scripts (but not functions). Scripts are saved as `.m` files.

- New scripts or functions can be created by clicking the "New" button in the MATLAB interface.
- To open a script or function, either a) navigate to a .m file in the directory and click to open it, or b) type `open <name of function>` into the command line.

NOTE: you can open some preexisting functions in MATLAB. **DO NOT MODIFY THESE.**

Clearing the workspace and figures

It is useful to purge the workspace memory at the beginning of a script using the `clear` command to avoid accidental use of old variables.

- `clear`: clears the Workspace. Can also be used to clear specific variables
- `clc`: purges the command line
- `close all`: closes all figure windows

```
% Example code demonstrating the clear command. a is cleared from the
% workspace below
```

```
a = 5;
b = 3;
clear a
```

Working directory and paths

MATLAB can see the contents (e.g. files, scripts, and functions) in a set of pre-installed folders (i.e. paths) and your current working folder. You can see a list of all paths MATLAB can see by inputting `path` at the command line. When invoking an `.m` file from a path that MATLAB can't see, it will query you if you would like to "Change Folder" or "Add to Path" - **usually "Change Folder" is the best selection**.

To avoid unnecessary headaches, you are encouraged to create a new folder for each homework assignment or project.

Interrupting code execution

Runaway code (code that does not terminate naturally) can be interrupted:

- `ctrl + c` (Windows)
- `command + period` (Mac)
- The "Pause" and "Stop" buttons in the editor can also be used.

```
% Example of infinite loop. Try interrupting the code. Be careful with
% infinite loops!
a = 1;
while a ~= 1

end
```

MATLAB documentation and help

MATLAB has extensive help documentation. Some routes to the help documentation include:

- Enter `help <term>` on the command line (*the first thing I often try*)
- Help symbol, F1, or the help search bar in MATLAB
- The “Google method” -> google "MATLAB" and then what you are looking for.

One goal of this course is that these notes will be a useful "help" resource for you in your future classes.

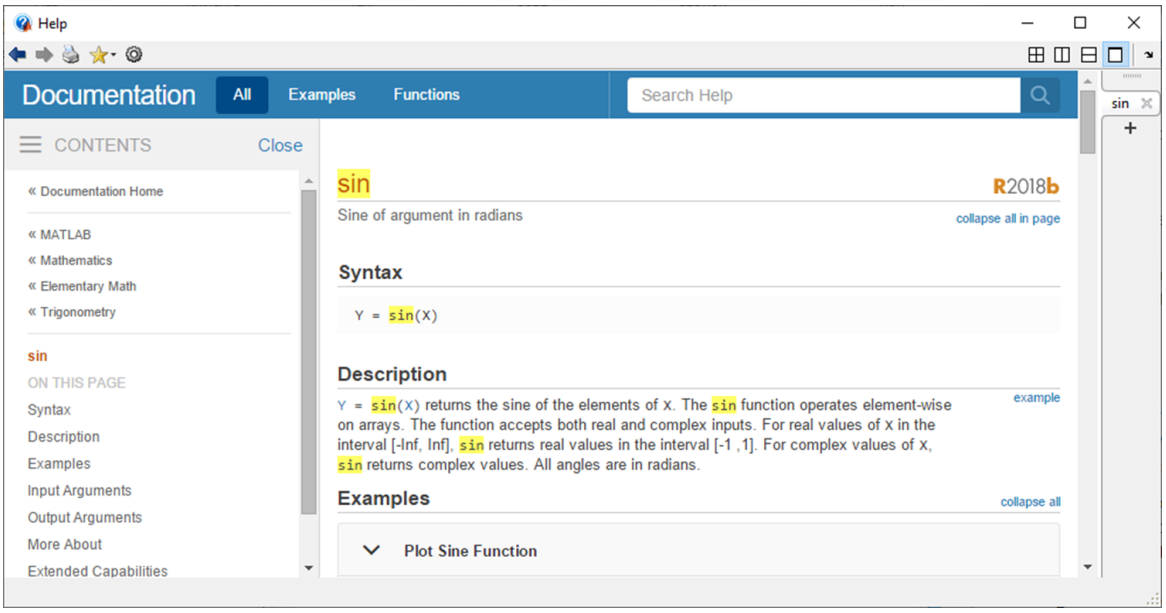
You can also open the reference documentation for a function from a link in the command line.

```
help sin
```

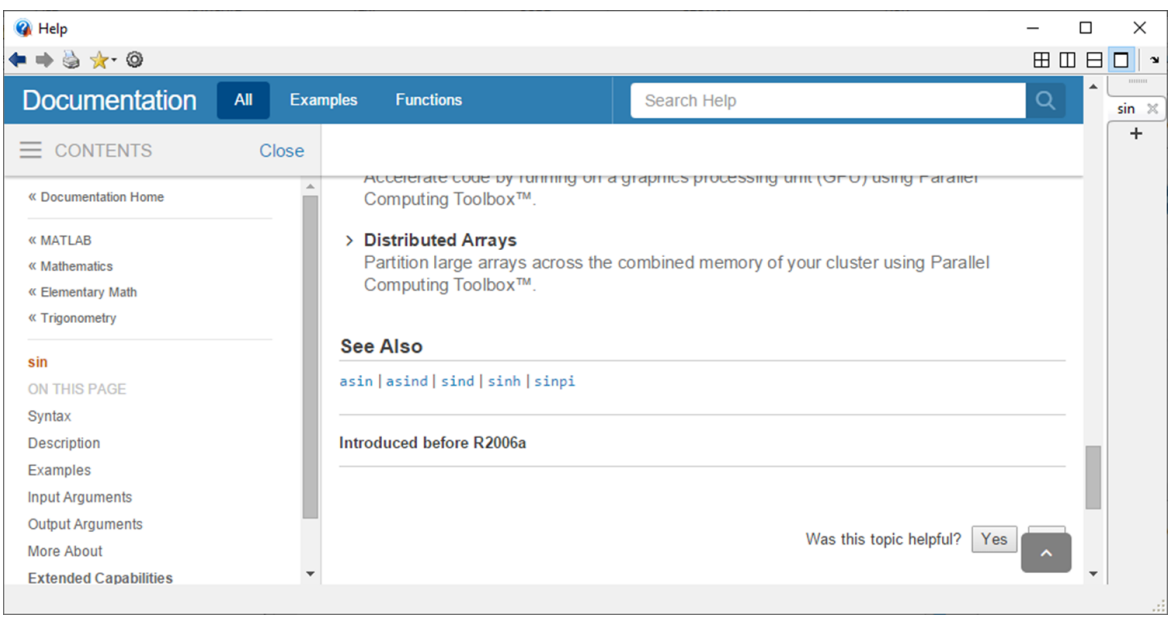
```
sin    Sine of argument in radians.
   sin(X) is the sine of the elements of X.

   See also asin, sind, sinpi.

   Documentation for sin
```

```
Other functions named sin
```

The following is the reference documentation for the `sin` function.



Note that the related functions ("See Also" section) at the end of each help page are useful for discovering functionality.



Built-in MATLAB functions

MATLAB has thousands of built-in functions for evaluating common mathematical and statistical functions. You will use these and **many** other functions both in this class and during the rest of your time at Penn. When in doubt, just check the help documentation.

```
sin(x), cos(x), exp(x), log(x), atan(x), cosh(x), sinh(x), mean(x),
sum(x), ...
```

```
a = sin(2*pi)
```

```
a = -2.4493e-16
```

```
b = sum([3, 4, 5])
```

```
b = 12
```

Basics of vectors: 1- D arrays

MATLAB is well-suited to work with arrays, which you can think of as sequences of numbers, where the **value** and **index** of each entry define the array. A 1-D array is known as a **vector** in MATLAB.

```
% Explicityly define an array with 5 numbers

x = [1, 3, 9, 11, -10.2];
```

An explicit description of a vector is useful when the vector components are irregular.

Creating vectors with slice notation

An often quicker way to produce vectors is to use the colon (`:`) operator. This enables you to create a vector by defining the **lower bound, upper bound, and increment size.**

```
% Create an array with numbers 1 though 15
x = 1:15;
% Create an array starting at 4 and increasing by 2 to 28
y = 4:2:28;
% Create an array starting at -1.3 and increasing by 0.1 to the value 1.3
z = -1.3:0.1:1.3;
```

Note: MATLAB will not include values above the upper bound specified in the slice notation. Note the values in the following array.

```
a = 1:2:6;
```

Creating vectors with `linspace()` and `logspace()`

Another way to create vectors is to use `linspace()` and `logspace()` when the **number of increments** is more important than the increment size.

- `linspace` creates a vector of uniformly spaced components in linear space

```
% Create a vector with 5 elements uniformly spaced between (and including) 1
and 5
a = linspace(1,5,5)
```

```
a = 1×5
     1     2     3     4     5
```

- `logspace` creates a vector of uniformly spaced components in (base-10) logarithmic space

```
% Create a vector with 5 elements uniformly spaced logarithmically between
% (and including) 10^1 and 10^5
b = logspace(1,5,5)
```

```
b = 1×5
          10         100        1000       10000      100000
```

Other vectors

In addtion to the above, many other types of vectors can be created. Here are a few more, useful examples.

- **Empty** vector

```
c = []
```

```
c =

     []
```

- **Ones** vector

```
d = ones(1,5)
```

```
d = 1×5
     1     1     1     1     1
```

- **Zeros** vector

```
f = zeros(1,5)
```

```
f = 1×5
     0     0     0     0     0
```

Accessing vector indices

Individual vector indices or groups of vector indices may be accessed/extracted. For the following assume we have defined the following vector in memory:

```
x = [5, 9, 4, 1, 7, 3, 4, 8];
```

- Access the first vector index:

```
y = x(1)
```

```
y = 5
```

- Access a range of vector indices:

```
x(1:3)
```

```
ans = 1×3
     5     9     4
```

```
x(6:8)
```

```
ans = 1×3
     3     4     8
```

- Access the last vector index

```
x(end)
```

```
ans = 8
```

- Access the second-to-last vector index

```
x(end-1)
```

```
ans = 4
```

- Access discrete indices in any order

```
x([3, 6, 1])
```

```
ans = 1×3
     4     3     5
```

Replacing indices and adding indices to vectors

The existing contents of a vector may be replaced and indices may be added. Assume we have defined the following vector in memory, which we will continuously modify:

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

```
x = 1×3
     1     2     3
```

- Replace the first index

```
x(1) = 5
```

```
x = 1×3
     5     2     3
```

- Replace the first and second indices

```
x(1:2) = [8,9]
```

```
x = 1×3
     8     9     3
```

- Replace the first and third indices

```
y = [8, 9, 3]
```

```
y = 1×3
     8     9     3
```

```
y([1,3]) = [5,1]
```

```
y = 1×3
     5     9     1
```

- Replace the ultimate and penultimate indices

```
y(end-1:end) = [6,8]
```

```
y = 1×3
     5     6     8
```

- Add a scalar after the last index

```
z = [2, 6, 8]
```

```
z = 1×3
     2     6     8
```

```
z(end+1) = 4
```

```
z = 1×4
     2     6     8     4
```

- Add multiple numbers to the end of the vector

```
z(end+1:end+2) = [5,7]
```

```
z = 1×6
     2     6     8     4     5     7
```

- Replace all odd indices with a scalar

```
z(1:2:end) = 1
```

```
z = 1×6
```

```
     1     6     1     4     1     7
```

- Replace all even indices with a scalar

```
z(2:2:end) = 4
```

```
z = 1×6
     1     4     1     4     1     4
```

- Add an index more than one index after the ultimate index

```
z(end+2) = 7
```

```
z = 1×8
     1     4     1     4     1     4     0     7
```

The rules for replacing vector indices are thus:

1. Scalars can replace a single index or multiple indices
2. Vectors can replace multiple indices of a vector if the length of the replacement equals the number of indices

```
x = [5, 2, 1, 8]
```

```
x = 1×4
     5     2     1     8
```

Removing vector indices

Vector indices may be removed by specifying an empty set (empty vector).

```
x = [5, 3, 9, 4, 1, 6]
```

```
x = 1×6
     5     3     9     4     1     6
```

- Remove the first index

```
x(1) = []
```

```
x = 1×5
     3     9     4     1     6
```

- Remove odd indices

```
x(1:2:end) = []
```

```
x = 1×2
```

```
     9     1
```

Concatenating vectors

Multiple vectors may be combined via concatenation, provided both vectors are oriented correctly (both row vectors).

```
x = [9, 5, 8, 2]
```

```
x = 1×4
     9     5     8     2
```

- Concatenate same vector

```
x(1) = []
```

```
x = 1×3
     5     8     2
```

- Concatenate multiple vectors

```
x(1:2:end) = []
```

```
x = 8
```

Operations on vectors

You can perform mathematical operations on arrays just as you can scalar values (in most cases).

- You can **add** and **subtract** two arrays in MATLAB as long as they are the same size - if they are not the same size you will get an error.

```
x = [1 2 3 4 5];
y = [7 8 9 10 11];

% element-wise addition adds corresponding elements
z = x + y;
```

- Multiplication and division are different -- simply using * and / will perform matrix multiplication and division (which we will explore later)
- Instead, **use a period (`.`) to perform element-wise multiplication and division**

```
z1 = x.*y; % element-wise multiplication
z2 = x./y; % element-wise division
```

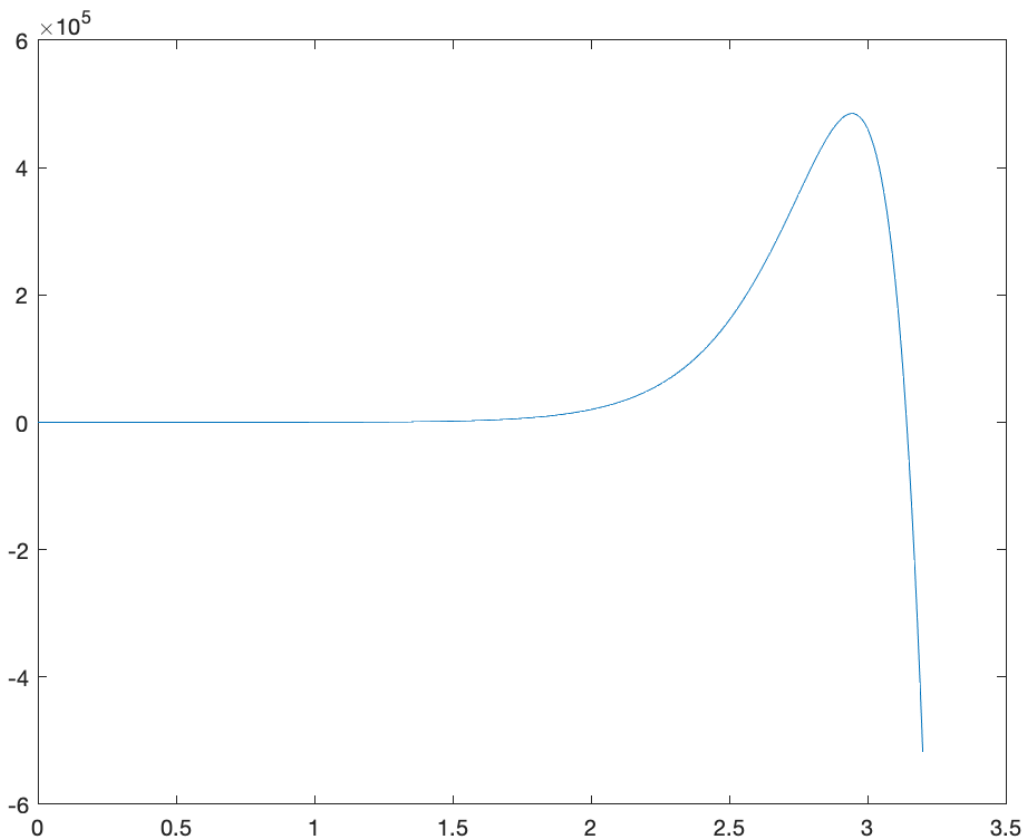
Almost all of MATLAB's built-in functions will accept arrays as arguments and perform operations on these arrays in an element-wise manner.

```
x = 1:10:1000;
```

```
% Compute the square root of each element in the array x
y = sqrt(x);
```

We will see later that **vectorization** in this way is more computationally efficient than using loops to operate on each element. In addition, operations can be performed on the output from each function.

```
% create an array of values
x = 0:0.01:3.2;

% evaluate an expression on every element in % the array x producing a new
array called y
y = exp(5*x).*sin(x);

% produce a plot
plot(x, y)
```



TOPIC 2: Introduction to arrays: vectors, matrices, and more

ENGR105, 9/8/20

Revisiting vectors

Previously, we learned how to *create vectors*, *access vector elements*, and *delete vector elements*. We did this for *row* vectors, the default orientation of vectors formed by slice notation, linspace, etc.

For example, consider operations on the following row vector, which has <u>one row</u> and four columns.

```
% Create an example vector
x = 1:2:8
```

```
x = 1×4
     1     3     5     7
```

```
% Call indices 1 and 4 of vector x
x([1,4])
```

```
ans = 1×2
     1     7
```

```
% Delete the element associated with the index 4
x(4) = []
```

```
x = 1×3
     1     3     5
```

```
% Create new vectors via concatenation
x1 = [x,x]
```

```
x1 = 1×6
     1     3     5     1     3     5
```

```
x2 = [x,linspace(1,3,6)]
```

```
x2 = 1×9
    1.0000    3.0000    5.0000    1.0000    1.4000    1.8000    2.2000 ⋯
```

Why are we doing this?

1. Array operations are a fundamental component of using MATLAB.
2. If you know how to create vectors and matrices, you can "easily" solve engineering problems (goal for this week).
3. Practice using "logic" and math concepts to try to create something new in the most efficient way possible.

Example

Starting with vector x, defined below, perform the following steps:

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

1. Delete very 4th entry starting with the first.
2. Square every odd element.
3. Move the last 5 entries to the front

```
% Step 1
x(1:4:end) = [];
% Step 2
x(1:2:end) = x(1:2:end).^2;
% Step 3
%end is an inclusive range
%we concatinate a new vector that starts with the last 5 numbers and
%carries on to the first 5 numbers
x = [x(end-4:end), x(1:end-5)]
```

```
x = 1×15
     1     4    25     6    16    16     2    25     4     9     1     4
5 ⋯
```

Column vectors

Column vectors can be created via direct input (i.e. explicitly defining the vector) or the transpose (`'`) command.

- Column vector created using semicolons

```
x = [5; 4; 6; 9]
```

```
x = 4×1
     5
     4
     6
     9
```

- Column vector created using the **transpose**

```
% Creating a new column vector
y1 = (1:4)';
% Creating a new vector by taking the transpose of an existing one
y2 = x'
```

```
y2 = 1×4
     5     4     6     9
```

Live row vectors, a created column vector may be addressed using single index notation to refer to rows.

- Single index notation: specifying an index or range of indices using a vector. This is exactly what we used proviously.

```
x = [5; 2; 1; 8]
```

```
x = 4×1
     5
     2
     1
     8
```

```
% Replacing the first element
x(1) = 9
```

```
x = 4×1
     9
     2
     1
     8
```

```
% Replacing the first two elements
x(1:2) = [4, 3]
```

```
x = 4×1
     4
     3
     1
     8
```

```
% Replacing the second-to-last element
x(end-1) = 9
```

```
x = 4×1
     4
     3
     9
     8
```

```
% Deleting the first and third elements
x([1,3]) = []
```

```
x = 2×1
     3
     8
```

In other words, the same index notation is used for both row and column vectors. Though we will often use row vectors because these are the default orientation, it is important to remember that the vector you have created has a particular orientation.

Determining the number of entries in a vector

It is sometimes helpful to know the length of a vector in order to perform operations based on its size. Use the `length()` command to determine the number of entries in a vector.

```
x = [5, 2, 1, 4, 9, 1]
```

```
x = 1×6
```

```
     5     2     1     4     9     1
```

```
L = length(x)
```

```
L = 6
```

You can use `length()` to operate on, e.g. the first half of a vector, even if you don't know it's size beforehand.

```
% Return the first half of the vector
x(1:length(x)/2)
```

```
ans = 1×3
     5     2     1
```

Reminder: if you assign a scalar to a vector, as below, the scalar will assigned to all indices of the specified vector.

```
x(1:length(x)/2) = 50;
x
```

```
x = 1×6
    50    50    50     4     9     1
```

Precedence

Note a detail in the above code:

```
1:length(x)/2
```

```
ans = 1×3
     1     2     3
```

This is not the same as the following, due to precedence.

```
%Divides the original vector length in half
(1:length(x))/2
```

```
ans = 1×6
    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
```

This is because MATLAB follows an order of operations for vector and array operations.

Precedence	Operator	MATLAB Representation
1	Parentheses	(A)
2	Power, transpose	A^x, A.^x, A′, A.′
3	Multiplication and division	A*B, A.*B, A/B, A./B, A\B, A.\B
4	Addition and subtraction	A+2; A-2
5	Colon operator	1:5

The colon operator takes the lowest precedence and so the following are equivalent:

```
1:length(x)/2
```

```
ans = 1×3
     1     2     3
```

```
1:(length(x)/2)
```

```
ans = 1×3
     1     2     3
```

We can add to the classic order of operations mnemonic: "Please Excuse My Dear Aunt Sally's Colon"

Operation precedence may be overidden using parentheses.

```
a = 1;
b = 2;
c = 3;
d = 4;
```

Consider the following examples

- Statements are evaluated from left to right with multiplication and division being executed first

```
a+b*c/d
```

```
ans = 2.5000
```

- Here, the parentheses specify that addition should be executed first (overriding mathematical precedence)

```
(a+b)*c/d
```

```
ans = 2.2500
```

- Upholds the execution order prescribed by mathematical operator precedence, the use of parentheses is unnecessary

```
a+(b*c)/d
```

```
ans = 2.5000
```

Parentheses can be included to improve the clarity of an expression even if not modifying precedence

- z1 = 2*pi*x^2*y+1;
- z2 = 2*pi*(x^2*y)+1;

`z1` and `z2` give the same result, but it is easier to see the relationships between pre-factors and independent variables in `z2`

Arrays

A *vector* is only one type of *array*. MATLAB stands for Matrix Laboratory because the MATLAB system is designed to efficiently work with data arranged in arrays of all sizes, especially 2-D *matrices*. In fact, all numbers in MATLAB are stored as arrays:

- **Scalar**: (`1x1` array)

```
a1 = 1;
disp(a1)
```

```
     1
```

- **Vector**: (`1xN` array) or (`Nx1` array). In this case, there are either N rows or N columns

```
a2 = [1,5,9];
disp(a2)
```

```
     1     5     9
```

```
a3 = [1;5;9];
disp(a3)
```

```
     1
     5
     9
```

- **2D matrix**: (`MxN` array). That is, a 2D matrix consists of M rows and N columns. In this case, there are 2 rows and 3 columns.

```
%remember the semicolon makes the vector an array
a4 = [1,5,9;9,5,1];
disp(a4)
```

```
     1     5     9
     9     5     1
```

- **Multidimensional matrix**: (`MxNx...xP` array). A 3D+ array consists of rows, columns, "pages", "volumes", etc. In this case there are 2 rows, 3 columns, and 2 "pages."
-

```
a5(:,:,1) = [1,5,9;9,5,1];
a5(:,:,2) = [2,3,4;4,3,2];
disp(a5)
```

```
(:,:,1) =
```

```
     1     5     9
     9     5     1


(:,:,2) =

     2     3     4
     4     3     2
```

Dimensions of an array

The `length()`, `size()`, and `numel()` commands are useful for evaluating the properties of an array.

- As seen above, `length()` is useful for vectors --it returns the number of elements in the vector. This command operates independently of the vector being a row or column vector.

```
length(a2)
```

```
ans = 3
```

```
length(a3)
```

```
ans = 3
```

- Using length on a matrix is less useful -- in general, it returns the greatest dimension of the array

```
length(a4)
```

```
ans = 3
```

- To evaluate the dimensions of an arbitrary-sized array, use the `size()` function. This functions returns a **vector**:

```
size(a4)
```

```
ans = 1×2
     2     3
```

This can be read as, "a4 is a `2x3` matrix." The first entry represents the number of rows of a4, the second entry represents the number of columns.

- size() can be used for higher-order arrays also

```
size(a5)
```

```
ans = 1×3
     2     3     2
```

This can be read as, "a5 is a `2x3x2` matrix." The first entry represents the number of rows of a4, the second entry represents the number of columns.

- The `numel()` command returns the total number of indices ("number of elements") in the array.

```
numel(a4)
```

```
ans = 6
```

Matrices

For now, we will focus on 2D matrices. For simplicity we will call these "matrices," but it is implied that we mean 2D matrices or 2D arrays. You will work with matrices a lot in your linear algebra courses, as well as advanced engineering courses.

- *Relevant UPenn Course*: ENM 240 Differential Equations and Linear Algebra
- *Relevant UPenn Course*: MATH 240: Calculus, pt. III

Constructing matrices

Matrices may be explicitly constructed by inputting row and column values as a list. A semicolon divides rows.

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

```
A = 2×3
     2     5     9
     6     8     3
```

Matrices may also be constructed from existing vectors using concatenation. **Note the orientation of the vectors.**

```
a = [1, 2, 3];
b = [4, 5, 6];
% Concatenate vertically
A1 = [a;b]
```

```
A1 = 2×3
     1     2     3
     4     5     6
```

```
% Concatenate horizontally
A2 = [a,b]
```

```
A2 = 1×6
     1     2     3     4     5     6
```

```
a = [1; 2; 3];
b = [4; 5; 6];
% Concatentate
```

```
B1 = [a;b]
```

```
B1 = 6×1
     1
     2
     3
     4
     5
     6
```

NOTE: Dimensions must match along the plane of concatenation

For vectors:

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix}\begin{bmatrix}5\\6\\7\\8\end{bmatrix}$$

Yes

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix}\begin{bmatrix}5\\6\\7\end{bmatrix}$$

No

$$\begin{bmatrix}1\\2\\3\\4\end{bmatrix}\begin{bmatrix}5\\6\\7\end{bmatrix}$$

Yes

For matrices:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \text{No}$$

$$\begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$$

Yes

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\begin{bmatrix} 5 & 6 \\ 7 & 8 \\ 9 & 10 \end{bmatrix}$$

No

Takeaway: **be aware of the dimensions of the vectors and matrices you are working with.**

```
% Concatenate vectors
a = [1 2 3; 4 5 6]
```

```
a = 2×3
     1     2     3
     4     5     6
```

```
b = [7 8 9; 10 11 12]
```

```
b = 2×3
     7     8     9
    10    11    12
```

```
% Concatenate matrices
C1 = [a;b]
```

```
C1 = 4×3
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

```
C2 = [a,b]
```

```
C2 = 2×6
     1     2     3     7     8     9
     4     5     6    10    11    12
```

Accessing array indices

Specific indices from a 2D matrix can be accessed by row and column indexing in the form **Matrix(row, column).**

```
A = [1:3;4:6;7:9]'
```

```
A = 3×3
     1     4     7
     2     5     8
     3     6     9
```

```
A(3,1)
```

```
ans = 3
```

```
A(1,3)
```

```
ans = 7
```

In the above, we took the **transpose** of a matrix. This swaps the row and column index of every element. For example, the following `2x3` matrix becomes a `3x2` matrix.

```
B = [1:3;4:6]
```

```
B = 2×3
     1     2     3
     4     5     6
```

```
B'
```

```
ans = 3×2
     1     4
     2     5
     3     6
```

Useful: Full rows or columns can be selected using the colon operator (`:`).

```
A = [1:3;4:6;7:9]'
```

```
A = 3×3
     1     4     7
     2     5     8
     3     6     9
```

```
% Return a column vector of all rows in the first column
A(:,1)
```

```
ans = 3×1
     1
     2
     3
```

```
%Colon returns all columns in that row?
% Return a row vector of all columns in the second row
A(2,:)
```

```
ans = 1×3
     2     5     8
```

Submatrices may also be extracted.

```
% Return a submatrix of entries in first and second rows and columns
A(1:2,1:2)
```

```
ans = 2×2
     1     4
     2     5
```

```
% Return a submatrix of entries in first and second rows and all columns
A(1:2,:)
```

```
ans = 2×3
     1     4     7
     2     5     8
```

In my opinion, row and column indexing is preferred because it is more intuitive. However, array indices may also be accessed using a single subscript indexing ("unwound" matrix). Indices are evaluated starting in the upper left hand corner and parsed by column.

$$A = \begin{bmatrix} 1^{1} & 4^{4} & 7^{7} \\ 2^{2} & 5^{5} & 8^{8} \\ 3^{3} & 6^{6} & 9^{9} \end{bmatrix}$$

```
A(5)
```

```
ans = 5
```

A range of entries may be called to return a row vector of the corresponding elements.

```
A(1:5)
```

```
ans = 1×5
     1     2     3     4     5
```

Note: the colon operator (:) may be used to unwind the entire matrix into a *column vector*.

```
A(:)
```

```
ans = 9×1
     1
     2
     3
     4
     5
     6
     7
     8
     9
```

The diagonal components of a matrix may be called using the `diag()` command. Often, these elements are important for engineering applications - e.g. normal stresses from a stress tensor.

```
% Diagonal components of A
diag(A)
```

```
ans = 3×1
     1
     5
     9
```

Overwriting indices

Specific indices of the matrix may be directly overwritten using row/column or single subscript notation.

```
A = [1:3;4:6;7:9]'
```

```
A = 3×3
     1     4     7
     2     5     8
     3     6     9
```

```
% Replace the entry in the third row, first column
A(3,1) = 50
```

```
A = 3×3
     1     4     7
     2     5     8
    50     6     9
```

```
% Replace all entries in the "upper left of the matrix"
A(1:2,1:2) = 0
```

```
A = 3×3
     0     0     7
     0     0     8
    50     6     9
```

If a matrix index is assigned a value outside of the original matrix size, indices of "unknown value" will be padded with 0's

```
A(4,4) = 33
```

```
A = 4×4
     0     0     7     0
     0     0     8     0
    50     6     9     0
     0     0     0    33
```

Array indices at any index location can be specified even if the array does not exist.

```
% C has not be previously defined
C(3,3,3) = 1
```

```
C =
C(:,:,1) =

     0     0     0
     0     0     0
     0     0     0


C(:,:,2) =

     0     0     0
     0     0     0
     0     0     0


C(:,:,3) =

     0     0     0
     0     0     0
     0     0     1
```

Array mathematics

Addition and Subtraction

General rules for addition and subtraction for arrays of any dimension (e.g. vectors or matrices)

1. Scalars may be added to/subtracted from any array - **the scalar value is added to all elements of the array**
2. Arrays added to arrays must have the **same dimensions** and elements in correlated index locations can be added or subtracted.

```
% Create new matrix
A = [1,5;2,3]
```

```
A = 2×2
     1     5
     2     3
```

```
% Add 2 to every element

% Create new array by reordering the first and second row
B = [A(2,:);A(1,:)]
```

```
B = 2×2
     2     3
     1     5
```

```
% Matrix addition
```

Remember: matrix dimensions must agree. In other words, the matrices must have the same dimensions.

Multiplication

General rules for matrix/array multiplication:

1. Arrays may be multiplied by a **scalar**.
2. **Arrays of the same dimensions** may be multiplied index-by-index using the array multiplier/vectorization (`.*`)
3. The matrix product (*) may be evaluated for matrices if the "inner dimensions" of the original matrices match and the resulting matrix will have dimensions equal to the outer dimensions of the matrices being multiplied.

Case 1: multiplication by a scalar

```
A
```

```
A = 2×2
     1     5
```

```
     2     3
```

Case 2: element-by-element multiplication. Arrays of the same dimensions may be multiplied index-by-index using the array multiplier/vectorization (.*)

```
A
```

```
A = 2×2
     1     5
     2     3
```

```
B
```

```
B = 2×2
     2     3
     1     5
```

Case 3: matrix multiplication. For `A*B`, the **columns of A must equal the rows of B**.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

The element in the <u>first</u> row and <u>first</u> column of `A*B` is:

$$A * B = \begin{bmatrix} \boxed{a_{11} \quad a_{12}} \\ a_{21} \quad a_{22} \end{bmatrix} \begin{bmatrix} \boxed{b_{11}} & b_{12} \\ \boxed{b_{21}} & b_{22} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & ? \\ ? & ? \end{bmatrix}$$

The element in the <u>first</u> row and <u>second</u> column of `A*B` is:

$$A * B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ ? & ? \end{bmatrix}$$

Filling in the matrix:

$$A * B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

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

```
A = 2×2
     1     4
     2     3
```

```
B = [1,2;3,4]
```

```
B = 2×2
     1     2
     3     4
```

```
% Matrix multiplication
A*B
```

```
ans = 2×2
    13    18
    11    16
```

Remember: A*B is not usually the same as B*A

```
% Matrix multiplication with multiplication order reversed.
B*A
```

```
ans = 2×2
     5    10
    11    24
```

Pro tip: if you feel stressed about this, write out each entry systematically! You will work a lot more with matrices in your linear algebra courses.

Remember: the dimensions of the matrix product are equal to the outer dimensions of the original matrices <u>and</u> the inner dimensions must match

$$\overset{1 \times 3}{A = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}} \quad \text{and} \quad \overset{3 \times 2}{B = \begin{bmatrix} 2 & 1 \\ 5 & 3 \\ 4 & 6 \end{bmatrix}}$$

MATLAB syntax

$$C = A * B \Rightarrow$$

$$\begin{bmatrix} 1*2+2*5+3*4 & 1*1+2*3+3*6 \end{bmatrix}$$

$$(1 \times 3) \times (3 \times 2) \rightarrow 1 \times 2 \text{ output matrix}$$

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

```
B = 3×2
     2     1
     5     3
     4     6
```

```
A = 1:3
```

```
A = 1×3
     1     2     3
```

```
% To perform matrix multiplication, ensure the columns of the first matrix
% match the rows of the second
A*B
```

```
ans = 1×2
    24    25
```

Division

Division of a scalar by an array necessitates an <u>array operator</u>.

```
A = [1 2; 4 8]
```

```
A = 2×2
     1     2
     4     8
```

```
B = 16./A
```

```
B = 2×2
    16     8
     4     2
```

NOTE: `B = 16/A` will return an error.

NOTE: Matrix division has a different interpretation than scalar division. <u>Right division is different than left division</u>.

Element-wise division:

- *Right* division: `A./B` is equivalent to `A(i,j)/B(i,j)`
- *Left* division: `A.\B` is equivalent to `B(i,j)/A(i,j)`

```
% Define matrix A and vector b
A = [2,4;2,6]
```

```
A = 2×2
     2     4
     2     6
```

```
B = [1,2;1,3]
```

```
B = 2×2
     1     2
     1     3
```

```
% Perform right division

% Perform left division
```

Left matrix division. Consider the following matrix equation, in which matrix **A** is multiplied by vector **x** to give vector **b**.

$$\mathbf{Ax} = \mathbf{b}$$

<u>Note on dimensions</u>: assuming A is square (e.g. `2x2`), then **x** and **b** must be (e.g. `2x1`) column vectors. To solve for **x**, you must perform <u>left division</u> of matrix **A**, or left multiply by the inverse of **A**.

$$\mathbf{x} = \mathbf{A}\backslash\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}\backslash\mathbf{b} = \mathbf{A}^{-1}\mathbf{b}$$

The above works because $\mathbf{A}^{-1}\mathbf{A} = \mathrm{I}$, where I is the identity matrix, and $\mathbf{IX} = \mathbf{X}$ (by definition).

```
% Define matrix A and vector b
A = [2,4;2,6]
```

```
A = 2×2
     2     4
     2     6
```

```
b = [5;4]
```

```
b = 2×1
     5
     4
```

```
% Perform left division
A\b
```

```
ans = 2×1
    3.5000
   -0.5000
```

```
inv(A)*b
```

```
ans = 2×1
    3.5000
   -0.5000
```

Right matrix division. Consider the following matrix equation, in which matrix **A** is multiplied by vector **x** to give vector **b**.

$$\mathbf{x}\mathbf{A} = \mathbf{b}$$

Note on dimensions: assuming A is square (e.g. `2x2`), then **x** and **b** must be (`e.g. 1x2`) row vectors. To solve for **x**, you must perform right division of matrix **A**, or right multiply by the inverse of **A**.

$$\mathbf{x} = \mathbf{x}\mathbf{A}/\mathbf{A} = \mathbf{x}\mathbf{A}\mathbf{A}^{-1} = \mathbf{b}/\mathbf{A} = \mathbf{b}\mathbf{A}^{-1}$$

```
A = [2,4;2,6]
```

```
A = 2×2
     2     4
     2     6
```

```
b = [5,4]
```

```
b = 1×2
     5     4
```

```
% Perform right division
b/A
```

```
ans = 1×2
    5.5000   -3.0000
```

```
b*inv(A)
```

```
ans = 1×2
    5.5000   -3.0000
```

Application: solving a set of linear equations

You are likely familiar with canonical linear equations of the form `y=mx+b` or `ax+by=c`, which define a straight line in 2D space. Consider a set of two linear equations:

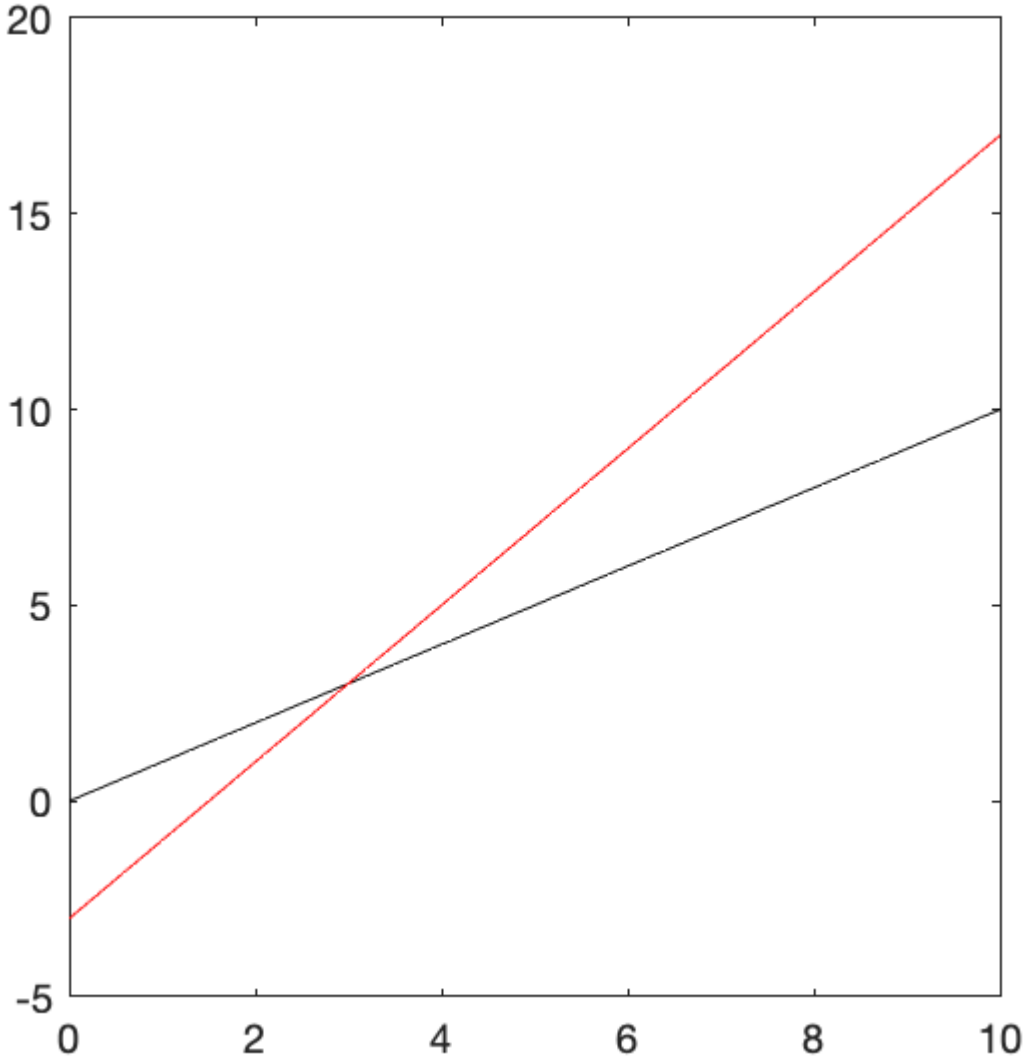
$$y = m_1 x + b_1$$
$$y = m_2 x + b_2$$

In engineering applications, you are often interested in solving a set of linear equations. There are three possibilities:

There is one solution.

$$y = x$$
$$y = 2x - 3$$

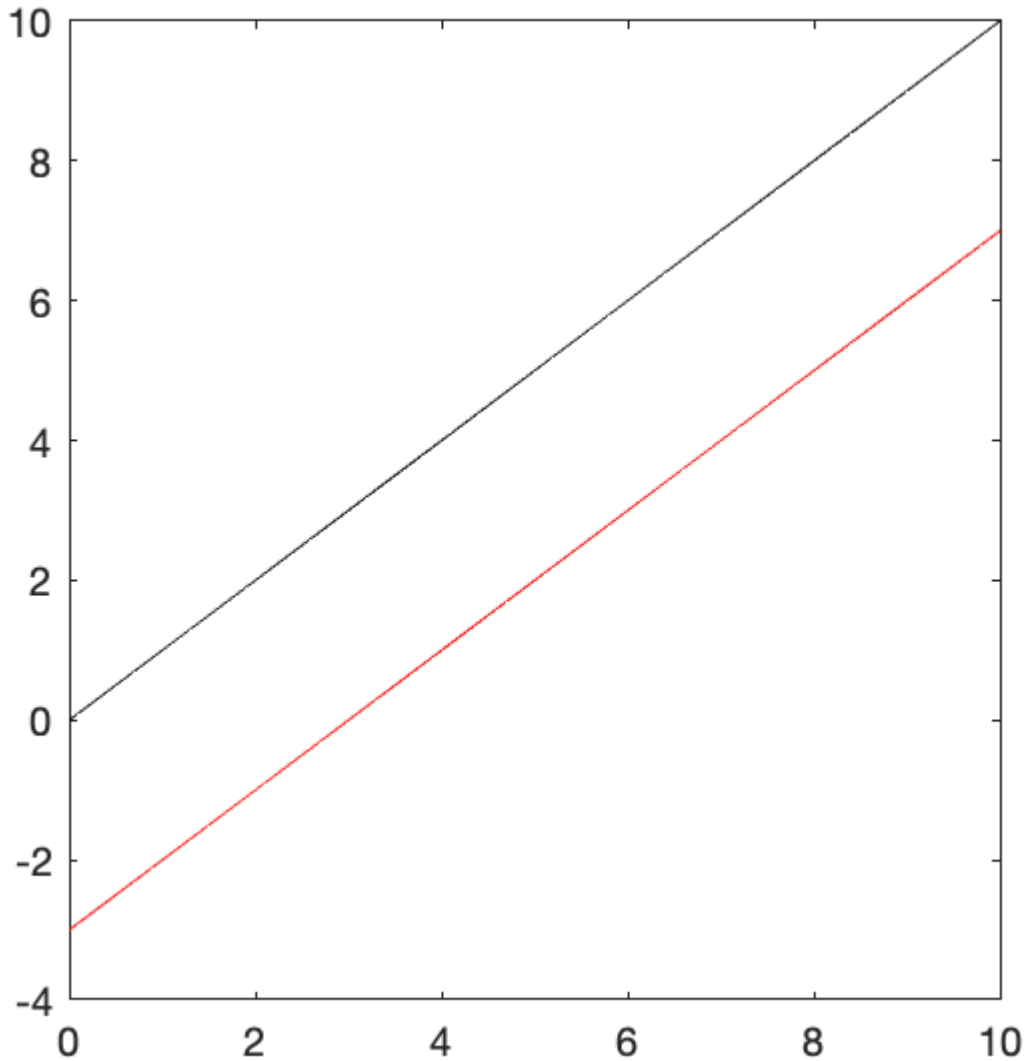
```
% x vectors
x = 0:.1:10;
% Create a figure window of a defined size
figure('Position',[0,0,300,300]);
plot(x,x,'k-',x,2*x-3,'r-')
```



There are no solutions.

$$y = x$$
$$y = x - 3$$

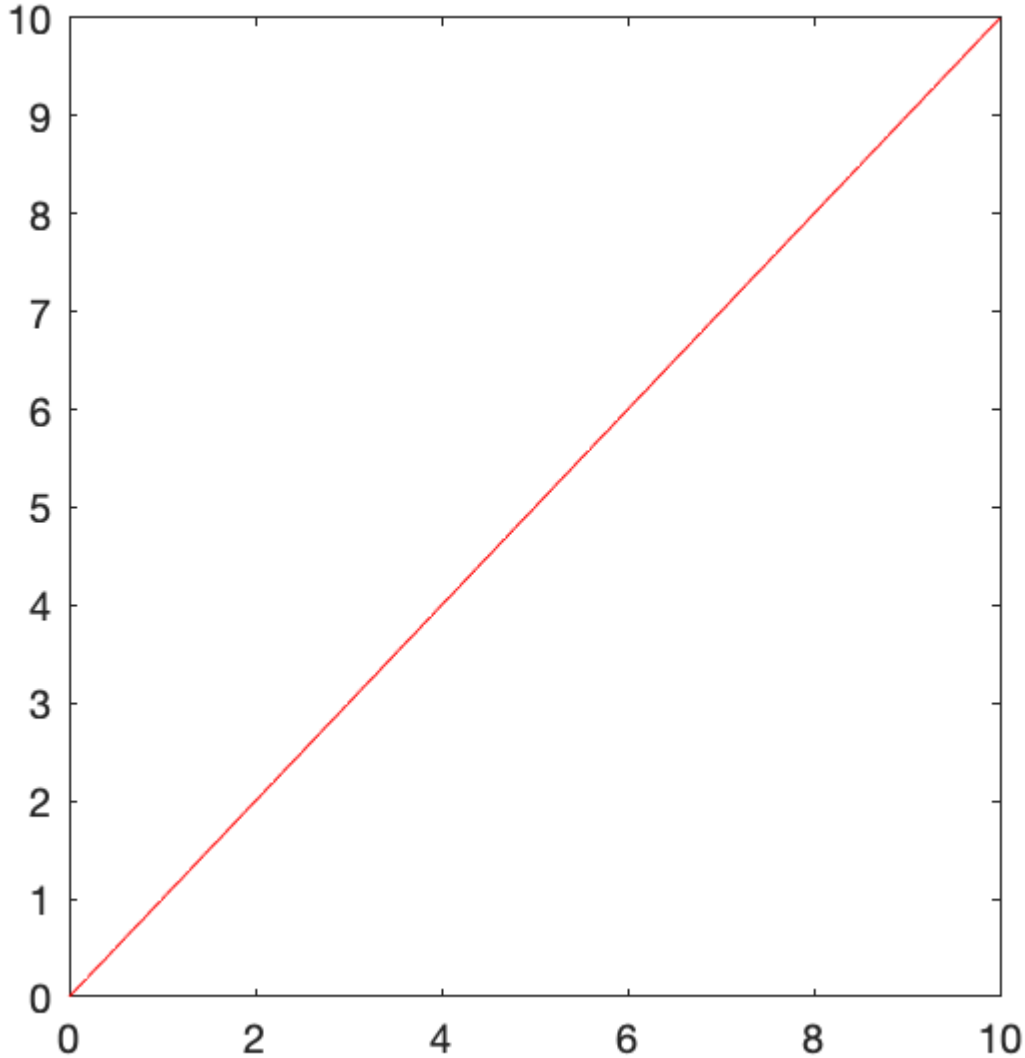
```
% Create a figure window of a defined size
figure('Position',[0,0,300,300]);
plot(x,x,'k-',x,x-3,'r-')
```



There are infinite solutions.

$$y = x$$
$$y = x$$

```
% Create a figure window of a defined size
figure('Position',[0,0,300,300]);
plot(x,x,'k-',x,x,'r-')
```



We will focus on systems that have <u>one solution</u>. Further study in linear algebra will provide conditions on matrix **A** and vector **b** that will enable you to determine the number of solutions.

One approach: substitution

Consider the two equations defined above. To solve, perform the following steps.

$$y = m_1 x + b_1$$
$$y = m_2 x + b_2$$

1) Set equations equal

$$m_1 x + b_1 = m_2 x + b_2$$

2) Solve for x

$$x(m_1 - m_2) = b_2 - b_1$$
$$x = \frac{(b_2 - b_1)}{(m_1 - m_2)}$$

3) Substitute to solve for y

$$y = m_1 \frac{(b_2 - b_1)}{(m_1 - m_2)} + b_1$$

This is fine, but requires equations in the above form and can get tedious.

Another approach: matrix representation

Rearranging the above system of equations:

$$y - m_1 x = b_1$$
$$y - m_2 x = b_2$$

Double check the matrix multiplication rules above. Based on the rules of matrix multiplication, the above set of equations can be represented in matrix form:

$$\begin{bmatrix} 1 & -m_1 \\ 1 & -m_2 \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Or:

$$\mathbf{Ax} = \mathbf{b}$$
$$\mathbf{A} = \begin{bmatrix} 1 & -m_1 \\ 1 & -m_2 \end{bmatrix}$$
$$\mathbf{x} = \begin{bmatrix} y \\ x \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

This is the case above corresponding to **left matrix division**. Thus, we can simply solve for y and x by performing a left division:

$$\mathbf{x} = \mathbf{A} \backslash \mathbf{Ax} = \mathbf{A} \backslash \mathbf{b}$$

Example 1. Solve the following system of equations.

$$y = x$$
$$y = 2x - 3$$

```
% Step 1: convert to matrix format
```

```
A = [1,-1;1,-2]; %Coefficient matrix defined above
b = [0;-3]; % Constant matrix
% Step 2: solve for x and y
```

Thus, the solution exists at `x = 3` and `y = 3`.

More general representation

More often, you will encounter systems with more than two variable. Thus, it is more convenient to think of a system of linear equations in the following form, where there are `n` variables and `n` equations.

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

The above can be represented compactly in matrix form:

$$
\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}
$$

Or again as:

$$\mathbf{Ax} = \mathbf{b}$$

Note: matrix `A` is slightly different than defined previously. This is a more general form.

Example 2. Solve the following system of equations.

$$
\begin{aligned}
x_1 + 3x_2 + x_3 &= 1 \\
2x_1 + 2x_2 - x_3 &= 2 \\
x_1 + x_2 + x_3 &= -2
\end{aligned}
$$

```
% Step 1: convert to matrix format
```

```
A = [1,3,1;2,2,-1;1,1,1]; %Coefficient matrix defined above
b = [1;2;-2]; % Constant matrix
% Step 2: solve for x
```

Thus, the solution to this problem is $x_1 = -1.5,\ x_2 = 1.5,\ x_3 = -2$.

TOPIC 3: Systems of equations, functions, and more

ENGR105, 9/10/20

Array mathematics (continued)

Addition and Subtraction

General rules for addition and subtraction for arrays of any dimension (e.g. vectors or matrices)

1. Scalars may be added to/subtracted from any array - **the scalar value is added to all elements of the array**
2. Arrays added to arrays must have the **same dimensions** and elements in correlated index locations can be added or subtracted.

```
% Create new matrix
A = [1,5;2,3]
% Add 2 to every element
A + 2
% Create new array by reordering the first and second row
B = [A(2,:);A(1,:)]
% Matrix addition
A + B
```

Remember: matrix dimensions must agree. In other words, the matrices must have the same dimensions.

Multiplication

General rules for matrix/array multiplication:

1. Arrays may be multiplied by a **scalar**.
2. **Arrays of the same dimensions** may be multiplied index-by-index using the array multiplier/vectorization (`.*`)
3. The matrix product (*) may be evaluated for matrices if the "inner dimensions" of the original matrices match and the resulting matrix will have dimensions equal to the outer dimensions of the matrices being multiplied.

Case 1: multiplication by a scalar

```
% Create same matrices as last time
```

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

```
A = 2×2
     1     5
     2     3
```

```
B = [A(2,:);A(1,:)]
```

```
B = 2×2
     2     3
     1     5
```

```
A
```

```
A = 2×2
     1     5
     2     3
```

```
A*5
```

```
ans = 2×2
     5    25
    10    15
```

Case 2: element-by-element multiplication. Arrays of the same dimensions may be multiplied index-by-index using the array multiplier/vectorization (.*)

```
A
```

```
A = 2×2
     1     5
     2     3
```

```
B
```

```
B = 2×2
     2     3
     1     5
```

```
A.*B
```

```
ans = 2×2
     2    15
     2    15
```

Case 3: matrix multiplication. For `A*B`, the **columns of A must equal the rows of B**.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

The element in the <u>first</u> row and <u>first</u> column of `A*B` is:

$$A * B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & ? \\ ? & ? \end{bmatrix}$$

The element in the <u>first</u> row and <u>second</u> column of `A*B` is:

$$A * B = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$
$$= \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ ? & ? \end{bmatrix}$$

Filling in the matrix:

$$A * B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

```
A = [1,4;2,3];
B = [1,2;3,4];
% Matrix multiplication
A*B
```

```
ans = 2×2
```

```
    13    18
    11    16
```

Remember: A*B is not usually the same as B*A

```
% Matrix multiplication with multiplication order reversed.
B*A
```

```
ans = 2×2
     5    10
    11    24
```

Pro tip: if you feel stressed about this, write out each entry systematically! You will work a lot more with matrices in your linear algebra courses.

Remember: the dimensions of the matrix product are equal to the outer dimensions of the original matrices <u>and</u> the inner dimensions must match

$$\overset{1 \times 3}{A = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}} \quad \text{and} \quad \overset{3 \times 2}{B = \begin{bmatrix} 2 & 1 \\ 5 & 3 \\ 4 & 6 \end{bmatrix}}$$

MATLAB syntax

$$C = A * B \Rightarrow$$

$$\begin{bmatrix} 1*2+2*5+3*4 & 1*1+2*3+3*6 \end{bmatrix}$$

$$(1 \times 3) \times (3 \times 2) \rightarrow 1 \times 2 \text{ output matrix}$$

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

```
B = 3×2
     2     1
     5     3
     4     6
```

```
A = 1:3
```

```
A = 1×3
     1     2     3
```

```
% To perform matrix multiplication, ensure the columns of the first matrix
% match the rows of the second
A*B
```

```
ans = 1×2
    24    25
```

Division

Division of a scalar by an array necessitates an array operator.

```
A = [1 2; 4 8]
```

```
A = 2×2
     1     2
     4     8
```

```
B = 16./A
```

```
B = 2×2
    16     8
     4     2
```

NOTE: `B = 16/A` will return an error.

NOTE: Matrix division has a different interpretation than scalar division. Right division is different than left division.

Element-wise division:

- *Right* division: `A./B` is equivalent to `A(i,j)/B(i,j)`
- *Left* division: `A.\B` is equivalent to `B(i,j)/A(i,j)`

```
% Define matrix A and vector b
A = [2,4;2,6];
B = [1,2;1,3];
% Perform right division
A./B
```

```
ans = 2×2
     2     2
     2     2
```

```
% Perform left division
A.\B
```

```
ans = 2×2
    0.5000    0.5000
    0.5000    0.5000
```

Left matrix division. Consider the following matrix equation, in which matrix **A** is multiplied by vector **x** to give vector **b**.

$$\mathbf{Ax} = \mathbf{b}$$

<u>Note on dimensions</u>: assuming A is square (e.g. `2x2`), then **x** and **b** must be (e.g. `2x1`) column vectors. To solve for **x**, you must perform <u>left division</u> of matrix **A**, or left multiply by the inverse of **A**.

$$\mathbf{x} = \mathbf{A}\backslash\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}\backslash\mathbf{b} = \mathbf{A}^{-1}\mathbf{b}$$

The above works because $\mathbf{A}^{-1}\mathbf{A} = \mathrm{I}$, where I is the identity matrix, and $\mathbf{IX} = \mathbf{X}$ (by definition).

```
% Define matrix A and vector b
A = [2,4;2,6]
```

```
A = 2×2
     2     4
     2     6
```

```
b = [5;4]
```

```
b = 2×1
     5
     4
```

```
% Perform left division
A\b
```

```
ans = 2×1
    3.5000
   -0.5000
```

```
inv(A)*b
```

```
ans = 2×1
    3.5000
   -0.5000
```

Right matrix division. Consider the following matrix equation, in which matrix **A** is multiplied by vector **x** to give vector **b**.

$$\mathbf{xA} = \mathbf{b}$$

Note on dimensions: assuming A is square (e.g. `2x2`), then **x** and **b** must be (`e.g. 1x2`) row vectors. To solve for **x**, you must perform right division of matrix **A**, or right multiply by the inverse of **A**.

$$\mathbf{x} = \mathbf{xA}/\mathbf{A} = \mathbf{xAA}^{-1} = \mathbf{b}/\mathbf{A} = \mathbf{bA}^{-1}$$

```
A = [2,4;2,6]
```

```
A = 2×2
     2     4
     2     6
```

```
b = [5,4]
```

```
b = 1×2
     5     4
```

```
% Perform right division
b/A
```

```
ans = 1×2
    5.5000   -3.0000
```

```
b*inv(A)
```

```
ans = 1×2
    5.5000   -3.0000
```

Application: solving a set of linear equations

You are likely familiar with canonical linear equations of the form `y=mx+b` or `ax+by=c`, which define a straight line in 2D space. Consider a set of two linear equations:

$$y = m_1 x + b_1$$
$$y = m_2 x + b_2$$

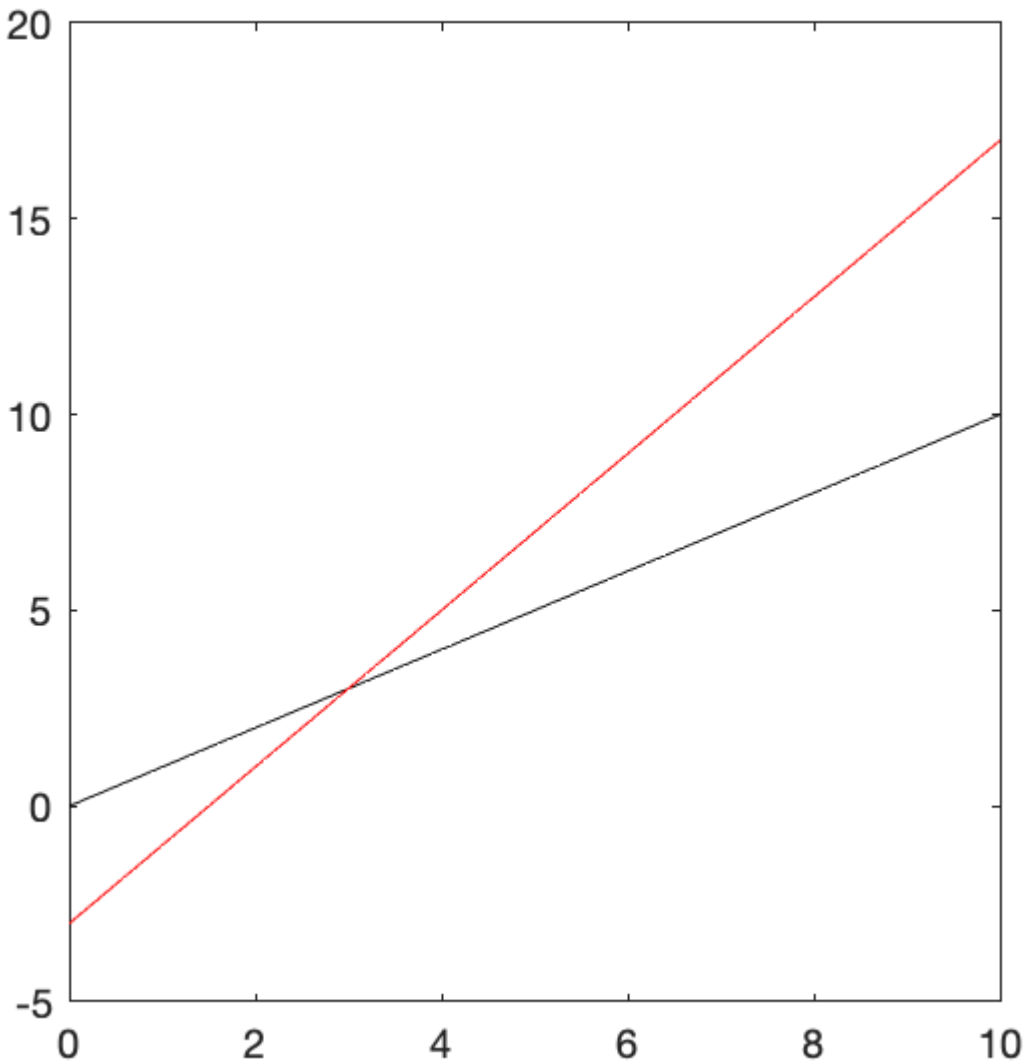
In engineering applications, you are often interested in solving a set of linear equations. There are three possibilities:

There is one solution.

$$y = x$$
$$y = 2x - 3$$

```
% x vectors
x = 0:.1:10;
% Create a figure window of a defined size
figure('Position',[0,0,300,300]);
```

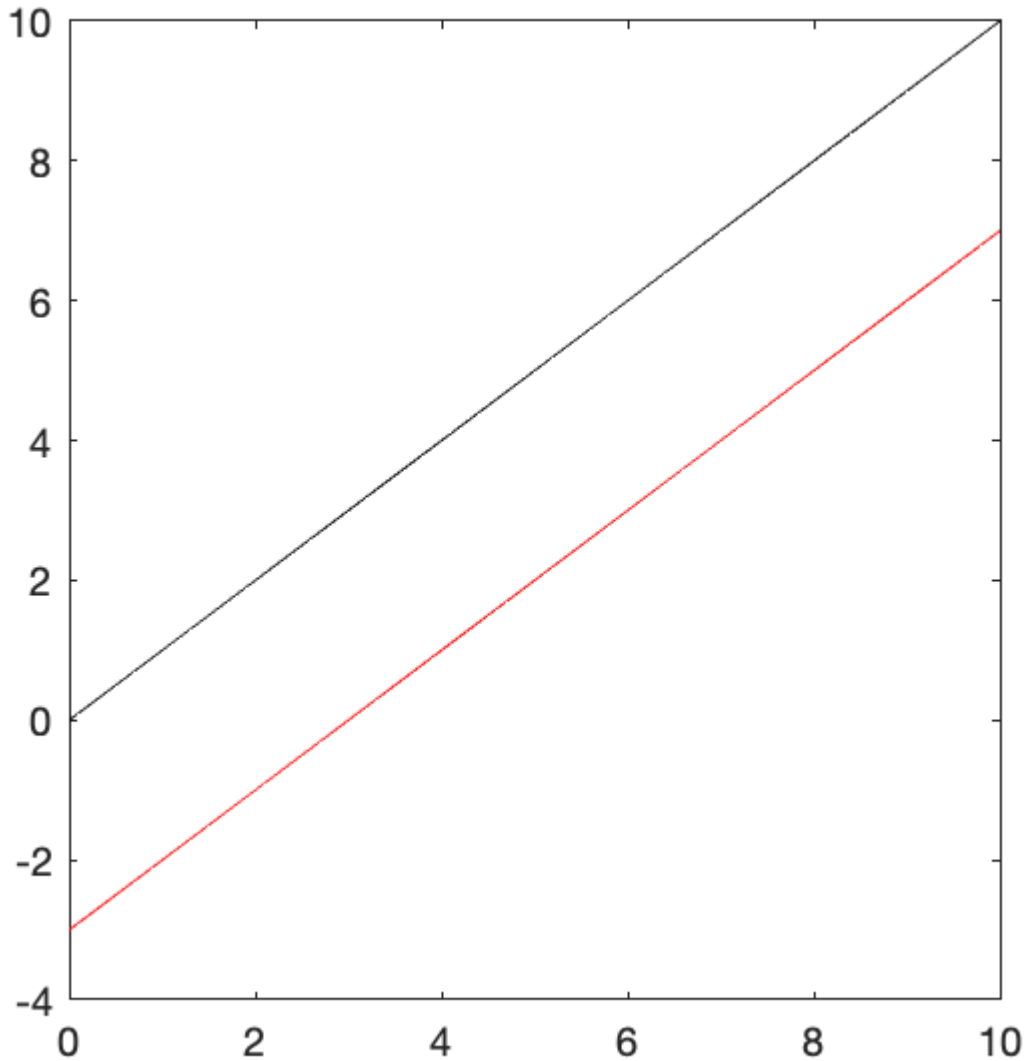
```
plot(x,x,'k-',x,2*x-3,'r-')
```



There are no solutions.

$$y = x$$
$$y = x - 3$$

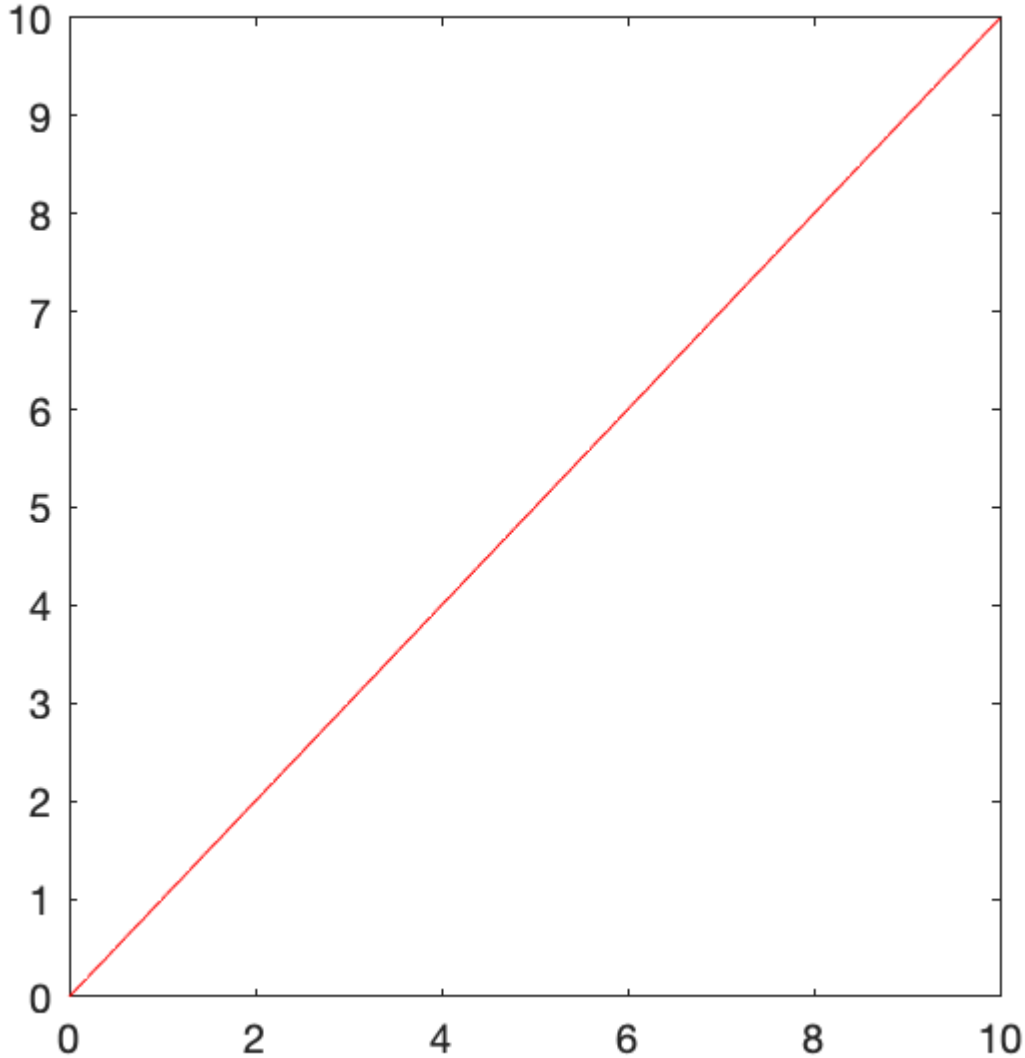
```
% Create a figure window of a defined size
figure('Position',[0,0,300,300]);
plot(x,x,'k-',x,x-3,'r-')
```



There are infinite solutions.

$$y = x$$
$$y = x$$

```
% Create a figure window of a defined size
figure('Position',[0,0,300,300]);
plot(x,x,'k-',x,x,'r-')
```



We will focus on systems that have <u>one solution</u>. Further study in linear algebra will provide conditions on matrix **A** and vector **b** that will enable you to determine the number of solutions.

One approach: substitution

Consider the two equations defined above. To solve, perform the following steps.

$$y = m_1 x + b_1$$
$$y = m_2 x + b_2$$

1) Set equations equal

$$m_1 x + b_1 = m_2 x + b_2$$

2) Solve for x

$$x(m_1 - m_2) = b_2 - b_1$$
$$x = \frac{(b_2 - b_1)}{(m_1 - m_2)}$$

3) Substitute to solve for y

$$y = m_1 \frac{(b_2 - b_1)}{(m_1 - m_2)} + b_1$$

This is fine, but requires equations in the above form and can get tedious.

Another approach: matrix representation

Rearranging the above system of equations:

$$y - m_1 x = b_1$$
$$y - m_2 x = b_2$$

Double check the matrix multiplication rules above. Based on the rules of matrix multiplication, the above set of equations can be represented in matrix form:

$$\begin{bmatrix} 1 & -m_1 \\ 1 & -m_2 \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Or:

$$\mathbf{Ax} = \mathbf{b}$$
$$\mathbf{A} = \begin{bmatrix} 1 & -m_1 \\ 1 & -m_2 \end{bmatrix}$$
$$\mathbf{x} = \begin{bmatrix} y \\ x \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

This is the case above corresponding to **left matrix division**. Thus, we can simply solve for y and x by performing a left division:

$$\mathbf{x} = \mathbf{A} \backslash \mathbf{Ax} = \mathbf{A} \backslash \mathbf{b}$$

Example 1. Solve the following system of equations.

$$y = x$$
$$y = 2x - 3$$

```
% Step 1: convert to matrix format
```

```
A = [1,-1;1,-2] %Coefficient matrix defined above
```

```
A = 2×2
     1    -1
     1    -2
```

```
b = [0;-3] % Constant matrix
```

```
b = 2×1
     0
    -3
```

```
% Step 2: solve for x and y
```

Thus, the solution exists at `x = 3` and `y = 3`.

More general representation

More often, you will encounter systems with more than two variable. Thus, it is more convenient to think of a system of linear equations in the following form, where there are `n` variables and `n` equations.

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
&\vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

The above can be represented compactly in matrix form:

$$
\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}
$$

Or again as:

$$\mathbf{Ax} = \mathbf{b}$$

Note: matrix `A` is slightly different than defined previously. This is a more general form.

Example 2. Solve the following system of equations.

$$x_1 + 3x_2 + x_3 = 1$$
$$2x_1 + 2x_2 - x_3 = 2$$
$$x_1 + x_2 + x_3 = -2$$

```
% Step 1: convert to matrix format
A = [1,3,1;2,2,-1;1,1,1]; %Coefficient matrix defined above
b = [1;2;-2]; % Constant matrix
% Step 2: solve for x
x = A\b
```

```
x = 3×1
   -1.5000
    1.5000
   -2.0000
```

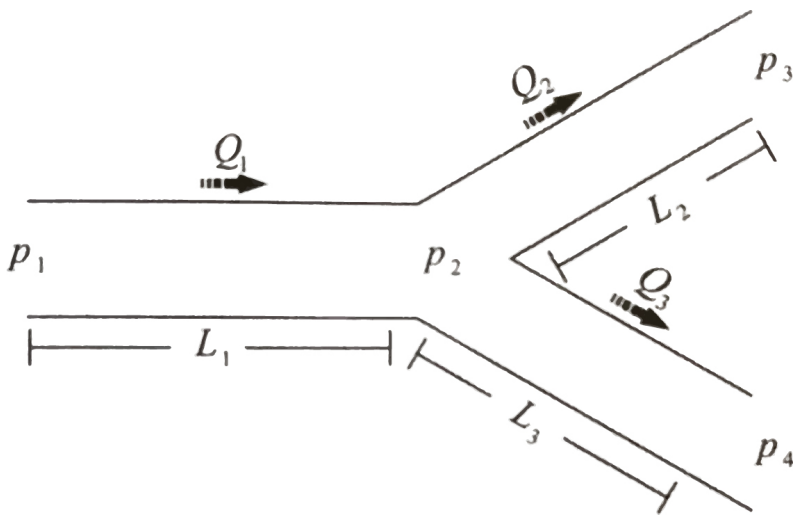
Thus, the solution to this problem is $x_1 = -1.5,\ x_2 = 1.5,\ x_3 = -2$.

Example application: fluid flow in a Y-channel

As an engineer, you might be interested in designing fluid flow in different systems. You will learn how to analyze these systems in your fluid mechanics and transport phenomena courses.

- *Relevant UPenn Course*: BE 350 Introduction to Biotransport Processes
- *Relevant UPenn Course*: CBE 350 Fluid Mechanics
- *Relevant UPenn Course*: MEAM 302 Fluid Mechanics

Consider the Y-junction in the following schematic. Since this system is physical, you the geometry of the channels and the input and output pressures. Specifically, all channels have a circular cross section with radius $R = 20\ \mu m$. Channel lengths are $L_1 = 0.5\ \text{cm},\ L_2 = 1\ \text{cm},\ L_3 = 2\ \text{cm}$. Pressures at the ports are $p_1 = 110\ \text{kPa},\ p_2 = 100\ \text{kPa}$, and $p_3 = 100\ \text{kPa}$. The fluid is water with viscosity $\eta = 1\text{x}10^{-3}\ \text{Pa} \cdot \text{s}$.



You are interested in determining the flow rates through each fluidic segment. From you (to be completed, future) engineering coursework, you know that you can describe this system as incompressible flow using Hagen-Poiseuille relations incorporating the equivalent hydraulic resistance. In this system, the hydraulic resistance of channel segment can be defined as follows:

$$R_i = \frac{8\eta L}{\pi R^4}$$

Step 1: write down the physical parameters you know and draw a schematic.

```
% Known physical parameters in this system
eta = 1*10^-3; %viscosity, Pa*s
L1 = 0.5*10^-2; % m
L2 = 1*10^-2; % m
L3 = 2*10^-2; % m
R = 20*10^-6; % m

% Calculate fluidic resistance of each channel segment
R1 = 8*eta*L1/(pi*R^4)*10^-12;   % kPa*s/mm^3
R2 = 8*eta*L2/(pi*R^4)*10^-12;   % kPa*s/mm^3
R3 = 8*eta*L3/(pi*R^4)*10^-12;   % kPa*s/mm^3
```

Step 2: state assumptions and write down equations describing the system. In this case, we are assuming steady, incompressible flow, yielding the following equations:

$$p_1 - p_2 = Q_1 R_1$$
$$p_2 - p_3 = Q_2 R_2$$
$$p_2 - p_4 = Q_3 R_3$$

Because the system is incompressible, we know from conservation of mass:

$$Q_1 = Q_2 + Q_3$$

In addition, we know the input pressure (perhaps supplied by a pump) and the output pressures (atmospheric pressure):

$$p_1 = 110\,\text{kPa}$$
$$p_3 = 100\,\text{kPa}$$
$$p_4 = 100\,\text{kPa}$$

Step 3: solve system of equations. Here, we have 7 variables and 7 equations. We can solve the problem using the matrix representation approach by first writing all of the equations in the form $a_{11}x_1 + a_{12}x_2 + \ldots + a_{13}x_3 = c_1$.

$$p_1 = 110$$
$$p_3 = 100$$
$$p_4 = 100$$
$$p_1 - p_2 - Q_1R_1 = 0$$
$$p_2 - p_3 - Q_2R_2 = 0$$
$$p_2 - p_4 - Q_3R_3 = 0$$
$$Q_1 - Q_2 - Q_3 = 0$$

Converting this a matrix (**NOTE: I have taken care to ensure all units are properly matched**):

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & -R_1 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & -R_2 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & -R_3 \\ 0 & 0 & 0 & 0 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ Q_1 \\ Q_2 \\ Q_3 \end{bmatrix} = \begin{bmatrix} 110 \\ 100 \\ 100 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This can be solved using the matrix division we talked about last time.

$$\mathbf{x} = \mathbf{A}\backslash\mathbf{A}\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$$

```
 clear A b   % Clear previously used terms from workspace
 % Define matrix A
 A(4,1:7)=[1,-1,0,0,-R1,0,0];    % We specify columns 1:7 because the matrix A
is not yet defined
 A(5,:)=[0,1,-1,0,0,-R2,0];
 A(6,:)=[0,1,0,-1,0,0,-R3];
 A(7,:)=[zeros(1,4),1,-1,-1];
 A([1,16,24]) = 1;               % Use single index notation to specify sparse 1
values
 disp(A)
```

```
    1.0000         0         0         0         0         0         0
         0         0    1.0000         0         0         0         0
         0         0         0    1.0000         0         0         0
    1.0000   -1.0000         0         0  -79.5775         0         0
         0    1.0000   -1.0000         0         0 -159.1549         0
         0    1.0000         0   -1.0000         0         0 -318.3099
         0         0         0         0    1.0000   -1.0000   -1.0000
```

```
% Define matrix b
b = [110,100,100,zeros(1,4)]';  % Note the transpose: we need b to be a column
vector
disp(b)
```

```
   110
   100
   100
     0
     0
     0
     0
```

```
% Solve system of equations
x = A\b;
disp(x)
```

```
  110.0000
  105.7143
  100.0000
  100.0000
    0.0539
    0.0359
    0.0180
```

Thus, in one line of code, we obtain the solution to this set of 7 linear equations. In this case, the solution is as follows.

$$
\begin{aligned}
p_1 &= 110\,\mathrm{kPa} \\
p_2 &= 106\,\mathrm{kPa} \\
p_3 &= 100\,\mathrm{kPa} \\
p_4 &= 100\,\mathrm{kPa} \\
Q_1 &= 0.054\,\mathrm{mm}^3/\mathrm{s} \\
Q_2 &= 0.036\,\mathrm{mm}^3/\mathrm{s} \\
Q_3 &= 0.018\,\mathrm{mm}^3/\mathrm{s}
\end{aligned}
$$

Side challenge: can you create matrix A, above, in four lines of code or less?

Functions

Recall: assemblages of code can be composed in the editor window and stored as an .m file for more permanence. These assemblages of code are referred to as either functions or scripts. Functions start with the `function` declaration whereas scripts are just assemblages of statements.

Functions vs. scripts

- The following refers to .m files created to contain a function or a script.

Function

```
function c = Pythagorean(a,b)
% Compute c for the relation
% a^2+b^2=c^2.
% a, b, and c are scalars.

% Compute c.
c = sqrt(a^2+b^2);
end
```

Script

```
% Pythagorean_script.m

clear, clc, close all

% Define values a and b.
a = 5;
b = 17;

% Compute c.
c = sqrt(a^2+b^2)
```

Features	Scripts	Function
File type	.m file	.m file
m file naming conventions	Any valid MATLAB filename	Named after the function name
Interfacing	Editor window	Editor window
How to invoke	Command window, other scripts, or other functions	Command window, other scripts, or other functions
Typical uses	Single use code or initial function development	Multiple use code and/or part of a larger code base
Memory	Variables stored in user workspace – can cause variable conflicts	Variables stored in function workspace and **only** accessible via explicit inputs/outputs
Input management	Use of the `clear` function is recommended and inputs are usually hard-coded	`clear` function should not be used and inputs are generalized

"Where" do we create functions?

- To run on the command line, you need to create a function in a new `.m` file.
- When using the LiveScript, each individual function is saved at the end of the `.mlx` document.

Examples of function declarations

Note: function **declarations** go at the top of a `.m` file and may only be preceded by comments.

- No inputs and no outputs

```
function myFunc or function myFunc()
```

- One input and one output

```
function out1 = myFunc(in1)
```

- Multiple outputs and no inputs

```
function [out1,out2,out3] = myFunc()
```

- Three outputs and two inputs

```
function [out1,out2,out3] = myFunc(in1,in2)
```

NOTE: the number of inputs does not need to match the number of outputs.

Function rules

1. When creating an `.m` file to contain the function, name the `.m` file the same as the function name.
2. Do not use a variable within a function that has the same name as the function.
3. Do not hard-code inputs.
4. Use only one function per .m file
5. All function outputs should be calculated at some point within the function
6. Use `end` at the conclusion of a function.
7. For ease of visualizing, indent the contents of a function.

Do not hard-code inputs. A poor example of a generalized function

```
function out = mySquareRoot(a)
    % Returns the square root of a scalar,
    % vector, or multidimensional array
    % input, a, to out

    % Defining the input a
    a = 5;

    % Calculates the square root
    % of the input
    out = sqrt(a);
end
```

The purpose of a function is to take in variables and manipulate them according to certain rules. The function will always return the same (usually incorrect) result if you redefine the input variables inside the function.

Variable names only matter to the function.

- Names matter within a function.

```
function [p,s] = myFunction(a,b,c)
    % p and s are computed within the function
    p = a+b;
    s = b*c;
end
```

- Order of inputs matters outside the function

```
% call_myFunction.m
% Define variables
m = 2;
n = 3;
q = 4;
[g,h] = myFunction(m,n,q);
```

```
% Order of input matters when calling a function
m = 2;
n = 3;
q = 4;
[g1,h1] = myFunction(m,n,q)
```

```
g1 = 5
h1 = 12
```

```
[g2,h2] = myFunction(q,m,n)
```

```
g2 = 6
h2 = 6
```

Note: "**what happens in a function, stays in a function** *(unless passed as an input or output)*." If we were to call `p` or `s`, MATLAB would give an error because those variables are only stored in the local function memory. Unless you specify a variable to be returned as an output, you cannot call it from the command line.

```
p
```

```
Unrecognized function or variable 'p'.
```

Function headers

It is customary to describe the purpose, inputs, outputs, author, and creation/modification date of a function right after the function definition.

```
function out = mySquareRoot(a)
    % Returns the square root of a scalar,
    % vector, or multidimensional array
    % input, a, to out
    % M. Siedlik
    % 9/6/2020

    % Calculates the square root
    % of the inputout = sqrt(a);
    out = sqrt(a);
```

```
end
```

It is a best practive to describe units and/or list inputs/outputs with specific descriptions of their assumptions and limitations. In general, **more (clear) information is better**.

- The first line of a function is referred to as the "H1 line" and is searchable by MATLAB using `lookfor`.
- The first contiguous block of text is returned as the help query for the function.

Function naming conventions

1. Must consist of only the letters a-z, the digits 0-9, and underscores (_)
2. Must start with a letter
3. The filename of functions should adopt the function name. The following would be saved as `myFunc.m`.

```
function output = myFunc(input1, input2)
```

NOTE: Camel caps and underscores are easier to distinguish than long strings of lowercase characters

- `piratePathA` and `pirate_path_A` are easier to interpret than `piratepatha`

Programmers often start low level functions with a lowercase character - e.g. `supportFunction` or `doSmallThing`

Example

Create a function to output the odd entries of an input vector.

```
% Define a vector to input into the function
u = linspace(1,10,8)
```

```
u = 1×8
    1.0000    2.2857    3.5714    4.8571    6.1429    7.4286    8.7143
10.0000
```

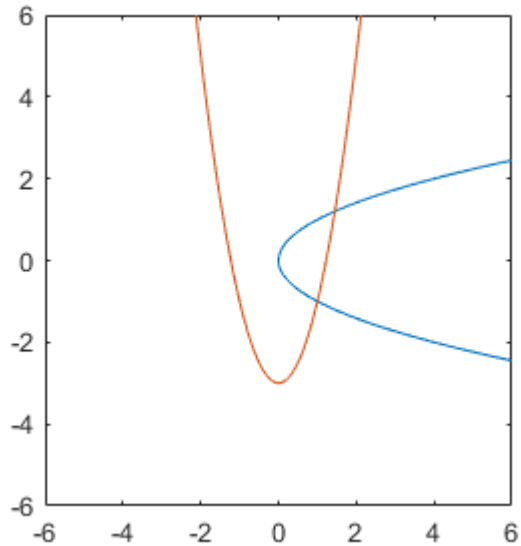
```
% Call the function and store output as variable.
v_odd = returnOddElements(u)
```

```
v_odd = 1×4
    1.0000    3.5714    6.1429    8.7143
```

Example: systems of nonlinear systems of equations

Nonlinear systems have many solution methods based on the nature of the equations and the application. For example, to solve the following set of nonlinear equations.

$$y^2 = x$$
$$y = 2x^2 - 3$$



- We can invoke MATLAB's nonlinear equation solver `fsolve()`, which demands equations of the form `F(x,y...)=0`
- Rearranging the equations to this form:

$$y^2 - x = 0$$
$$y - 2x^2 + 3 = 0$$

- Renaming the variables for convenience:

$${x_2}^2 - x_1 = 0$$
$$x_2 - 2{x_1}^2 + 3 = 0$$

To solve, we first define a <u>function</u> that contains a description of the equations that will eventually be called by `fsolve()`.

```
function F = eqnSet(x)
    % System of two non linear equations
    % F(1) and F(2): "equation" 1 and 2, respectively
    % x(1) and x(2): variable 1 and 2, respectively
    F(1) = x(2)-2*x(1)^2+3;
    F(2) = x(2)^2-x(1);
end
```

Note: the `0`'s have been replaced by `F`'s and mathematical subscripts have been changed to indexing subscripts

The set of equations can now be solved using `fsolve()`- this code is contained in a function block in this LiveScript document or in a separate m-file.

- Since we have two valid solutions (which we know because we plotted the functions first), we must call `fsolve()` twice.
- In each case we will provide different initial guess conditions that promote convergence to the two different solutions.

```
% Determine solution 1
sol_1 = fsolve(@eqnSet,[0,0])
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
sol_1 = 1×2
    1.0000   -1.0000
```

Notes on the above code:

- `sol_1`: a 1x2 vector containing the first intersection point (*x*and *y*, respectively).
- `@eqnSet`: the first input into the fsolve(), this tells the function where the equations are located.
- `[0,0]`:the second input in fsolve(), this specifies our initial guess as to where the first intersection is located.
- `fsolve()` will attempt to converge to a solution until some stopping criteria (threshold) is met.

```
% Determine solution 2
sol_2 = fsolve(@eqnSet,[1,1])
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
sol_2 = 1×2
    1.4498    1.2041
```

NOTE: the `@` symbol is called a function handle and is a way to pass one function into another function. We will discuss this more later.

TOPIC 4: Loops and logic

ENGR105, 9/15/20

Introductory remarks

We have accomplished a lot so far -- we've created arrays of varying sizes, manipulated the entries in these arrays through indexing and array mathematics, and learned how to package collections of code into functions.

We've worked a lot with **vectors**, in particular. We have created *row* or *column* vectors through explicitly defining entries, using slice/colon (`:`) notation to create vectors with a defined increment size between entry values, using `linspace()` to create vectors with a defined number of increments, and so forth.

```
% Explicitly define a vector
a = [5,3,2,6,2,1]
```

```
a = 1×6
     5     3     2     6     2     1
```

```
% Create a vector with entries starting at 1 and incrementing by 2 until 9
b = 1:2:9
```

```
b = 1×5
     1     3     5     7     9
```

```
% Create a vector consisting of 5 evenly spaced values between (and
% including) 1 and 8;
c = linspace(1,8,5)
```

```
c = 1×5
    1.0000    2.7500    4.5000    6.2500    8.0000
```

Example - Fibonacci sequence

The above methods are very efficient and should be used when possible. However, in other cases we may wish to create vectors with entries defined by *more sophisticated patterns*. For example, consider the Fibonacci sequence, an infinite sequence defined by the following rules, that you may recall from previous math classes or popular culture.

$$
\begin{aligned}
x(1) &= 0 \\
x(2) &= 1 \\
x(3) &= x(2) + x(1) \\
&\vdots \\
x(n) &= x(n-1) + x(n-2)
\end{aligned}
$$

Using what we've learned so far, **create a vector `x` containing the first 10 elements of the Fibonacci sequence**.

```
% The first two entries are explicitly defined
x(1) = 0;
x(2) = 1;
```

```
% The next eight entries must be calculated individually based upon the
% previous two values
x(3) = x(1) + x(2);
x(4) = x(2) + x(3);
x(5) = x(3) + x(4);
x(6) = x(4) + x(5);
x(7) = x(5) + x(6);
x(8) = x(6) + x(7);
x(9) = x(7) + x(8);
x(10) = x(8) + x(9);
disp(x)
```

```
     0
     1
     1
     2
     3
     5
     8
    13
    21
    34
```

Or alternatively, by doing the math "in your head" and explicitly defining a vector:

```
% Explicitly define a fibonacci vector, x
x = [0,1,1,2,3,5,8,13,21,34]
```

```
x = 1×10
     0     1     1     2     3     5     8    13    21    34
```

In both cases, this process was extremely tedious and impractical for creating larger vectors or those with more complicated rules.

Note: always proofread your code before running (and submitting an assignment!). Just like when writing a paper, it is easy to make "typos" that could dramatically affect the output of your code.

FOR loops

Fortunately, loops allow us to more easily repeat statements, repeatedly invoke functions, or manipulate multidimensional data. The first type of loop we will discuss is a **`for`** loop, which enable us to repeat tasks **for** a **fixed number of iterations**. A **`for`** loop has the following form:

```
for index = <values>
    <statements>
end
```

```
% Iterate for k equaling each value in the row vector
% [1,2,3,4,5,6,7,8,9,10]
for k = 1:10
    % For each iteration of the for loop, display the value of the index,
```

```
    % k, squared.
    disp(k^2)
end
```

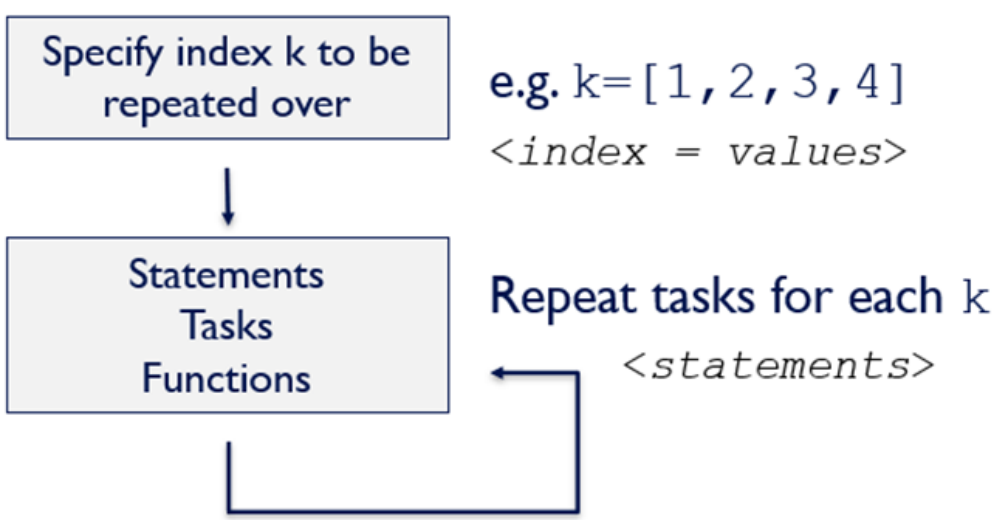
```
     1

     4

     9

    16

    25

    36

    49

    64

    81

   100
```

Anatomy of a `for` loop

- The **`for`** command indicates that the statements to follow should be repeated
- The variable **`index`** is the loop counter that sequentially takes the values contained in a row vector **`<values>`**.
- Any number of **statements** (i.e. productive code) may be contained within the loop and will be executed each time the **`for`** loop iterates. **The statement operations often depend on the index**, k.
- The **`end`** statement returns control to **`for`** until **`index`** has iterated through all entries in the **`<values>`** vector.

Note: we indent the contents of the for loop for ease of visualization.

The procedure that occurs when a for loop is called may also be explained visually:



In other words, for loops are used when the number of repeats are known a priori.

Note: There is nothing special about the row vector `<values>` - it may be created in any of the ways we have discussed so far. The most common scenario involves iterating from index 1 by 1 to some end value by specifiying the colon (`:`) operator.

The following are examples of using for loops to create a new vector.

1) Defining entries *by the index of the for loop*.

```
% jj takes the values of 1 in the first iteration, 2 in the second
% iteration, etc.
for jj = 1:5
    % The value of the computation 2*jj is stored in the jj-th location of
    % vector v, which grows during each iteration of the for loop
    v(jj) = 2*jj
end
```

```
v = 2
v = 1×2
     2     4
v = 1×3
     2     4     6
v = 1×4
     2     4     6     8
v = 1×5
     2     4     6     8    10
```

```
disp(v)
```

```
     2     4     6     8    10
```

Note: The above example is simply a more compact way of doing the vector operations that we have already mastered.

2) Indexing a generalized vector using a `for` loop and `length()`.

General vectors may be fully parsed by creating a loop counter based on the length of the vector

```
% An arbitrary vector
v0 = [5, -2, 4, 1];
for jj = 1:length(v0)
    % In the jj-th position of v1, store the value at the jj-th position of
    % v0 multiplied by 2.
   v1(jj) = 2*v0(jj)
end
```

```
v1 = 10
v1 = 1×2
    10    -4
```

```
v1 = 1×3
    10    -4     8
v1 = 1×4
    10    -4     8     2
```

```
disp(v0)
```

```
     5    -2     4     1
```

```
disp(v1)
```

```
    10    -4     8     2
```

In the above case, `length(v0)` is equivalent to 4.

3) Growing vectors using a `for` loop and `end+1` assignments

Recall vector operations using `end`:

```
v = 1:4;
% Return the last element of v
v(end)
```

```
ans = 4
```

```
v(end+1) = 1
```

```
v = 1×5
     1     2     3     4     1
```

The above approach may be used to sequentially populate an initially empty vector.

```
v = [];
%Vector v is initialized as an empty set and is grown by adding to its
%end-ith + 1 index within the for loop
for jj = 1:.1:2
    %jj takes the values of 1 in the first iteration, 1.1 in the second
    %iteration, 1.2 in the third iteration, etc. - note that non-integer
    %values cannot be used to index a vector
    v(end+1) = 2*jj;
end
v
```

```
v = 1×11
    2.0000    2.2000    2.4000    2.6000    2.8000    3.0000    3.2000
3.4000    3.6000    3.8000    4.0000
```

Note: non-integes values cannot be used to index a vector. Also, this examaple is for visualizing the **`for`** loop. A general rule-of-thumb is to <u>use vectorization when possible</u>. For example, the above output could be generated from **one line of code:**

```
v = 2*(1:.1:2)
```

```
v = 1×11
    2.0000    2.2000    2.4000    2.6000    2.8000    3.0000    3.2000
3.4000    3.6000    3.8000    4.0000
```

4) Indexing vectors using a `for` loop and incrementing counter.

In this alternate formulation, a counter is initialized/incremented in lieu of adding to the end + 1 index of vector v

```
n = 0;
% n is a counter the increments by 1 in every iteration of the for loop and
% identifies the vector index into which jj is placed
for jj = 1:.1:2
    %n is incremented by 1 in each iteration of the loop
    n = n+1; %counter
    v(n) = 2*jj;
end
v
```

```
v = 1×11
    2.0000    2.2000    2.4000    2.6000    2.8000    3.0000    3.2000
3.4000    3.6000    3.8000    4.0000
```

Additional (pecular) notes

```
for index = <values>
    <statements>
end
```

Recall that the vector to be looped through (`<values>` above) is a row vector. Technically index will take on every column entity in the `<values>` array.

- If `<values>` is a **column vector**, the loop will perform once and index will take on the entire column vector

```
%For the values it must take in a row verctor
for jj = (1:5)'
    jj
end
```

```
jj = 5×1
     1
```

```
     2
     3
     4
     5
```

Compared to the "normal case" of `<values>` being a row vector:

```
for jj = (1:5)
    jj
end
```

```
jj = 1
jj = 2
jj = 3
jj = 4
jj = 5
```

- If `<values>` is a matrix, the loop will iterate according to the number of columns and index will take on each column of the matrix as a column vector.

```
M = [5, 3; 1, 9]
```

```
M = 2×2
     5     3
     1     9
```

```
for jj = M
    jj
end
```

```
jj = 2×1
     5
     1
jj = 2×1
     3
     9
```

- If `<values>` is a matrix, and you want index to take on all entries of the matrix, you could use the unwinding command `(:)` mentioned previously. In this case, `M` is unwrapped and transposed to yield a row vector.

```
for jj = M(:)'
    jj
end
```

```
jj = 5
jj = 1
jj = 3
```

```
jj = 9
```

Nested `for` loops

Multiple `for` loops may used such that one for loop is run *inside* of another for loop, yielding ***nested for loops***. In the following example, the "inside" `for` loop runs to completion for each loop of the "outside" `for` loop.

```
v = [];
p = [];
for jj = 1:3
    for kk = 1:3
        v(end+1) = jj;
        p(end+1) = kk;
    end
end





disp(v)
```

```
     1     1     1     2     2     2     3     3     3
```

```
disp(p)
```

```
     1     2     3     1     2     3     1     2     3
```

In the above case, the nested `for` loop circulates 3 times before passing back to the next iteration of the higher level `for` loop.

Note: we indent the contents of each `for` loop for ease of visualization.

Example - back to the Fibonacci sequence

Previously, we wanted to create a vector containing the first 10 elements of the Fibonacci sequence.

$$
\begin{aligned}
x(1) &= 0 \\
x(2) &= 1 \\
x(3) &= x(2) + x(1) \\
&\vdots \\
x(n) &= x(n-1) + x(n-2)
\end{aligned}
$$

This can be done much more efficiently using a **`for`** loop.

```
% Populate an array containing the first ten numbers of the sequence.
n = 10;
% Empty the vector x - we used it previously
```

```
x = [];
x(1) = 0;
x(2) = 1;
% Create each subsequent entry by taking the sum of the previous two
% entries
for k = 3:n
    x(k) = x(k-1) + x(k-2);
end


% Display the vector
disp(x)
```

```
     0     1     1     2     3     5     8    13    21    34
```

Question: what if we want to display all numbers in the Fibonacci sequence that are less than 100? This is tedious because we do not know beforehand how many entries that corresponds to.

Takeaway: `for` loop are good when you know the number of iterations to perform. **`while`** loops are better when you do not know the number of repeats a priori.

WHILE loops

`while` loops are a type of iteration in which a sequence of instructions may be repeated as long as some condition is true. To understand `while` loops, we first need to discuss *relational and logic operators* in MATLAB.

In both cases, `true` is equivalent to `1` and `false` is equivalent to `0`.

Relational operators

Relational operators compare values (or vectors, matrices, arrays) and yield a `true` (`1`) or `false` (`0`) result.

Relational operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
~=	Not equal to
>	Larger than
>=	Larger than or equal to

```
% Examples
3>4;
```

```
3<4;
3<= 0;
3<3;
```

Logical operators

Logical operators are commands that yield a `true` (`1`) or `false` (`0`) result.

Logical operator	Meaning
~	Not, the opposite of
true	Logical 1, true
false	Logical 0, false

```
true
```

```
ans = logical
   1
```

```
~true
```

```
ans = logical
   0
```

```
true == ~false
```

```
ans = logical
   1
```

Examples

The logical state (`true` or `false`) can be stored in a variable, just as one stores a number.

```
% Logical statement examples



% Relational operators can also operate on variables
```

The `true`/`false` output may also be incorporated into mathematical expressions.

```
s = true>false
```

```
s = logical
   1
```

```
b = true + true
```

```
b = 2
```

```
c = 5 + (2==2)*5
```

```
c = 10
```

Anatomy of a `while` loop

`while` loops take the following form:

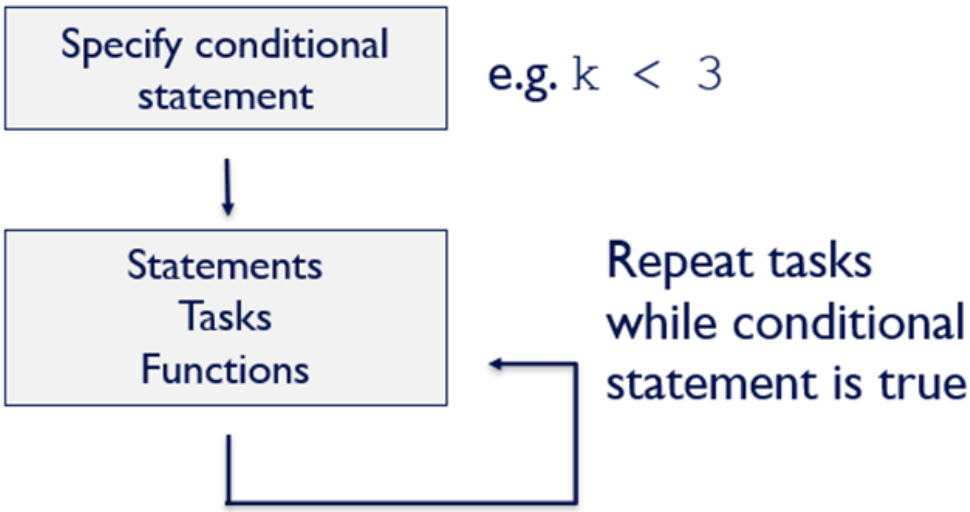
```
while <logical statement>
    <statements>
end
```

```
ind = 1;
% ind is incremented within the loop and terminates the while loop after 4
% iterations, when ind < 5 is no longer true.
while ind<5
    % The incremetnation of ind is not suppressed with a semicolon so that
    % you can see the line of code that are run in each iteration.
    ind = ind+1
end
```

```
ind = 2
ind = 3
ind = 4
ind = 5
```

- The `while` continues execution of the loop **as long as its `<logical statement>` evaluates to `true`.**
- the `<logical statement>` can be any set of numbers, relational operators, commands, or even functions that yield a logical output.
- Usually the `<logical statement>` is dependent upon the state of some variable that is modified in the `<statements>` within the loop.
- As with a `for` loop, any number of `<statements>` may be contained within the loop and will be executed each time the `while` loop iterates.
- The `end` statement returns control to `while` as long as `<logical statement>` is true.
- **<u>Warning</u>: if `<logical statement>` is always true, than your code will become an infinite loop!** And MATLAB might crash.

The procedure that occurs when a `while` loop is called may also be explained visually:



Example: simulating the roll of two dice

The following code is an example of simulation to observe the number of rolls until two dice are rolled with the same value.

```
% Variable to store the number of rolls until the same dice reading is
% observed
r2s = 0;
% Initialize while loop logical. As long as contRoll is true, we will
% "continue rolling"
```

Example: back again to the Fibonacci sequence

Previously, we wanted to create a vector containing all elements of the Fibonacci sequence less than 100.

$$
\begin{aligned}
x(1) &= 0 \\
x(2) &= 1 \\
x(3) &= x(2) + x(1) \\
&\vdots \\
x(n) &= x(n-1) + x(n-2)
\end{aligned}
$$

This can be done much more efficiently using a **`while`** loop.

```
% Max value of entries in Fibonacci sequence
n = 100;
% Define first two entries
x(1) = 0;
x(2) = 1;
% Iterate as long as the last element is less than 100.
```

Note: why do we remove the last element in this example? We remove it because the loop ends when the last element of the vector is greater than 100. Since we do not want *any* of the vector elements to be greater than 100, we must remove the last entry.

Vector preallocation

Preallocation is the process of creating a data structure (e.g. a vector) of final size (usually padded with zeros) before modifying the contents. This reduces the computational time by eliminating the operations associated with resizing the array.

Preallocation is a best practice to use whenever possible -- it is important if you use a `for` loop to populate a vector.

Consider the Fibonacci sequence example using the `for` loop, in which we populated a vector with the first 10 entries of the Fibonacci sequence.

```
% Populate an array containing the first ten numbers of the sequence.
n = 10;
% Empty the vector x - we used it previously
x = [];
x(1) = 0;
x(2) = 1;
% Create each subsequent entry by taking the sum of the previous two
% entries
for k = 3:n
    x(k) = x(k-1) + x(k-2);
end
```

Let's calculate the time required to produce this Fibonacci vector of *differing lengths, with* and *without* preallocation.

To perform this analysis, we are going to:

1. For sequence length, we will determine the time required to calculate the Fibonacci sequence, with and without preallocation
2. Repeat with a greater number of Fibonacci elements.
3. Plot *sequence (vector) length* vs. *computation time*.

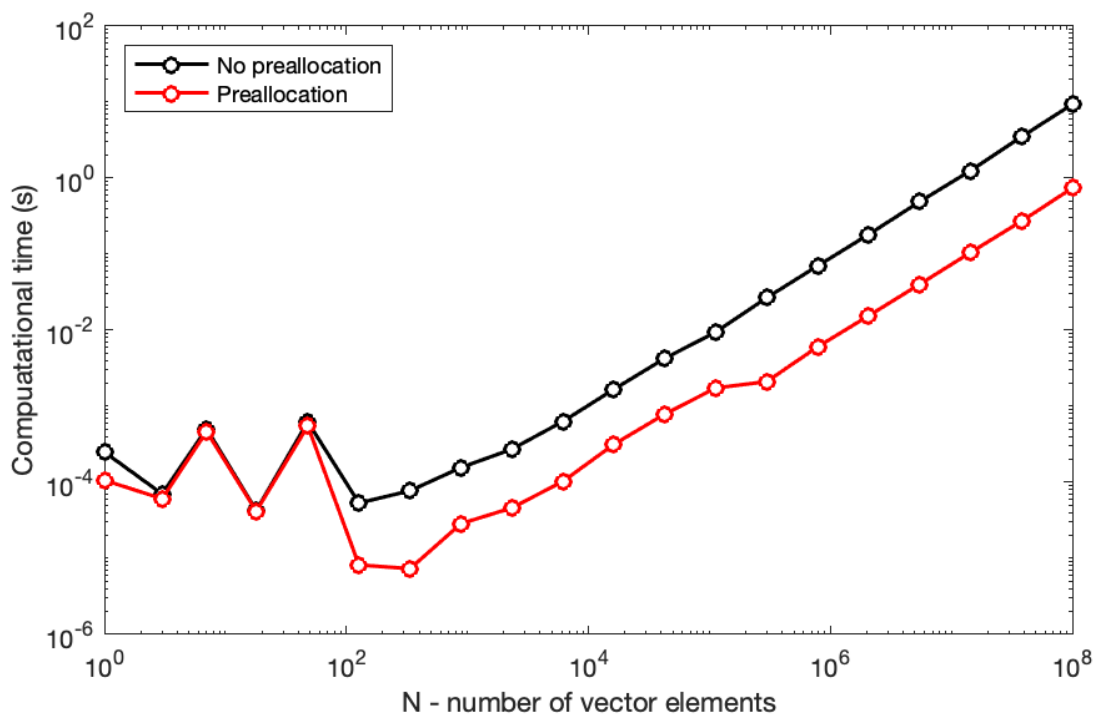
```
% Define vector containing the number of entries in each successive
% Fibonacci vector.
N = round(logspace(0,8,20))
```

```
N = 1×20
            1           3           7          18          48         127
          336         886        2336        6158       16238       42813
       112884      297635      784760     2069138     5455595    14384499
     37926902   100000000
```

```
% Preallocate vectors to store time information
t_no = zeros(1,length(N));
t_yes = zeros(1,length(N));
% Calculate the time for each vector of length N(m)
for m = 1:length(N)

    % Calculate Fibonacci sequence vector without preallocation
    x = [];
    tic
    % Define first two entries
    x(1) = 0;
    x(2) = 1;
    % Create each subsequent entry by taking the sum of the previous two
    % entries
    for k = 3:N(m)
     x(k) = x(k-1) + x(k-2);
    end
    t_no(m) = toc;

    % Calculate Fibonacci sequence vector with preallocation
    tic;
    x = zeros(1,N(m));
    % Define first two entries
    x(1) = 0;
    x(2) = 1;
    % Create each subsequent entry by taking the sum of the previous two
    % entries
    for k = 3:N(m)
     x(k) = x(k-1) + x(k-2);
    end
    t_yes(m) = toc;
end
```

Plotting the above simulation results. Note the addtions to the code to produce a more professional plot. We will discuss plotting in detail later in the semester.

```
% Define dimensions of plot
figure('Position',[0,0,500,300]);
% Plot the functions
plot(N,t_no,'ko-',N,t_yes,'ro-','LineWidth',1.5,'MarkerFaceColor','auto')
% Graphical parameters
set(gca, 'XScale', 'log')   % "log" axes - x
set(gca, 'YScale', 'log')   % "log" axes - y
set(gca,'XMinorTick','on')
xlim([10^0,10^8]);  % range of x values on plot
ylim([10^-6,10^2]); % range of y values on plot
xlabel('N - number of vector elements');
ylabel('Compuatational time (s)');
legend('No preallocation','Preallocation','Location','northwest')
```



Major takeaway: you can save an **order of magnitude** of computational time using **preallocation**.

For loops vs. vectorization

As described above, loops can be very useful for repeating many tasks a known or unknown number of times. In the case of for loops, we performed examples that could also be accomplished with vectorization, *i.e.* operating on a vector. Which of these approaches is better? To evaluate this, let's compare the computational time for each.

Specifically, let's calculate `b(k) = a(k)*a(k)`, where a is some vector. That is, we want to create a vector b in which each entry in the vector is the square of the corresponding entry in the vector a.

1) Using a `for` loop.

```
for k = 1:length(a)
    b(k) = a(k)*a(k);
end
```

2) Vectorization (e.g. using an element-wise operation)

```
b = a.^2;
b = a.*a;
```

To perform this analysis, we are going to:

1. Create a vector of a known length.
2. Calculate the time required to perform the above operation on that vector using either a for loop or with vectorization.
3. Repeat with a longer vector.
4. Plot *vector length* vs. *computation time*.

```
% Define vector containing number of entries in potential vector.
N = round(logspace(0,8,20))
```

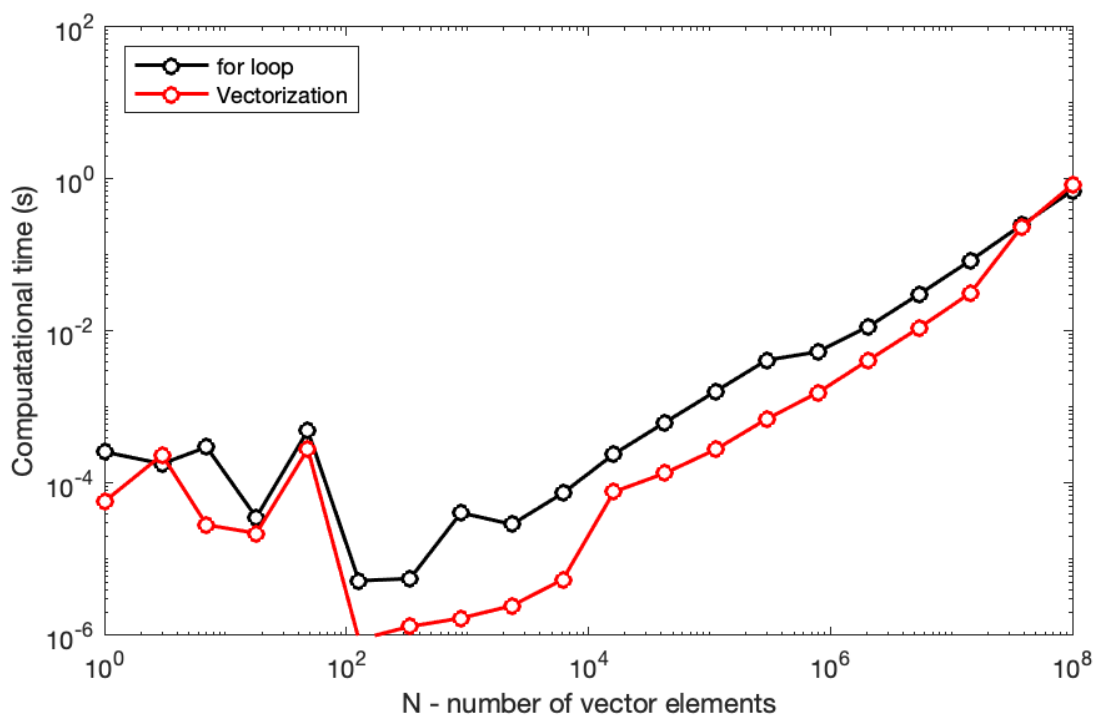
```
N = 1×20
           1           3           7          18          48         127
         336         886        2336        6158       16238       42813
      112884      297635      784760     2069138     5455595    14384499
    37926902   100000000
```

```
% Preallocate vectors to store information
t_for = zeros(1,length(N));
t_vec = zeros(1,length(N));
% Calculate the time for each vector of length N(m)
for m = 1:length(N)
    % Create vector of random numbers with dimension 1xN(m)
    a = rand(1,N(m));

    % Calculate time for for loop
    tic
    % Preallocate for loop vector
    b_for = zeros(1,N(m));
    % Calculate each entry in b by looping through all of vector a
    for n = 1:N(m)
        b_for(n) = a(n)^2;
    end
    t_for(m) = toc;

    % Calculate time for vectorization
    tic
    % Calculate all entries in b by element-wise exponentiation
    b_vec = a.^2;
```

```
    t_vec(m) = toc;

    % Clear b vectors for next iteration
    clear b_for b_vec
end
```

Plotting the above simulation results. Note the addtions to the code to produce a more professional plot. We will discuss plotting in detail later in the semester.

```
% Define dimensions of plot
figure('Position',[0,0,500,300]);
% Plot call and graphical commands
plot(N,t_for,'ko-',N,t_vec,'ro-','LineWidth',1.5,'MarkerFaceColor','auto')
set(gca, 'XScale', 'log')
set(gca, 'YScale', 'log')
set(gca,'XMinorTick','on')
xlim([10^0,10^8]);
ylim([10^-6,10^2]);
xlabel('N - number of vector elements');
ylabel('Compuatational time (s)');
legend('for loop','Vectorization','Location','northwest')
```



Major takeaway: you can save an **order of magnitude** of computational time using **vectorization** rather than a for loop (even with preallocation!).

Remember: we want code that is correct, clear, and computationally efficient.

TOPIC 5: Conditionals and Boolean Operators

ENGR105, 9/17/20

Loops (continued): for and while loops

As a reminder, `for` loops enable us to repeat tasks <u>for</u> a fixed number of iterations. In the example last time, we used a `for` loop to create a vector containing the first 10 elements of the Fibonacci sequence.

```
% Populate an array containing the first ten numbers of the sequence.
n = 10;
% Empty the vector x - we used it previously
x = [];
x(1) = 0;
x(2) = 1;
% Create each subsequent entry by taking the sum of the previous two
% entries
for k = 3:n
     x(k) = x(k-1) + x(k-2);
end
% Display the vector
disp(x)
```

```
     0     1     1     2     3     5     8    13    21    34
```

On the other hand, `while` loops are a type of iteration in which a sequence of instructions may be repeated <u>while</u> some condition is true. Last time, we used a `while` loop to create a vector containing all elements of the Fibonacci sequence less than 100.

```
% Max value of entries in Fibonacci sequence
n = 100;
% Define first two entries
x(1) = 0;
x(2) = 1;
% Iterate as long as the last element is less than 100.
while x(end) < n
    % If the last element is less that 100, add the next element of the
    % Fibonacci sequence to the vector
     x(end+1)=x(end)+x(end-1);
end
x(end)=[];  % Remove last element
disp(x)
```

```
     0     1     1     2     3     5     8    13    21    34    55    89
```

Vector preallocation

Preallocation is the process of creating a data structure (e.g. a vector) of final size (usually padded with zeros) before modifying the contents. This reduces the computational time by eliminating the operations associated with resizing the array.

Preallocation is a best practice to use whenever possible -- it is important if you use a `for` loop to populate a vector.

Consider the Fibonacci sequence example using the `for` loop, in which we populated a vector with the first 10 entries of the Fibonacci sequence.

```
% Populate an array containing the first ten numbers of the sequence.
n = 10;
% Empty the vector x - we used it previously
x = [];
x(1) = 0;
x(2) = 1;
% Create each subsequent entry by taking the sum of the previous two
% entries
for k = 3:n
    x(k) = x(k-1) + x(k-2);
end
```

Let's calculate the time required to produce this Fibonacci vector of *differing lengths, with* and *without* preallocation.

To perform this analysis, we are going to:

1. For sequence length, we will determine the time required to calculate the Fibonacci sequence, with and without preallocation
2. Repeat with a greater number of Fibonacci elements.
3. Plot *sequence (vector) length* vs. *computation time*.

```
% Define vector containing the number of entries in each successive
% Fibonacci vector.
N = round(logspace(0,8,20))
```

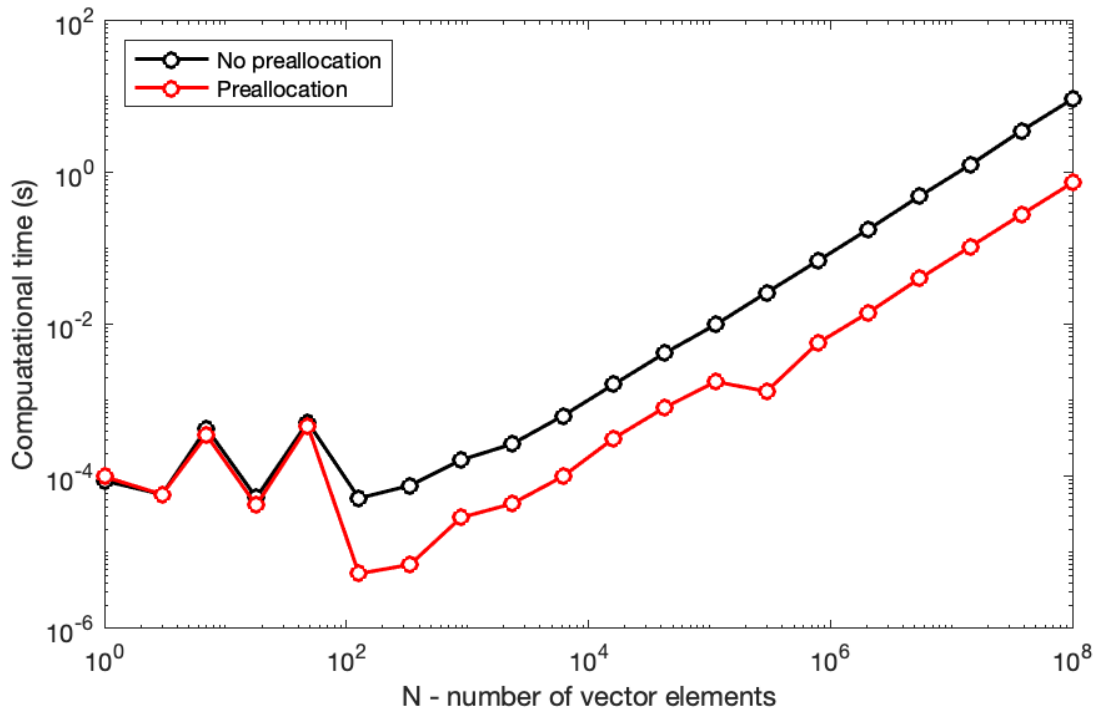
```
N = 1×20
           1           3           7          18          48         127
 336         886        2336        6158       16238       42813
 112884      297635      784760     2069138     5455595    14384499
 37926902   100000000
```

```
% Preallocate vectors to store time information
t_no = zeros(1,length(N));
t_yes = zeros(1,length(N));
% Calculate the time for each vector of length N(m)
for m = 1:length(N)

    % Calculate Fibonacci sequence vector without preallocation
    x = [];
```

```
    tic
    % Define first two entries
    x(1) = 0;
    x(2) = 1;
    % Create each subsequent entry by taking the sum of the previous two
    % entries
    for k = 3:N(m)
     x(k) = x(k-1) + x(k-2);
    end
    t_no(m) = toc;

    % Calculate Fibonacci sequence vector with preallocation
    tic;
    x = zeros(1,N(m));
    % Define first two entries
    x(1) = 0;
    x(2) = 1;
    % Create each subsequent entry by taking the sum of the previous two
    % entries
    for k = 3:N(m)
     x(k) = x(k-1) + x(k-2);
    end
    t_yes(m) = toc;
end
```

Plotting the above simulation results. Note the addtions to the code to produce a more professional plot. We will discuss plotting in detail later in the semester.

```
% Define dimensions of plot
figure('Position',[0,0,500,300]);
% Plot the functions
plot(N,t_no,'ko-',N,t_yes,'ro-','LineWidth',1.5,'MarkerFaceColor','auto')
% Graphical parameters
set(gca, 'XScale', 'log')   % "log" axes - x
set(gca, 'YScale', 'log')   % "log" axes - y
set(gca,'XMinorTick','on')
xlim([10^0,10^8]);  % range of x values on plot
ylim([10^-6,10^2]); % range of y values on plot
xlabel('N - number of vector elements');
ylabel('Compuatational time (s)');
legend('No preallocation','Preallocation','Location','northwest')
```



Major takeaway: you can save an **order of magnitude** of computational time using **preallocation**.

For loops vs. vectorization

As described above, loops can be very useful for repeating many tasks a known or unknown number of times. In the case of for loops, we performed examples that could also be accomplished with vectorization, *i.e.* operating on a vector. Which of these approaches is better? To evaluate this, let's compare the computational time for each.

Specifically, let's calculate `b(k) = a(k)*a(k)`, where `a` is some vector. That is, we want to create a vector `b` in which each entry in the vector is the square of the corresponding entry in the vector `a`.

1) Using a `for` loop.

```
for k = 1:length(a)
    b(k) = a(k)*a(k);
end
```

2) Vectorization (e.g. using an element-wise operation)

```
b = a.^2;
b = a.*a;
```

To perform this analysis, we are going to:

1. Create a vector of a known length.
2. Calculate the time required to perform the above operation on that vector using either a for loop or with vectorization.
3. Repeat with a longer vector.

4. Plot *vector length* vs. *computation time*.

```
% Define vector containing number of entries in potential vector.
N = round(logspace(0,8,20))
```

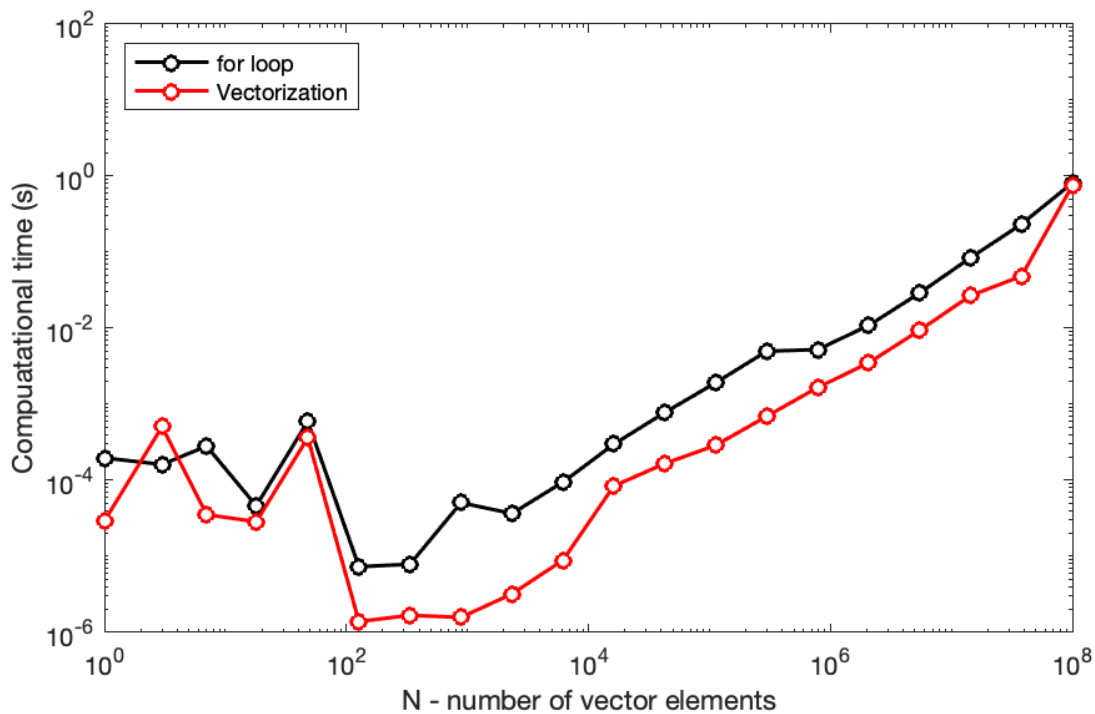
```
N = 1×20
            1           3           7          18          48         127
 336         886        2336        6158       16238       42813
 112884      297635      784760     2069138     5455595    14384499
 37926902   100000000
```

```
% Preallocate vectors to store information
t_for = zeros(1,length(N));
t_vec = zeros(1,length(N));
% Calculate the time for each vector of length N(m)
for m = 1:length(N)
    % Create vector of random numbers with dimension 1xN(m)
    a = rand(1,N(m));

    % Calculate time for for loop
    tic
    % Preallocate for loop vector
    b_for = zeros(1,N(m));
    % Calculate each entry in b by looping through all of vector a
    for n = 1:N(m)
        b_for(n) = a(n)^2;
    end
    t_for(m) = toc;

    % Calculate time for vectorization
    tic
    % Calculate all entries in b by element-wise exponentiation
    b_vec = a.^2;
    t_vec(m) = toc;

    % Clear b vectors for next iteration
    clear b_for b_vec
end
```

Plotting the above simulation results. Note the addtions to the code to produce a more professional plot. We will discuss plotting in detail later in the semester.

```
% Define dimensions of plot
figure('Position',[0,0,500,300]);
% Plot call and graphical commands
plot(N,t_for,'ko-',N,t_vec,'ro-','LineWidth',1.5,'MarkerFaceColor','auto')
```

```
set(gca, 'XScale', 'log')
set(gca, 'YScale', 'log')
set(gca,'XMinorTick','on')
xlim([10^0,10^8]);
ylim([10^-6,10^2]);
xlabel('N - number of vector elements');
ylabel('Compuatational time (s)');
legend('for loop','Vectorization','Location','northwest')
```



Major takeaway: you can save an **order of magnitude** of computational time using **vectorization** rather than a for loop (even with preallocation!).

Remember: we want code that is correct, clear, and computationally efficient.

Recall: while loops iterate as long as a logical condition is true

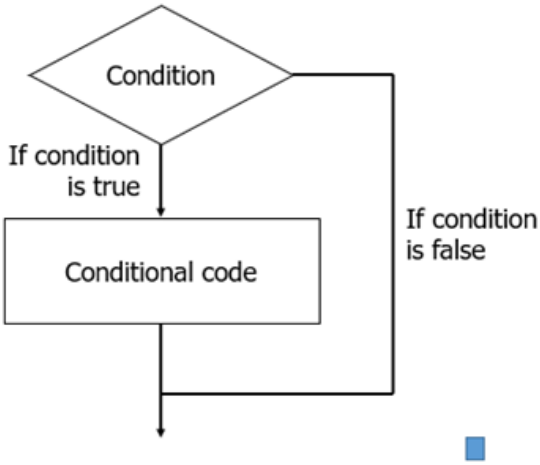
Loops can be used to repeat statements of code either for a defined number of iterations (as in a **`for`** loop) or as long as a logical condition is met (as in a **`while`** loop). In the case of while loops, below, the logical condition can be any relational or logical operation that equates to `true`. When the logical condition is no longer true, the loop ends.

```
while <logical condition>
    <statements>
end
```

Another potentially useful programming tool that makes use of a `<logical condition>` is the **conditional statement**, also known as an `if/then` statement or a **decision**.

Conditional statements

Much of programming is focused on conditionals – the concept of *if this*…then *do that*. A visual representation of a conditional statement is shown below, in which conditional code is only carried out if a logical condition is true.



Anatomy of a simple `if` statement

In MATLAB, conditional statements have the following form.

```
if <logical condition>
    <statements>
end
```

- The `if` command indicates that the `<statements>` that follow should only be executed if the logical condition is **`true`**.
- The `<logical condition>` is any statement that evaluates to a logical result.
- Statements included after the `if` command will execute if the `<logical condition>` is `true`.
- The `end` statement indicates the termination point of any statements that should be executed if the `<logical condition>` is true

Example. Establish a variable with a value and overwrite the value if some statement is true.

```
a = 20;
% The output of b is first established as NaN (nothing special about this
% choice)
b = NaN; % "not a number"

if a >10
    %b would be displayed if the condition is met
    b = 25;
end
```

```
disp(b)
```

```
25
```

Example. Evaluate the sum of a vector to determine if a variable should change value

```
% Declare vector and variable to store a value
v = [1, 5, 3];
res = 5;

% Overwrite a variable if the condition is true
if sum(v)>10
    res = 25;
end

disp(res)
```

```
5
```

Note: Conditional statements must be one-sided – MATLAB will only evaluate the first relational operator of a series or bounded set of logical expressions

```
var1 = 40;
var2 = 12;

if 10 <var1<30
    var2 = 2*var2;
end

disp(var2)
```

```
24
```

Nested `if` statements

`if` statements, like `for`and `while` loops, can be nested. An `end` statement is needed for each `if` statement

```
p = 6;
q = 15;
res = NaN;

% If condition one is true, "enter" the first if statement

if p>5
    if q ==15
        res = p*q
    end
```

```
end
```

```
res = 90
```

```
disp(res)
```

```
90
```

Note: each instance of an `if` statement and the statements within are indented, just like with loops and function.

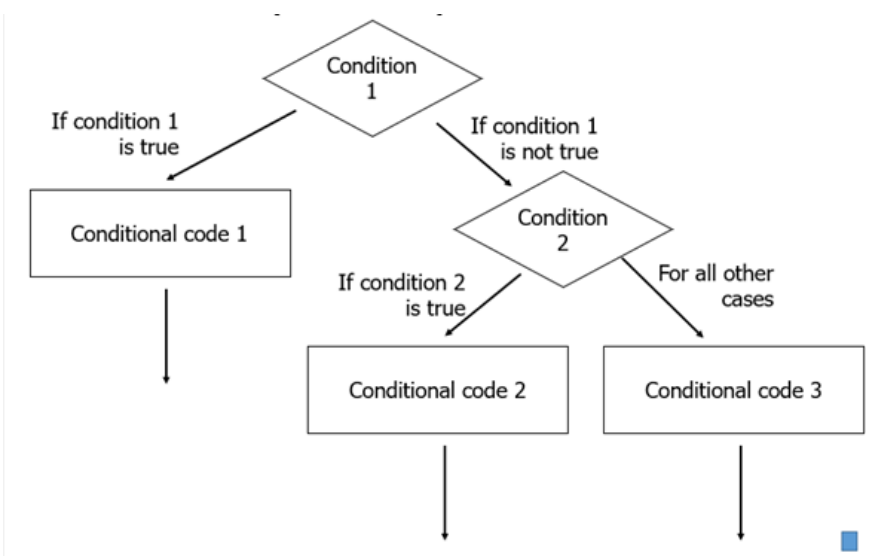
Anatomy of an `if-elseif-else` statement

More complex sets of conditions can be evaluated using the `if-elseif-else` construction

```
if <logical condition>
    <statements>
elseif <logical condition #2>
    <statements>
else
    <statements>
end
```

- Both the `if` and `elseif` statements require a separate `<logical condition>`.
- The `else` statement is a catch all and should not include a `<logical condition>`. If the `<logical condition>` associated with the `if` statement and all `elseif` statements equate to `false`, then the code following the `else` statement will execute.

A visual representation for an `if-elseif-else` statement can be seen below. This visual representation is known as a **decision tree**.



Note: there can be many `elseif` statements in an if statement, though they are not required.

Example. `if-elseif-else` statement for one of my typical mornings.

```
if 'I am out of coffee'
    'Go back to sleep, it will be a terrible day'
elseif 'I am out of nutella'
    'Be grumpy'
elseif 'My garbage is full'
    'Question why I did not take it out yesterday'
else
    'Drink coffee, eat nutella, clean up, nbd'
end
```

Example. "Catching" one condition with `if` and all other conditions with `else`.

```
var1 = 1;
var2 = 2;

% "If var1 equals var2, then assign "true" to out. Otherwise, assign
% "false" to out.

if var1 == var2
    out = true;
else
    out = false;
end
disp(out)
```

```
    0
```

Note: this code is demonstrational. The same output can be achieved using the single expression:

```
out = var1==var2;
disp(out)
```

```
    0
```

Example. Assign value to variable based on input variable.

```
% Classic problem: if my cereal is bad, the solution is clearly to add
% chocolate to it.
cerealGrade = 95;

if cerealGrade < 60
    chocolateAdded = 20;
elseif cerealGrade < 70
    chocolateAdded = 15;
elseif cerealGrade < 80
    chocolateAdded = 10;
elseif cerealGrade  <90
    chocolateAdded = 5;
else
```

```
    chocolateAdded = 0;
end
disp([cerealGrade,chocolateAdded])
```

```
     95     0
```

Note: The decision statement will terminate after the first condition is met. If computing with large data sets, catch the most common decision with the first `if` statement when possible.

The above code can be made more compact (*and easier to read*) so that it does not require the use of the `else` statement.

```
cerealGrade = 95;
chocolateAdded = 0;
%reduces the amoung of code becasue the end statement does not need to be
%included

if cerealGrade < 60
    chocolateAdded = 20;
elseif cerealGrade < 70
    chocolateAdded = 15;
elseif cerealGrade < 80
    chocolateAdded = 10;
elseif cerealGrade  <90
    chocolateAdded = 5;
end
disp([cerealGrade,chocolateAdded])
```

```
     95     0
```

In the above, `chocolateAdded` is assigned a value of `0` before the decision and is only overwritten if one of the `if` or `elseif` statements reads `true`.

Boolean operators

Boolean (logical) operators can be used in conjunction with `while` loops and conditionals (`if` and `elseif`) to yield more complex terminating behaviors.

Logical operator (scalars)	Logical operator (element-wise on vectors)	Function call	Meaning
&&	&	and(a,b)	And
||	|	or(a,b)	Or, inclusive
		xor(a,b)	Or, exclusive
~	~	not(a)	Not / compliment

The above operators can be stated as:

- **And**: if both a and b are `true`, the a&b is `true`.
- **Or**: if at least one of a and b is `true`, then a|b is `true`.
- **Xor**: if only one of a and b is `true`, then `xor(a,b)` is `true`.
- **Not**. If a is `true`, then ~a is `false`. If a is `false`, then ~a is `true`.

The behavior of logical operators can be compactly described in **truth tables**.

Truth table for `A & B`

A	B	A & B
T = true	T	T
T	F	F
F	T	F
F = false	F	F

Truth table for `A | B`

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

Truth table for `xor(A,B)`

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

Examples: scalars

When the scalar operations are used, the logical operator simply compares the two scalar values and returns `true` (1) or `false` (0).

```
% Examples using And
res1 = (1<2) && (4==5)
```

```
res1 = logical
   0
```

```
Alpha = (1<2) && (4<5)
```

```
Alpha = logical
   1
```

```
% Examples using Or
var_2 = true || false
```

```
var_2 = logical
   1
```

```
ind = (1<2) || (4==5)
```

```
ind = logical
   1
```

```
% Examples using xOr
ip = xor(1<2,4<5)
```

```
ip = logical
   0
```

Examples: element-wise on vectors

When the element-wise operations are used, the **output is a logical vector** where each entry is a result of the comparison between the values in corresponding indices from the two input vectors.

```
a = [1,0,0,1]
```

```
a = 1×4
     1     0     0     1
```

```
b = [1,1,0,0]
```

```
b = 1×4
     1     1     0     0
```

```
a & b
```

```
ans = 1×4 logical array
   1   0   0   0
```

Note: when performing element-wise logical operations, be sure to use the correct (single) operators.

Precedence revisited

Boolean operators may be combined to yield interesting decisions. Combinations of operators are executed left to right with **`and()` commands having higher precedence than `or()`** commands. An updated precedence table can be seen here.

Precedence	Operator	MATLAB Representation
1	Parentheses	(A)
2	Power, transpose	A^x, A.^x, A', A.'
3	Multiplication and division	A*B, A.*B, A/B, A./B, A\B, A.\B
4	Addition and subtraction	A+2; A-2
5	Colon operator	1:5
6	Relational operators	A>B, A<B, A>=B, A<=B, A==B, A~=B
7	Element-wise AND	&
8	Element-wise OR	|
9	Scalar AND	&&
10	Scalar OR	||

Example. Assume variables a through `f` represent scalar numbers *(e.g.* `true` or `false`). What will the following code return for each possible combination?

```
switchState = a>b | c>d & e>f
```

The value of `switchState` as a function of all possible combinations of values a through `f` can be represented as a truth table.

a>b	| c>d	& e>f	switchState
T	T	T	?
T	T	F	?
T	F	T	?
F	T	T	?
T	F	F	?
F	T	F	?
F	F	T	?
F	F	F	?

The **`and()`** operation is performed **first**, based on higher precedence.

a>b	| c>d & e>f	switchState
T	T	?
T	F	?
T	F	?
F	T	?
T	F	?
F	F	?
F	F	?
F	F	?

Taking **`or()`** of column one and then "`and` column" two yields the `switchState` given any combination of inputs `a` through `f`.

a>b | c>d & e>f	switchState
T	T
T	T
T	T
T	T
T	T
F	F
F	F
F	F

Or, to summarize:

a>b	| c>d	& e>f	switchState
T	T	T	T
T	T	F	T
T	F	T	T
F	T	T	T
T	F	F	T
F	T	F	F
F	F	T	F
F	F	F	F

Example.

```
A = true;
B = false;
C = false;
```

Due to precedence, the above code is equivalent to the following, which has unnecessary parentheses ...

```
out
```

```
out = logical
   0
```

And different from the following, in which precedence has been overriden.

Boolean operators and conditional statements

Logical operators may be used to remove nested `if` statements. Consider the example above:

```
p = 6;
q = 15;
res = NaN;

if p > 5
    if q == 15
        res = p*q;
    end
end
disp(res)
```

```
90
```

By using a logical operator, we can simplify this code.

```
if p>5 && q==15
    res = p*q
end
```

```
res = 90
```

```
disp(res)
```

```
90
```

Example. Consider the following code.

```
% Declare initial variable
res = false;

a = 15;
b = 20;
c = 30;

% If this condition is true, change the value of res
if a<20 && b==20 || c>20
    res = true;
end
disp(res)
```

```
1
```

Remember: precedence order is important. The above is read:

1. `a < 20` is `true`, `b == 20` is `true`, `c > 30` is `true`.
2. Thus, `(a < 20)` AND `(b == 20)` is `true`
3. Thus, `((a < 20)` AND `(b == 20))` OR `c>20` is `true`.

Logical statements and `while` loops

Logical statements may also be used to continue or stop `while` loop execution.

```
% Assign initial values to variables variables
a = 15;
b = 17;

% As long as AT LEAST ONE of the current values of a or b is greater than
% 5, the loop will continue. That is, the loop will stop when BOTH a and b
% are less than 5.
while a(end)>5 || b(end)>5
    %generates random numbers from 1 to 20 and adds them to the vector.

    a(end+1) = randi([1 20]);
    b(end+1) = randi([1 20]);
end


disp([a',b'])
```

```
    15    17
    20    14
    20    15
     6     5
    17     9
     9     8
     6     7
    10     4
    10    18
     2    16
    11     2
    14     8
    16    12
     7    19
     7     7
    16     7
     7     6
     2    10
    15     9
     1    19
    14    14
    14     1
     4    17
    18     2
     3    14
     8    15
    16     7
     2     9
    13    11
```

```
    11     7
     3    17
    11     7
    14     5
    16     7
    11    18
    12     4
     5    16
    18     2
    20     9
    15    17
     9    10
     9    12
    18     5
    19    11
    13    16
     9    15
     5    16
     8    14
     9    16
    14     3
    13    20
     9    14
    14     2
     9     7
    19    19
    19     5
    19     4
    12     1
    11    14
    20     5
     5     3
```

With this, we now have some basic

Next time: we will use loops and conditionals to practice solving some problems.

TOPIC 6: Building solutions to problems, part 1

ENGR105, 9/22/20

Recap of topics covered so far:

1. MATLAB syntax and basics: using MATLAB to perform calculations
2. Scalars, vectors, and matrices: storing, accessing, and manipulating numerical information
3. Functions and scripts: creating useful collections of code
4. `for` and `while` loops: creating code that efficiently repeats tasks
5. Conditionals: incorporating decision making into your code

1) MATLAB syntax and basics: using MATLAB to perform calculations

We've learned how to use MATLAB as a tool for performing simple mathematical calculations.

```
% Using comments to explain our code, we can perform basics tasks, look up
```

```
% information about functions, etc.
1 + 2
```

```
ans = 3
```

```
% We have learned how to use the resources in MATLAB to perform basic
% calculations
5*tan(pi/4)
```

```
ans = 5.0000
```

```
a = 15; % Assigning a scalar variable
b = 13+2/a;  % Creating a statement based on the previous variable
disp(b) % Display the newly defined variable
```

```
   13.1333
```

2) Scalars, vectors, and matrices: storing, accessing, and manipulating numerical information

MATLAB excels with and is optimized for performing operations on vector, matrices, and higher order arrays. We have learned how to create vector, matrices, etc., access their components through indexing, and perform array mathematics. This is the basis for **vectorization** of solutions.

```
% Create vectors
a = 1:5;
b = linspace(1,4,5);
c = zeros(3,3);
c(1:9) = randi(5,[1,9]);
disp(a)
```

```
     1     2     3     4     5
```

```
disp(b)
```

```
    1.0000    1.7500    2.5000    3.2500    4.0000
```

```
disp(c)
```

```
     2     2     1
     1     5     2
     4     2     5
```

```
% Access elements in the vectors and matrix
a([1:2,end])
```

```
ans = 1×3
     1     2     5
```

```
b(randi(5,[1,10]))
```

```
ans = 1×10
```

```
    4.0000    4.0000    2.5000    3.2500    3.2500    1.7500    1.0000
3.2500    4.0000    1.0000
```

```
c([1,3],[1,3])
```

```
ans = 2×2
     2     1
     4     5
```

```
% Perform array math
a+b     % vector addition
```

```
ans = 1×5
    2.0000    3.7500    5.5000    7.2500    9.0000
```

```
a.*b    % element-wise multiplication
```

```
ans = 1×5
    1.0000    3.5000    7.5000   13.0000   20.0000
```

```
(a')*b  % matrix multiplication
```

```
ans = 5×5
    1.0000    1.7500    2.5000    3.2500    4.0000
    2.0000    3.5000    5.0000    6.5000    8.0000
    3.0000    5.2500    7.5000    9.7500   12.0000
    4.0000    7.0000   10.0000   13.0000   16.0000
    5.0000    8.7500   12.5000   16.2500   20.0000
```

```
a(1:3)+c(1,:)   % Adding subvectors derived from original array
```

```
ans = 1×3
     3     4     4
```

3) Functions and scripts: creating useful collections of code

We have created assemblages of code to run a series of tasks. Scripts are simply collections of multi-line code. Functions perform collections of code, take inputs, and return outputs to the base MATLAB workspace.

An example function declaration:

```
function [out1,out2] = myFunction(input1,input2,input3)
```

This example function takes **three** inputs and returns **two** outputs.

4) `for` and `while` loops: creating code that efficiently repeats tasks

- **`for`** loops are useful when one knows (or can calculate) the number of iterations to perform ahead of time.

```
a = zeros(1,10);    % Preallocate vector to store values
% Perform 10 iterations, each iteration calculate square root of different
% random number
for k = 1:10
    a(k) = sqrt(randi(k));
end
disp(a)
```

```
    1.0000    1.4142    1.7321    1.7321    1.4142    1.4142    2.4495    2.4495
 1.4142    2.4495
```

- **`while`** loops are useful when one does not know how many iterations must be performed.

```
% Create a vector containing all multiples of 13 less than 300;
a = 13; % vector to store integer multiples of 13. Start by populating first
(obvious) case
num = 1;    % counter to determine next multiple
while a(end) < 300
    num = num + 1;
    a(end+1) = num*13;
end
a(end) = [];    % Delete the last entry because it will be over 200;
disp(a)
```

```
    13    26    39    52    65    78    91   104   117   130   143   156   169   182
 195   208   221   234   247   260   273   286   299
```

5) Conditionals: incorporating decision making into your code

Like `while` loops, conditionals use **logical and relational operators** to determine which piece of conditional code will be executed. This provides instructions that enables your code to *decide* how to proceed.

```
% Generate two random numbers
a = randi(5);
b = randi(5);

if a==b % if variables are equal
    disp(a*b*ones(1,a*b))
elseif a<b % if b is greater than a
    disp(b*ones(1,a))
else    % if a is greater than b
    disp(a*ones(1,b))
end
```

Using scientific computing to approach problems

The above syntax and programming capabilities are tools for solving scientific problems. We are ultimately interest in applying these tools (and other syntax, computational capabilities, and programming concepts) to solve problems.

NOTE: developing a solution to a problem often seems **daunting** at first. As you are getting started with scientific computing, approach new problems in a **systematic** way, breaking the overall task into simpler sub-tasks. One systematic approach is discussed here.

PRO: a roadmap for building computational solutions

1) Plan, **P**repare, and **P**seudocode

The first thing to do, before writing any code, is to develop a plan breaking down the steps that your code should accomplish to produce the desired solution.

- Identify the problem being solved and propose the type of computational solution necessary.
- Identify what is known, what should be calculated, and what is to be solved for or output. Write out equations and draw pictures by hand.
- While you may start by writing or sketching this on paper (I do this), convert your handwritten notes and thoughts into **MATLAB comments**.
- Lay out your comments spatially in a script (.m) file or Livescript code. The spatial layout of your comments should indicate the **order of tasks** you want to perform.
- Your comments should be **pseudocode**: they should be written in English and convey what you want your code to accomplish.

***** Your pseudocode is the foundation for your solution *****

2) w**R**ite **R**ough (increments of code) and **R**efine

The second thing to do, once you have produced pseudocode, is to start filling in sections of code *in increments*, with rough code that simply works. That is, build your computational

- Do **NOT** try to "hit a home run" on your first attempt by generating perfect code immediately.
- Systematically fill in pieces of code, ensure that they run properly and as expected.
- Start with rough (potentially inefficient) code that simply produces the correct output.
- Deal with errors and malfunctions as necessary.
- Use your rough code in one section to start the next section.
- Consistently refine your code throughout this process.
- If your ultimate goal is to produce a complex piece of code involving functions, start by writing all functions as scripts to ensure that each function performs as you expect

***** Your rough code, built upon the pseudocode foundation, is the "core structure" and "meat" of your solution *****

3) Optimize and improve

The final thing to do is to convert your rough code into an optimal solution.

- Ensure that all functions are packaged properly and clear.
- Ensure your code is clear and well-commented.
- Ensure your code is operating as computationally efficient as possible. Look for ways to increase performance (e.g. vectorization vs. for loops, preallocation, etc.)

***** Your optimized solution "polishes" the code structure and foundation you have built *****

Troubleshooting and debugging techniques

When you start developing a computational solution to a problem, expect errors and issues to arise. Anticipating potential errors will help you more quickly overcome them and produce working code.

Potential errors and issues include:

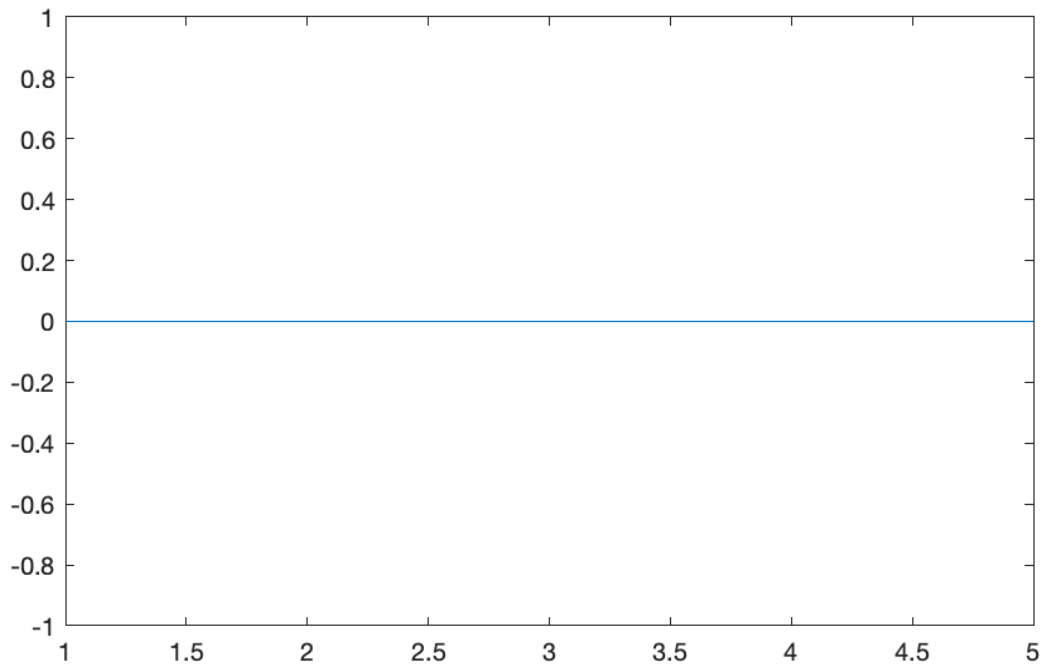
1) Your code does not run

Debugging techniques and tips:

- The issue is most likely either MATLAB syntax or a function being used improperly.
- Use the error information to identify where in your code the issue is happening.
- Unsuppress outputs (remove `;`) to print the results of computations to the command line.
- Insert a **breakpoint** to step through a script or function. Compare the computations to what you expect.
- If you are making a function, start by creating it as a script. Once that runs properly, expand that code into a function.
- Use `disp()` to output a variable to the command line.
- Run sections of code at a time to observe how the expect outputs change through the script.
- Comment large sections of code to isolate where the error is occuring.

Example:

```
a = [5,2,1];
b = zeros(3,1);

% Goal: produce a plot
plot(a,b)
```



2) Your code runs but the output is not in the correct form

Debugging techniques and tips:

- The issue is most likely MATLAB syntax -- you might be using a function incorrectly, incorrectly produced an array of the wrong size, etc.
- Compare the output from your code at various points to determine where values deviate from what is expected.
- Unsuppress outputs (remove `;`) or use `disp()` to print the results of computations to the command line.
- Insert a **breakpoint** to step through a script or function. Compare the computations to what you expect.

Example:

```
% Goal: create  matrix through matrix multiplication
a = 1:5;
b = ones(5,1);
a*b
```

```
ans = 15
```

3) Your code runs and produces the correct form of output, but it is not physical or realistic

Debugging techniques and tips:

- As above, the issue is most likely MATLAB syntax -- you might have mis-entered an equation, misused an operator, etc.

- Compare the output from your code at various points to determine where values deviate from what is expected.
- Unsuppress outputs (remove `;`) or use `disp()` to print the results of computations to the command line.
- Insert a **breakpoint** to step through a script or function. Compare the computations to what you expect.

Example:

```
% % Goal: calculate the terminal velocity of a falling object (positive is
% % down)
g = 9.81; % m/s^2
d = 0.001; % diameter, m
mu = 10^-3; % Pa*s
pm = 1000; % kg/m^3
ps = 1050; % kg/m^3

Vt = g*d^2/(12*mu)*(ps - pm)
```

```
Vt = 0.0409
```

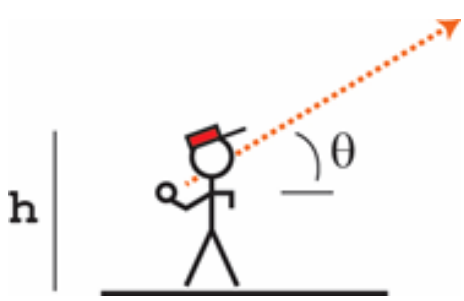
Getting started with building computational solutions to problems.

We will practice applying what we have learned to four different example problems representing a few different applications of scientific computing.

1. Modeling the motion of a projectile
2. Creating a numerical method to find the root of an equation
3. Programming a gameplay algorithm

Practice problem 1: modeling the motion of a projectile

In your introductory physics courses, you studied kinematics and the motion of projectiles. Such analyses may apply in many circumstances, including a system in which a ball is thrown at some angle relative to the ground, as seen below.



Consider a **2 m** tall athlete who can easily throw the ball **30 m/s**. Assume the ball is released at the top of the player's head. **Determine and plot the maximum height of the ball and the distance thrown (assuming the ball doesn't bounce) as a function of the angle by which the ball is thrown.**

<u>PRO, Step 1</u>: <u>P</u>lan, **<u>P</u>**repare, and **<u>P</u>**seudocode

Helpful tasks:

- When faced with a problem, it is helpful to first take a step back, take a deep breath, and write down what we know. In this case, we know from a force balance on the ball immediately after it is thrown that the ball's acceleration, velocity, and position can be represented by the following equations (neglecting any air resistance):

$$a_x(t) = \frac{d^2x}{dt^2} = 0$$

$$a_y(t) = \frac{d^2y}{dt^2} = -g$$

$$v_x(t) = \frac{dx}{dt} = v_0 \cos(\theta)$$

$$v_y(t) = \frac{dy}{dt} = v_0 \sin(\theta) - g\,t$$

$$x(t) = v_0 \cos(\theta) t$$

$$y(t) = h + v_0 \sin(\theta) t - \frac{g}{2} t^2$$

- We know the equations describing our system. **How can we calculate the max height and distance thrown?**

Distance thrown. To calculate the distance the ball travels in the air. we first find the time at which the ball lands by setting `y(t)` to zero and solving for `t`.

$$0 = h + v_0 \sin(\theta) t_{\text{end}} - \frac{g}{2} t_{\text{end}}^2$$

$$t_{\text{end}} = \frac{-v_0 \sin(\theta) - \sqrt{v_0^2 \sin^2(\theta) + 2g\,h}}{-g}$$

The above is obtained simply by using the quadratic formula. Substituting `t_end` into `x(t)` produces the distance the ball travels.

$$x_{\text{end}} = v_0 \cos(\theta) t_{\text{end}}$$

Maximum height. To calculate the maximum height of the ball, calculate when the `y` component of the velocity (or the derivative of the y position) is zero, as this will be the peak of the trajectory.

$$v_y(t) = 0 = v_0 \sin(\theta) - g\,t_{\text{peak}}$$

$$t_{\text{peak}} = \frac{v_0 \sin(\theta)}{g}$$

Substituting into the equation for y(t), we determine the maximum height of the ball.

$$y_{\max} = h + v_0 \sin(\theta) t_{\text{peak}} - \frac{g}{2} t_{\text{peak}}^2$$

<u>Note</u>: *(as seen above)* before producing a computational solution, it is useful to set up the problem by writing out (by hand) all of the known variables and equations describing the system.

- What is the objective of our code? In this case, we want to calculate two values as a function of an input angle and then repeat that calculation for multiple input angles. Thus, you may envision eventually using **some iteration**.
- We now know all the equations that we need to solve this problem, but how is that converted into code? Write out the **pseudocode** for what the script may look like.

```
%step 1: specify angle


%step 2: constants

% Repeat for all angles
%step 3: calculate distance for each theta

%step 4: calculate maximum height
```

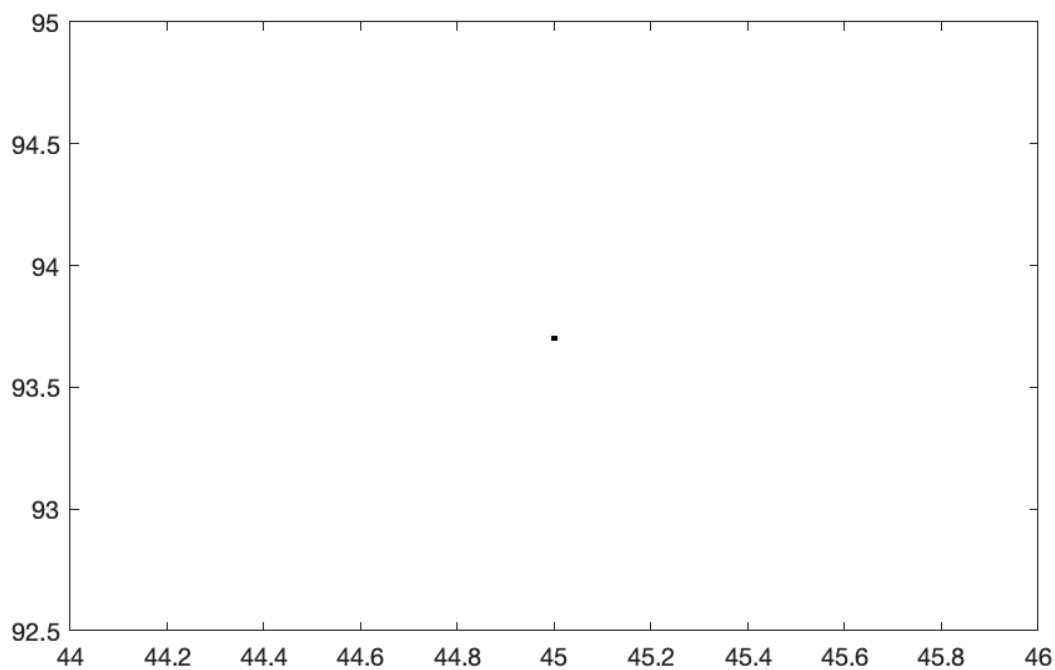
<u>PRO, Step 2</u>: w**<u>R</u>**ite **<u>R</u>**ough (increments of code) and **<u>R</u>**efine

Helpful tasks:

- Starting with your pseudocode, roughly fill in code.
- While we eventually may want to use a loop, write out all the code as a script with one "test input."

```
% %step 1: specify angle
% th = 45; %in degrees
% th_r = th*pi/180;%radians
%
% %step 2: constants
% g = 9.81; %m/s^2
% h = 2; %height
% v0 = 30; %initial velocity
%
% %step 3: calculate distance for theta
% t_end = (-v0*sin(th_r) - sqrt(v0^2*sin(th_r)^2 + 2*g*h))/(-g);
% x_end = v0 *cos(th_r)*t_end
%
% %step 4: caculate max height
% t_peak = v0*sin(th_r)/g;
% y_peak = h + v0*sin(th_r)*t_peak - g/2*t_peak
```

```
%step 1: specify angle
th = 45; %in degrees
th_r = th*pi/180;%radians

%preallocationg
x_end = zeros(1,length(th));
y_peak = zeros(1,length(th));

%step 2: constants
g = 9.81; %m/s^2
h = 2; %height
v0 = 30; %initial velocity

for k = 1:length(th)
    %step 3: calculate distance for theta
    t_end = (-v0*sin(th_r(k)) - sqrt(v0^2*sin(th_r(k))^2 + 2*g*h))/(-g);
    x_end(k) = v0 *cos(th_r(k))*t_end;

    %step 4: caculate max height
    t_peak = v0*sin(th_r(k))/g;
    y_peak(k) = h + v0*sin(th_r(k))*t_peak - g/2*t_peak^2;
end
plot(th,x_end,'k.')
```



PRO, Step 3: Optimize and improve

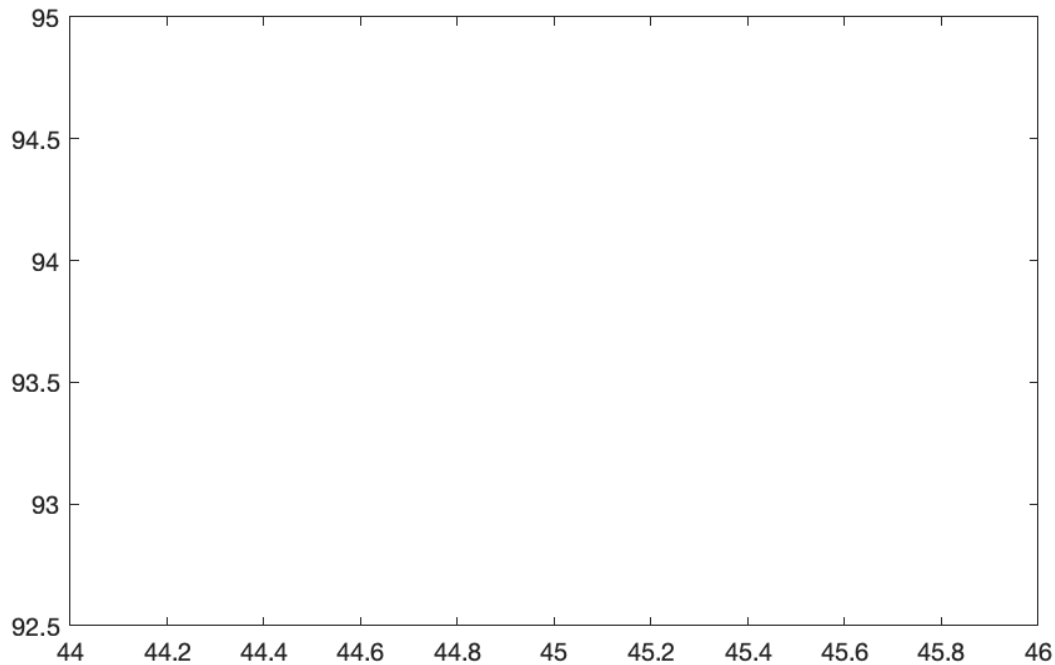
There are a number of issues to optimize here:

- Constants are being defined inside the for loop
- For loops are generally more inefficent that vectorization
-

```
% %step 1: specify angle
th = 45; %in degrees
th_r = th*pi/180;%radians

% %step 2: constants
g = 9.81; %m/s^2
h = 2; %height
v0 = 30; %initial velocity
%
% %step 3: calculate distance for theta
t_end = (-v0*sin(th_r) - sqrt(v0^2*sin(th_r)^2 + 2*g*h))/(-g);
x_end = v0 *cos(th_r)*t_end
```

```
x_end = 93.7013
```

```
% %step 4: caculate max height
t_peak = v0*sin(th_r)/g;
y_peak = h + v0*sin(th_r)*t_peak - g/2*t_peak;
plot(th,x_end,'k-')
```



Practice problem 2: numerical methods and root-finding

Numerical analysis is the domain of mathematics and computer science that creates, analyzes, and implements algorithms for solving numerically the problems of continuous mathematics. Numerically solving a complex math problem often involves iterating or following a given algorithm until it converges (hopefully) upon a solution, within some defined tolerance.

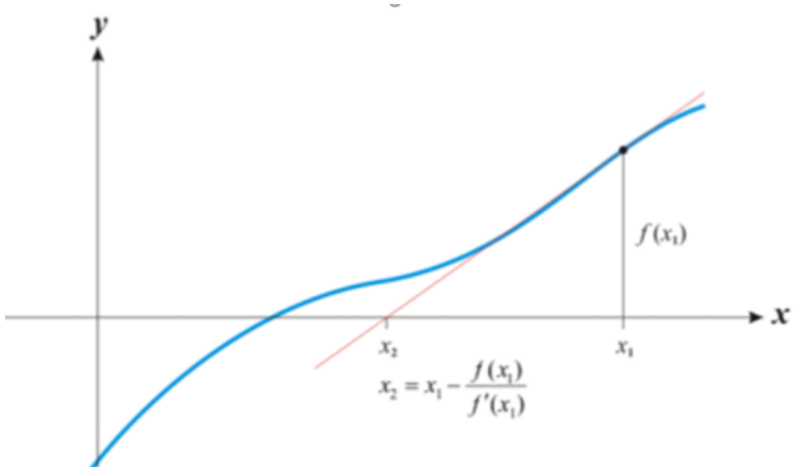
One such application of numerical methods is root-finding, the algorithms for which may quickly find the root ($F(x) = 0$) for some arbitrary function $F(x)$. Say we have a (potentially very nonlinear) function for which we want to find $F(x) = 0$ quickly. The Newton-Raphson method searches for the root of a function by iteratively following where the tangent of a function crosses the x axis.

Specifically, **the Newton-Raphson method has the following algorithm:**

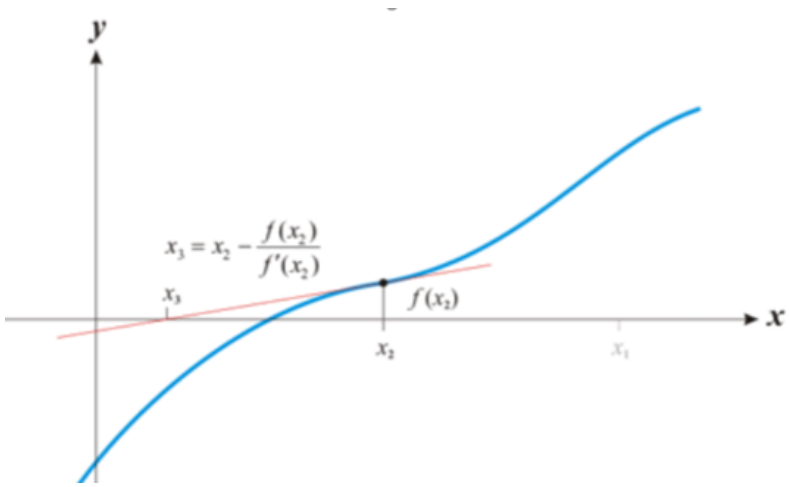
1. Pick an initial guess (x_1) of the root of a function.
2. At that x value, calculate the tangent to the curve of the function.
3. Find where the tangent to the curve intersects the x axis.This x value becomes x_2.
4. At this new x value, repeat steps 2-3.
5. When the new x value is not much different than the previous x value (as determined by some pre-defined tolerance), conclude that you have found an approximation for the root of the function. That is, you found a numerical approximation for the x value for which $F(x) = 0$.

Visual depiction of the Newton-Raphson algorithm

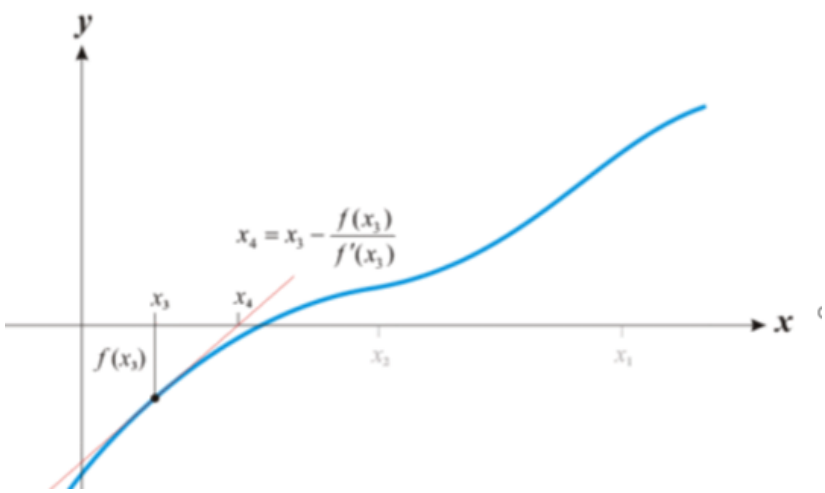
Iteration 1: Make an initial guess, calculate where the tangent line at that point intersects x axis.



Iteration 2: Using this new x value, repeat the process



Iteration 3: The algorithm is beginning to converge on the root of $F(x)$.



In these images, x_n is the current guess for the root and x_{n+1} is the next input in this iterative process. When x_n -> x_{n+1}, we have found the root of the function.

Derivation of the Newton-Raphson method

1) The tangent line of the function at some point *x* can be written generally as:

$$y = m\,x + b$$

2) The slope, *m*, is simply the derivative of the function at xn:

$$y = f'(x_n)\,x + b$$

3) At *x* = xn, y = *f(*xn*)*, which allows us to solve for *b*

$$f(x_n) = f'(x_n)x_n + b \longrightarrow b = f(x_n) - f'(x_n)x_n$$

4) Substituting, we can write an equation for the tangent line in terms of $f(x_n), f'(x_n), x_n$:

$$y = f'(x_n)x + (f(x_n) - f'(x_n)x_n)$$

5) In this algorithm, we solve for *y* = 0 for the tangent line. Thus, at y=0, and *x* = xn+1:

$$0 = f'(x_n)x_{n+1} + (f(x_n) - f'(x_n)x_n)$$

6) Rearranging the previous for xn+1 produces the Newton-Raphson equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Thus, we have derived a numerical method for iterating until we converge on the root of a function.

Question 1

Use the Newton-raphson method to find the roots to the following equation:

$$f(x) = x^4 + 2(x-2)^2 - 20$$

<u>PRO, Step 1</u>: <u>P</u>lan, **<u>P</u>**repare, and **<u>P</u>**seudocode

Helpful tasks:

- We know the equation we want to solve for, can we plot it to get an idea of where the roots might be?
- We know the equation and the algorithm, can we write explicitly write out what the algorithm will look. That is, here:

$$f(x) = x^4 + 2(x-2)^2 - 20$$
$$f'(x) = 4x^3 + 4(x-2)$$

- We know the algorithm, but how is that converted into code? Write out pseudocode for what the script will look like.
-

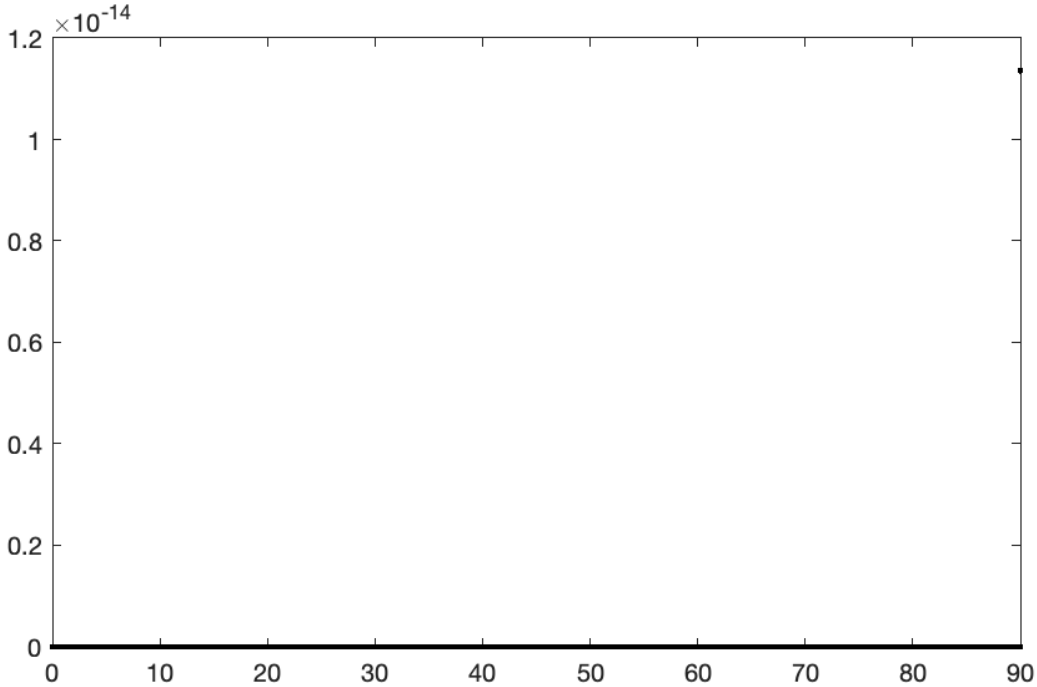
```
% Step 1: specify 1 angle%
% Step 2: constants% Repeat for all angles%
% Step 3: calculate distance for each theta%
% Step 4: calculate maximum height
```

PRO, Step 2: w**R**ite **R**ough (increments of code) and **R**efine

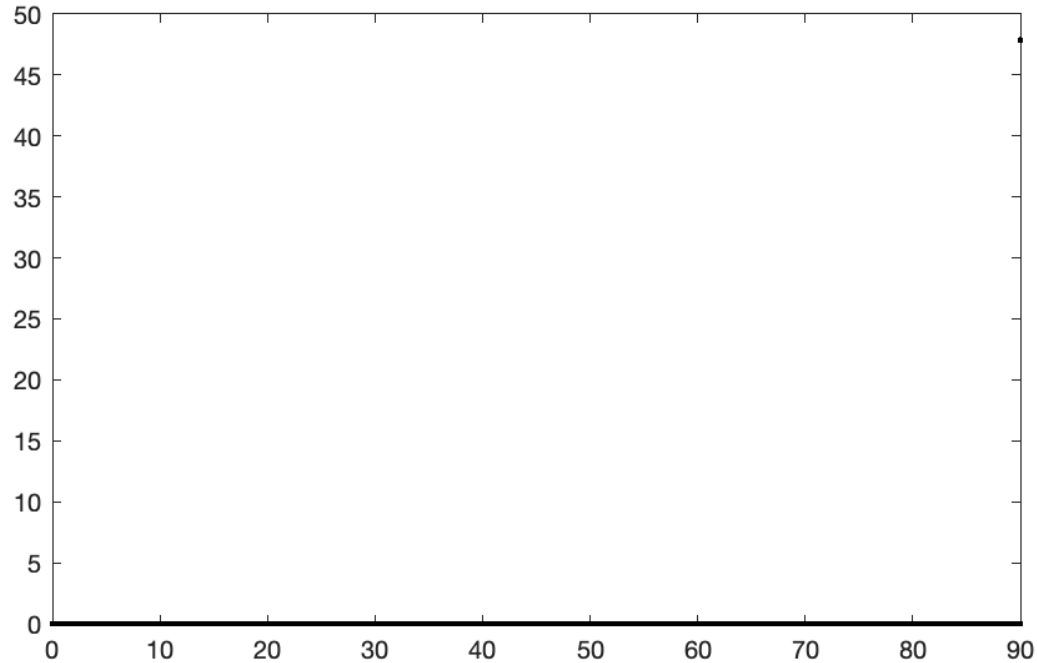
Helpful tasks:

- Starting with your pseudocode and what you know about the function, roughly fill in code.

```
%Step 1: specify angle
th = 0:0.005:90; %degrees
th_r = th*pi/180; %radians

%Preallocate vector
x_end = zeros(1,length(th));
y_peak = zeros(1,length(th));

%Step 2: constants
g = 9.81; %m/s^2
h = 2; %m
v0 = 30; %m/s

%repeat for each theta
for k = length(th)
    %repeat for all angles
    %step 3: calculate distance for each theta

    t_end = (-v0*sin(th_r(k)) - sqrt(v0^2*sin(th_r(k))^2 + 2*g*h))/(-g);
    x_end(k) = v0*cos(th_r(k))*t_end;

    %Step 4L calculate maximum height
    t_peak = v0*sin(th_r(k))/g;
    y_peak(k) = h + v0*sin(th_r(k))*t_peak - g/2*t_peak^2;
end

%plot the distance
plot(th,x_end,'k.')
```



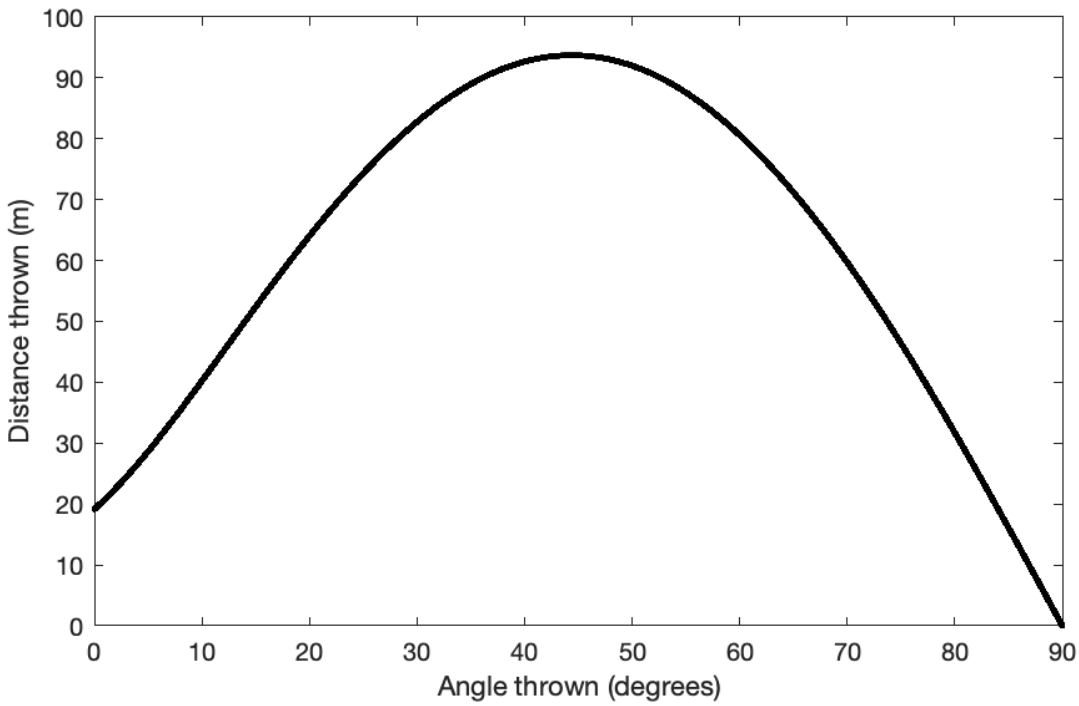

```
% Plot the height of the ball
plot(th,y_peak,'k.')
```



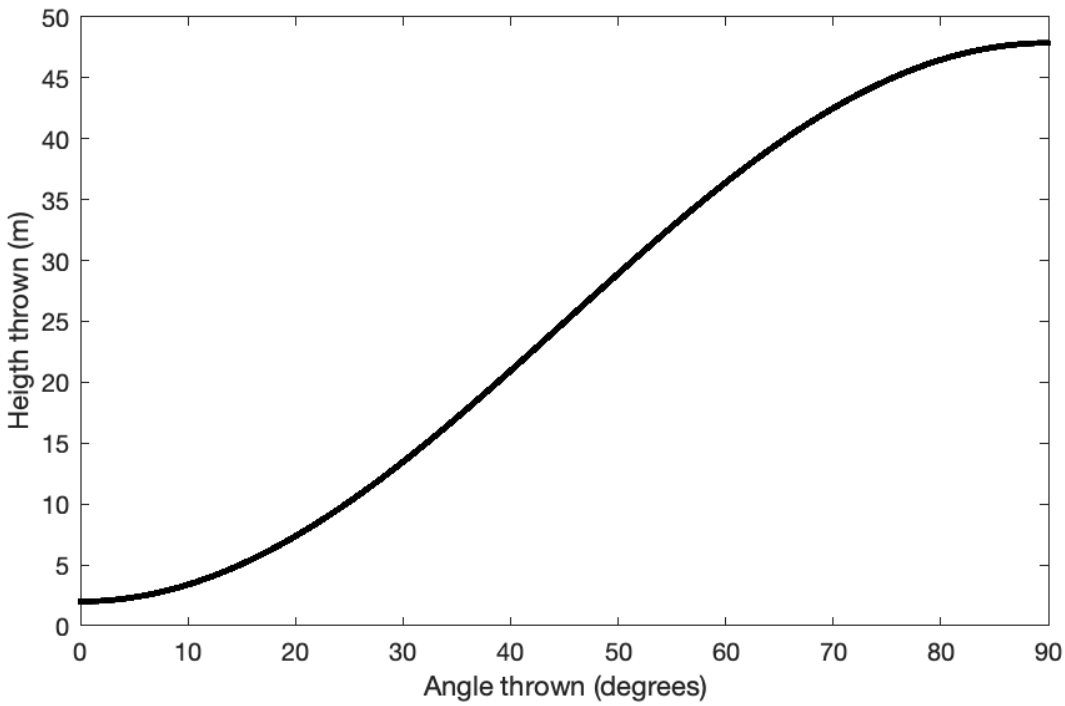
PRO, Step 3: **O**ptimize and improve

We don't actually know how many loop iterations to perform ahead of time. Refine the code so that it is in the form of a `while` loop.

```
%Step 1: specify angle
th = 0:0.005:90; %degrees
th_r = th*pi/180; %radians

%Preallocate vector
x_end = zeros(1,length(th));
y_peak = zeros(1,length(th));

%Step 2: constants
g = 9.81; %m/s^2
h = 2; %m
v0 = 30; %m/s

% Step 3: Calculate distance thrown - element-wise
t_end = (-v0*sin(th_r) - sqrt(v0^2*sin(th_r).^2 + 2*g*h))/(-g);
x_end = v0*cos(th_r).*t_end;

% Step 4: calculate maximun height
t_peak = v0*sin(th_r)/g;
y_peak = h + v0*sin(th_r).*t_peak - g/2*t_peak.^2;

%plot the distance
plot(th,x_end,'k.')
xlabel('Angle thrown (degrees)')
ylabel('Distance thrown (m)')
```




```
% Plot the height of the ball
plot(th,y_peak,'k.')
xlabel('Angle thrown (degrees)')
ylabel('Heigth thrown (m)')
```



Note: this example has two roots. The root to which the function converges is dependent on the initial guess of the root location

Question 2

Use the Newton-raphson method to create a function for numerically approximating the square root of any number.

<u>PRO, Step 1</u>: <u>P</u>lan, **<u>P</u>**repare, and **<u>P</u>**seudocode

Helpful tasks:

- We know the equation we want to solve for, let's convert it into a form that is easily used with Newton-raphson

  $$x = \sqrt{a}$$
  $$x^2 = a$$
  $$0 = x^2 - a$$

  In other words, we want to find the root for the following equation:

  $$f(x) = x^2 - a$$

- We know the equation and the algorithm, can we write explicitly write out what the algorithm will look. That is, here:

$$f(x) = x^2 - a$$
$$f'(x) = 2x$$

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

- We know the algorithm, but how is that converted into code? Write out pseudocode for what the script will look like.

```
% Step 1: specify initial guess

% Step 2 L Iterate until xn -> xn+1;

% Step 3L xn + 1 from xn

% Step 4: compare xn and xn+1 to determine if loop should continue
```

TOPIC 7: Building solutions to problems, part 2

ENGR105, 9/24/20

Tinkering with fundamental MATLAB operations

Practice problem:

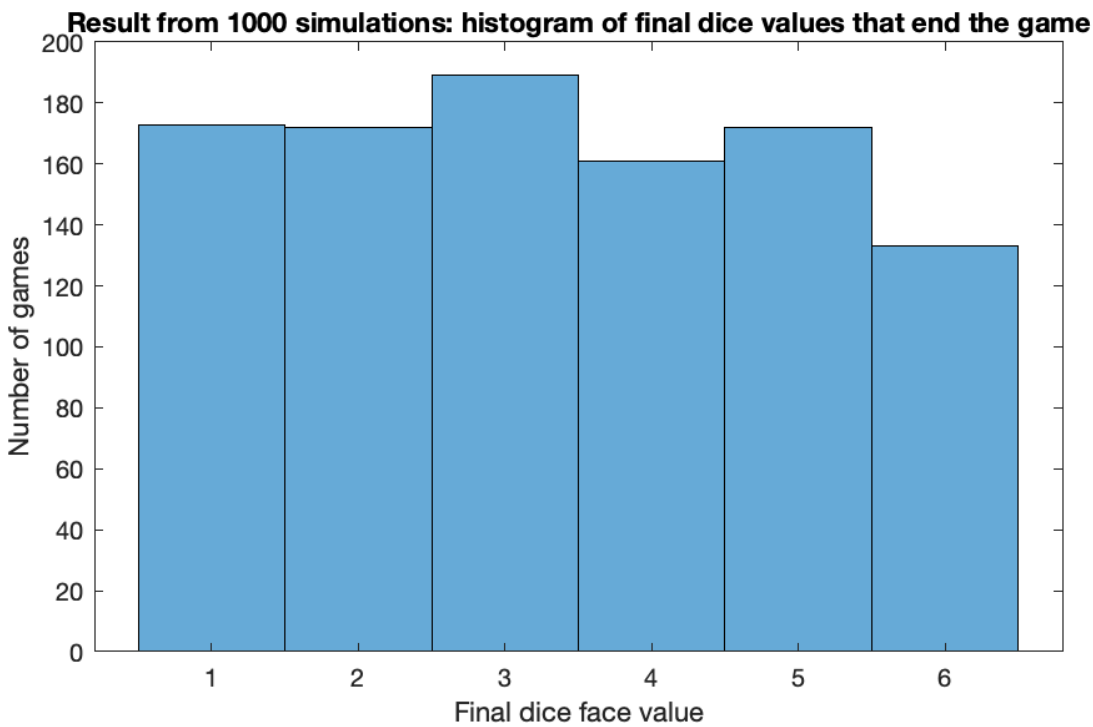
- We want to simulated rolling one dice until three of one number in a row show up.
- Repeat this simulation 1000 times.
- Plot the frequency that each side (1-6) is the side that comes up 3 times in a row. Plot the distribution of turns required to achieve three in a row.

1) Functions

- **What are functions good for?** Functions are ideal for consolidating your code into a standalone collection that can be executed in response to some inputs into the code.
- **When should you use a function?** You should use a function when need to repeat tasks with different inputs. Functions provide an organized way to perform a series of tasks with only the input specified.
- **How do inputs and outputs work?** You specify these in the function declaration -- when you call your function, you should match the form of your declaration. **See practice code.**
- **What if you do not care about all outputs?** You can disregard inputs with ~.
-

```
%The final form of the practice problem from class

% Preallocate storage vectors
```

```
numTurns = zeros(1,1000);      % Store requred number of turns from k-th game
diceFaceValue = zeros(1,1000); % Store final dice face value from the k-th game
for k = 1:1000
    [numTurns(k), diceFaceValue(k)] = diceRollSimulator(3,6);
end
histogram(diceFaceValue)
ylabel('Number of games')
xlabel('Final dice face value')
title('Result from 1000 simulations: histogram of final dice values that end
the game')
```

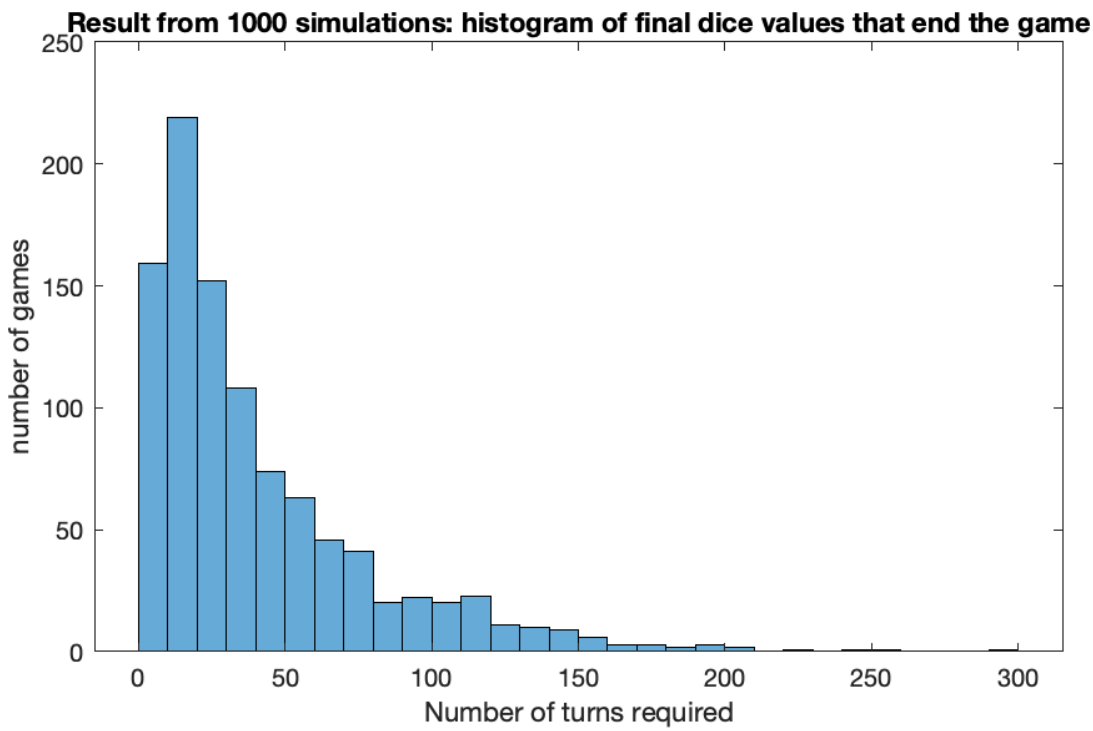


2) Conditional statements

- **How does the flow of information through conditional statements work?** Conditional statements provide structure that enables certain lines of code to run only if some logical condition is met.
- **When should you have an `else` statement?** You should include an else statement if you wish certain lines of code to run if all conditional statements provided in an `if-elseif-else` statement are not true.

```
% Plot 2 - plot distributuion of number of turns
histogram(numTurns)
```

```
ylabel('number of games')
xlabel('Number of turns required')
title('Result from 1000 simulations: histogram of final dice values that end
the game')
```



3) `for` loops

- **When do you use for loops?** Use a for loop when you know (or can calculate) how many iterations must be performed.
- **How does the index vector work?** The index vector simply specifies which values the loop index will take in each iteration. **See practice code.**

4) `while` loops

- **When do you use while loops?** Use a while loop when you do not know ahead of time how many loop iterations to perform.
- **How do you keep track of iterations (or specific iterations)?** Use a counter! This is a variable initialized outside of the loop that can be incremented in each iteration performed. **See practice code.**
- How does a stopping condition relate to what is happening in the loop?

```

%work while a conditional statement is true
```

```
n = 0;
a = 0;
while n<5
    %keep track of when n is odd to perform tasks
    if mod(n,2) == 1
        a = a+1;
    end
    n = n+1;

end
```

Practice problem 2: numerical methods and root-finding

Numerical analysis is the domain of mathematics and computer science that creates, analyzes, and implements algorithms for solving numerically the problems of continuous mathematics. Numerically solving a complex math problem often involves iterating or following a given algorithm until it converges (hopefully) upon a solution, within some defined tolerance.

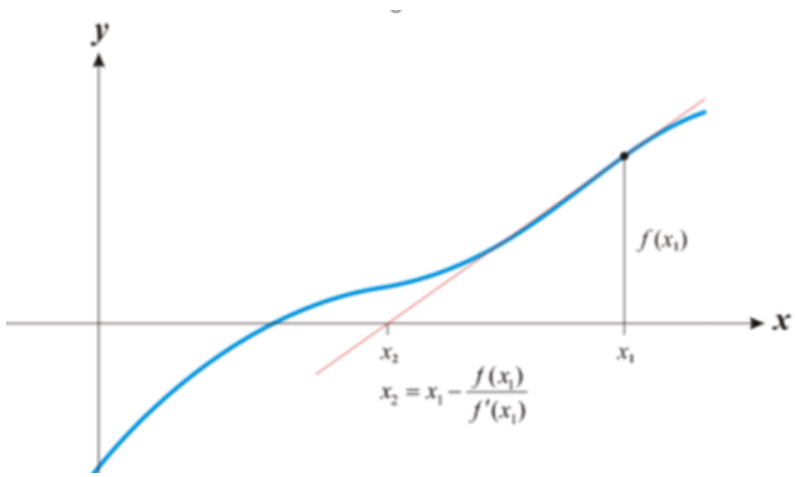
One such application of numerical methods is root-finding, the algorithms for which may quickly find the root ($F(x) = 0$) for some arbitrary function $F(x)$. Say we have a (potentially very nonlinear) function for which we want to find $F(x) = 0$ quickly. The Newton-Raphson method searches for the root of a function by iteratively following where the tangent of a function crosses the x axis.

Specifically, **the Newton-Raphson method has the following algorithm:**

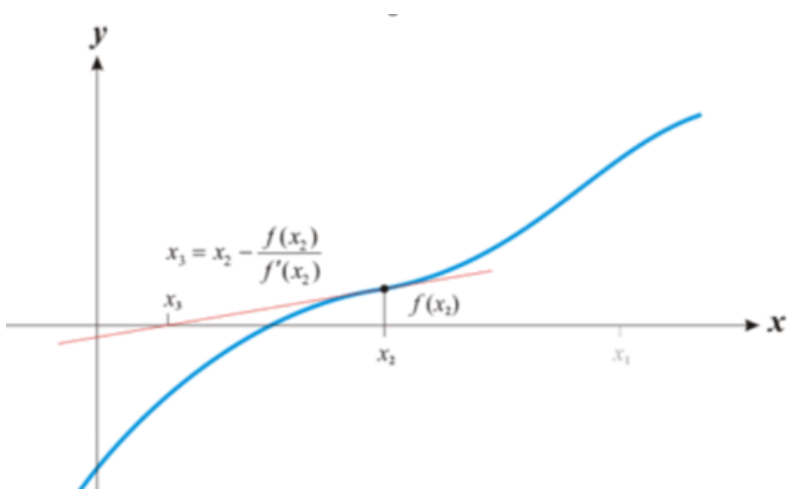
1. Pick an initial guess (x_1) of the root of a function.
2. At that x value, calculate the tangent to the curve of the function.
3. Find where the tangent to the curve intersects the x axis.This x value becomes x_2.
4. At this new x value, repeat steps 2-3.
5. When the new x value is not much different than the previous x value (as determined by some pre-defined tolerance), conclude that you have found an approximation for the root of the function. That is, you found a numerical approximation for the x value for which $F(x) = 0$.

Visual depiction of the Newton-Raphson algorithm

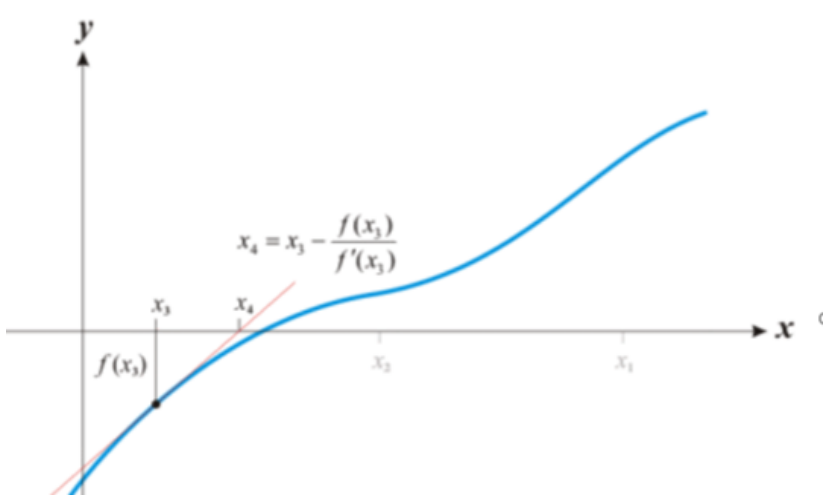
Iteration 1: Make an initial guess, calculate where the tangent line at that point intersects x axis.



Iteration 2: Using this new x value, repeat the process



Iteration 3: The algorithm is beginning to converge on the root of $F(x)$.



In these images, x_n is the current guess for the root and x_{n+1} is the next input in this iterative process. When x_n -> x_{n+1}, we have found the root of the function.

Derivation of the Newton-Raphson method

1) The tangent line of the function at some point *x* can be written generally as:

$$y = m\,x + b$$

2) The slope, *m*, is simply the derivative of the function at xn:

$$y = f'(x_n)\,x + b$$

3) At *x = xn, y = f(xn)*, which allows us to solve for *b*

$$f(x_n) = f'(x_n)x_n + b \longrightarrow b = f(x_n) - f'(x_n)x_n$$

4) Substituting, we can write an equation for the tangent line in terms of $f(x_n), f'(x_n), x_n$:

$$y = f'(x_n)x + (f(x_n) - f'(x_n)x_n)$$

5) In this algorithm, we solve for *y* = 0 for the tangent line. Thus, at y=0, and *x* = xn+1:

$$0 = f'(x_n)x_{n+1} + (f(x_n) - f'(x_n)x_n)$$

6) Rearranging the previous for xn+1 produces the Newton-Raphson equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Thus, we have derived a numerical method for iterating until we converge on the root of a function.

Question 1

Use the Newton-raphson method to find the roots to the following equation:

$$f(x) = x^4 + 2(x-2)^2 - 20$$

<u>PRO, Step 1</u>: <u>P</u>lan, **<u>P</u>**repare, and **<u>P</u>**seudocode

Helpful tasks:

- We know the equation we want to solve for, can we plot it to get an idea of where the roots might be?
- We know the equation and the algorithm, can we write explicitly write out what the algorithm will look. That is, here:

$$f(x) = x^4 + 2(x-2)^2 - 20$$
$$f'(x) = 4x^3 + 4(x-2)$$

- We know the algorithm, but how is that converted into code? Write out pseudocode for what the script will look like.

```
% Step 1: specify initial guess

% Step 2: iterate until xn -> xn+1

% Step 3: xn + 1 from xn

% Step 4: compare xn and xn+1 to dtermine if loop should continue
```

PRO, Step 2: w**R**ite **R**ough (increments of code) and **R**efine

Helpful tasks:

- Starting with your pseudocode and what you know about the function, roughly fill in code.

```
% "Rough Solution:" use a for loop to solve this problem

% Step 1: specify initial guess
xn = 3;

% Step 2: interate until xn -> xn+1
for k = 1:5
    %Step 3: evaluate function
    fn = xn^4 + 2*(xn-2)^2 - 20;

    % Step 3a: evaluate derivative
    f_prime_n = 4*xn^3 + 4*(xn-2);

    % Step 3b: calculate next x value
    xnplus1 = xn -fn/f_prime_n;

    % Step 4: update xn
    xn = xnplus1;
end
disp(xn)
```

```
2.1141
```

PRO, Step 3: **O**ptimize and improve

We don't actually know how many loop iterations to perform ahead of time. Refine the code so that it is in the form of a `while` loop.

```
% Improved solution: use a while loop to iterate until convergence
```

```
% We will start by revisiting on Monday. The first is the approach we were
% heading towards in class before we ran out of time: using an if statement
% inside the while loop.

% Step 1: specify initial guess and other variables
xn = 3; % initial guess
keepIterating = 1; % Variable that specifies if we should keep iterating.
% We will set the value to zero if we meet some stopping criteria.

% Step 2: iterate until xn -> xn+1
while keepIterating

    % Step 3: evaluate function
    fn = xn^4 + 2*(xn-2)^2 - 20;
    %Step 3a: evaluate derivative
    f_prime_n = 4*xn^3 + 4*(xn-2);

    %Step 3b: calculate the next value of xn
    xnplus1 = xn - fn/f_prime_n;

    % Step 4: Evaluate if we have reached a stopping criteria. Note the
    % form of this relational expression
    if abs(xn - xnplus1) < 10^-5
        % One approach: use an if statement
        keepIterating = 0;
    end

    % Step 5: update xn and repeat
    xn = xnplus1;
end
disp(xn)
```

```
2.1141
```

TOPIC 8: Revisiting arrays, pt. 1: useful math and logical arrays

ENGR105, 9/29/20

Revisiting arrays: useful structures for storing and processing numerical information

We have mostly discussed 1-D (vectors) and 2-D (matrices) arrays, but MATLAB works with N-dimensional arrays. Remember that all of the array concepts we discuss also apply to higher order arrays.

Reminder: Vector indexing

Vectors may be indexed using discrete vector indices or single index subscripts.

```
% Define initial vector
v = [5 3 4 6 8]
```

```
v = 1×5
     5     3     4     6     8
```

```
% Define subsequent vectors through vector indexing
p = v([1 5 4 2])
```

```
p = 1×4
     5     8     6     3
```

```
q = v(1:2)
```

```
q = 1×2
     5     3
```

```
r = v(end-2:end)
```

```
r = 1×3
     4     6     8
```

```
s = v(end:-1:end-3)
```

```
s = 1×4
     8     6     4     3
```

```
t = v(1:2:end)
```

```
t = 1×3
     5     4     8
```

Matrix (and array) indexing

Arrays may be indexed with reference to row, column, page, … etc. subscripts or using single index subscripts.

```
% Define initial matrix
M = [5 2 4; 3 7 5]
```

```
M = 2×3
     5     2     4
     3     7     5
```

```
% Create subsequent matrices through indexing
B = M(1:2,1:2)
```

```
B = 2×2
     5     2
     3     7
```

```
C = M(:,1)
```

```
C = 2×1
     5
     3
```

```
D = M([1,2],[2,3])
```

```
D = 2×2
     2     4
     7     5
```

```
F = M(1,end-1:end)
```

```
F = 1×2
     2     4
```

```
G = M(end:-2:1)
```

```
G = 1×3
     5     7     3
```

Adding, subtracting, multiplying and dividing arrays and their elements

We have learned how to perform mathematical operations on vectors and matrices, from element-wise multiplication and addition to matrix multiplication and division.

```
% Perform array math
B + D     % matrix addition
```

```
ans = 2×2
     7     6
    10    12
```

```
B.*D    % element-wise multiplication
```

```
ans = 2×2
    10     8
    21    35
```

```
B*M  % matrix multiplication
```

```
ans = 2×3
    31    24    30
    36    55    47
```

Other useful mathematical operations on arrays

Using arrays with built-in functions. As was the case with vectors - many built-in MATLAB functions operate on any N-dimensional arrays element by element. For example, if A is a 2x2 matrix, tan(A) will be 2x2 with each element being the tangent of the corresponding element of A.

```
A = [pi/4,3*pi/4;5*pi/4,7*pi/4]
```

```
A = 2×2
    0.7854    2.3562
    3.9270    5.4978
```

```
% Tangent operation on entire array
tan(A)
```

```
ans = 2×2
    1.0000   -1.0000
    1.0000   -1.0000
```

The same holds true for most of the common, built-in MATLAB functions (*e.g.* `abs(A)`, `sin(A)`, `log10(A)`)

Array sums. Recall that `sum()` of any vector yielded the sum of all elements in that vector.

```
a = 1:3
```

```
a = 1×3
     1     2     3
```

```
sum(a)
```

```
ans = 6
```

N-dimensional arrays may be summed across any of the dimensions or the entire array using the same function. For an N-dimensional array: the sum along the `N`-th dimension may be invoked using `sum(A,N)`, where `N` is the dimension number

- Dimension "1": rows
- Dimension "2": columns
- Dimension "3": pages (and so on)

```
% Create matrix
A = [1:3;4:6]
```

```
A = 2×3
     1     2     3
     4     5     6
```

```
% Sum across each row of A
sum(A,1)
```

```
ans = 1×3
     5     7     9
```

```
% Sum across each column of A
sum(A,2)
```

```
ans = 2×1
     6
    15
```

The **sum of all elements in the array** may be output by nesting `N` number of sum calls (ex. `sum(sum(A))` for a 2D matrix) - the sum starts with the innermost dimension for the most highly-nested sum call.

```
sum(sum(A))
```

```
ans = 21
```

You may also achieve the previous by unwrapping the matrix to a vector via `sum(A(:))`

```
sum(A(:))
```

```
ans = 21
```

Cumulative array sums. `cumsum()` sums cumulatively along a *vector* or *some dimension of an array* - unless a dimension is specified, it works on the first non-singleton dimension.

```
a = 1:4
```

```
a = 1×4
     1     2     3     4
```

```
%adds to the next index as it goes through
cumsum(a)
```

```
ans = 1×4
     1     3     6    10
```

```
A
```

```
A = 2×3
     1     2     3
     4     5     6
```

```
cumsum(A,2)
```

```
ans = 2×3
     1     3     6
     4     9    15
```

Array products. `prod()` calculates the product of array elements using the specified dimension. If no dimension is input, it takes the product along the first non-singleton dimension.

```
% Vector example
a
```

```
a = 1×4
     1     2     3     4
```

```
prod(a)
```

```
ans = 24
```

```
% Matrix example
A
```

```
A = 2×3
     1     2     3
     4     5     6
```

```
prod(A,2)
```

```
ans = 2×1
     6
   120
```

Cumulative array products. `cumprod()` calculates the cumulative product of array elements along a vector or some dimension of an array - unless a dimension is specified, it works on the first non-singleton dimension.

```
% Vector example
a
```

```
a = 1×4
     1     2     3     4
```

```
cumprod(a)
```

```
ans = 1×4
     1     2     6    24
```

```
% Array example
A
```

```
A = 2×3
     1     2     3
     4     5     6
```

```
cumprod(A,2)
```

```
ans = 2×3
     1     2     6
     4    20   120
```

Note: many other mathematical functions may be able to operate along rows, columns, or other dimension. Always check the MATLAB `help` documentation to determine how to input the dimension of interest.

Useful built-in arrays

MATLAB includes a lot of pre-defined matrices that may be scaled to fit your needs. Generally, there are two ways to specify the dimensions of these matrices:

- A single dimension (`n`) creates an `nxn` square matrix
- Two dimensions (n, m) create a `nxm` matrix

Some pre-defined matrices support `N`-dimensional arrays (ex. `rand(m,n,p)`and `randi(m,n,p)`)

```
% Create an array with all zeros
zeros(3,2)
```

```
ans = 3×2
     0     0
     0     0
     0     0
```

```
% Create a matrix with all ones
ones(4)
```

```
ans = 4×4
```

```
     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

```
% Create a matrix with equal row, column, and diagonal sums
magic(3)
```

```
ans = 3×3
     8     1     6
     3     5     7
     4     9     2
```

```
% Create an identity matrix
eye(3)
```

```
ans = 3×3
     1     0     0
     0     1     0
     0     0     1
```

```
% Create a matrix of random integers
randi(4,[2,3])
```

```
ans = 2×3
     1     4     2
     3     1     3
```

Logical Arrays

Logical numbers

Recall that we can use relational operators compare values and yield a `true` (`1`) or `false` (`0`) numerical result.

Relational operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
~=	Not equal to
>	Larger than
>=	Larger than or equal to

```
% Relational operator examples
an_1 = 5==2
```

```
an_1 = logical
   0
```

```
c = 5==2
```

```
c = logical
   0
```

```
% Relational operators can also operate on variables
dog = 4;
cat = 9;
animal = dog==cat
```

```
animal = logical
   0
```

Note: each of these variables is a **scalar of logical *class***. Each variable and number in MATLAB is of a certain data type. The logical data type specifies that a variable can have one of two possible values: either `1` (`true`) or `0` (`false`).

Logical arrays: vectors, matrices, etc. with logical numbers

A logical array is any n-dimensional array that contains only logical type numbers – `logical 0` (`false`) or `logical 1` (`true`). In practice, **logical arrays are most often created by subjecting an array to a relational operator**.

```
% Define arrays
M = [1,5,8;9,4,3]
```

```
M = 2×3
     1     5     8
```

```
     9     4     3
```

```
A = [7,2,6;3,0,5]
```

```
A = 2×3
     7     2     6
     3     0     5
```

In each of the following examples, **an array is created with the same size as the array subjected to the relational operator**. However, the entries are either `0` or `1`, depending on if the **relational statement is true or false when each corresponding element of the original array is operated on.**

```
% Creating logical arrays by performing relational operations
V = M == 5
```

```
V = 2×3 logical array
   0   1   0
   0   0   0
```

```
% Logical matrix based on A, specify which elements are greater than 6
r3 = A > 6
```

```
r3 = 2×3 logical array
   1   0   0
   0   0   0
```

```
% Create logical matrix indicating which elements in M are greater than the
% corresponding elements in A
out = M>A
```

```
out = 2×3 logical array
   0   1   1
   1   1   0
```

<u>Note</u>: as above, the output from each of these is an array of the **<u>logical</u>** class. A logical variable can only take two values: `0` or `1`. We will talk more about data types and classes next week.

A logical array can also be specified by passing an array through the `logical()` command - although this is more rarely used in practical application. In the following, the original vector is converted to a logical vector -- all nonzero elements are converted `logical 1`.

```
v = [6,4,0,1]
```

```
v = 1×4
     6     4     0     1
```

```
g2 = logical(v)
```

```
g2 = 1×4 logical array
   1   1   0   1
```

Practice Problem: tic tac toe

Rules of tic tac toe:

- The game starts with an empty `3x3` grid.
- Players take turn picking spots (with their marker, usually entering either "x" or "o") in the grid.Thus, there are a possible 9 turns before either one of the players wins or the game ends in a draw.
- If any player gets three spots in a row (horizontal, vertical, or diagonal), she or he wins.

<u>Goal</u>: produce a script that "plays" tic tac toe.

Simpler (initial) goal: produce a two player version of tic tac toe.

```
%choose x or o's to be 1 or zeros and then gerate random matrices
%in order to check how you win you need to check if all indices in a
%row/column/diagonal equal each other


%step 1: define initial variables
gameboard = zeros(3);
%create array to keep track of positions
availablePos = 1:9;

%Repeat: alternate between players
player = 1; %player 1
%checks who's turn it is
gameStillGoing = 1;
%iterate until someone has one, or game is a draw
availablePos
```

```
availablePos = 1×9
     1     2     3     4     5     6     7     8     9
```

```
while gameStillGoing
    %Step 2: make a move, play position
    %populate using single index notation

    %logical corresponding to valid move

    invalidMove = 1;
```

```
    while invalidMove
        playPos = input("Select a position to play...")
        invalidMove = prod(availablePos ~= playPos)
    end
    %Update gameboard
    gameboard(playPos) = player;
    %Step 3: display gameboard
    disp(gameboard)

    %Step 4: evaluate stopping condition
    gameStillGoing = checkEndGame(gameboard, player);
    %step 5 change player
    player = mod(player,2) + 1;

end
```

```
playPos = 1
invalidMove = 0
     1     0     0
     0     0     0
     0     0     0
playPos = 2
invalidMove = 0
     1     0     0
     2     0     0
     0     0     0
playPos = 4
invalidMove = 0
     1     1     0
     2     0     0
     0     0     0
playPos = 3
invalidMove = 0
     1     1     0
     2     0     0
     2     0     0
playPos = 7
invalidMove = 0
     1     1     1
     2     0     0
     2     0     0
1wins
```

TOPIC 9: Revisiting arrays, pt. 2: logical arrays and more

ENGR105, 10/1/20

Logical Arrays (cont'd)

Last time, we talked about how logical arrays can be created by relating arrays with relational operators.

```
% Define arrays
M = [1,5,8;9,4,3]
```

```
M = 2×3
     1     5     8
     9     4     3
```

```
A = [7,2,6;3,0,5]
```

```
A = 2×3
     7     2     6
     3     0     5
```

In each of the following examples, **an array is created with the same size as the <u>array subjected to the relational operator</u>**. However, the entries are either `0` or `1`, depending on if the **relational statement is true or false when each corresponding element of the original array is operated on.**

```
% Creating logical arrays by performing relational operations
v = M == 5
```

```
v = 2×3 logical array
   0   1   0
   0   0   0
```

```
% Logical matrix based on A, specify which elements are greater than 6
r3 = A > 6
```

```
r3 = 2×3 logical array
   1   0   0
   0   0   0
```

Indexing with logical arrays

Logical arrays can be used to index "regular" arrays.

```
% Define vector
v = [5,2,4]
```

```
v = 1×3
     5     2     4
```

```
% Define logical vector
a = logical([1 0 1])
```

```
a = 1×3 logical array
   1   0   1
```

NOTE: This is different than "regular" vector indexing. You are not creating a new vector by specifying the index to "call". You are specifying which "true" indices (as determined by the position in the logical array that is `1` (`true`)) to include in the final matrix.

Using *logical indexing*, we "select" elements based on the indices in the logical array that are `true`.

```
res = v(a) %since first and third entries are true you return those
```

```
res = 1×2
     5     4
```

NOTE: when using logical indexing, the index vector must be of the logical data type.

In this example, this is equivalent to the *vector indexing* that we have done previously:

```
res = v([1,3]) %enables us to select elemants that satisfy a condition
```

```
res = 1×2
     5     4
```

Logical array indexing rules

1. An array may be indexed with logical class 0's and 1's
2. Only the elements of the array that were indexed as 1's are returned or modified
3. The logical array that specifies the indices must be the same size as the array being indexed*

Examples with vectors

```
% Define two arrays
v = [6,5,2]
```

```
v = 1×3
     6     5     2
```

```
p = [9,5,3]
```

```
p = 1×3
     9     5     3
```

```
% Perform logical indexing
r = v(p==3)
```

```
r = 2
```

```
s = v(v>3)
```

```
s = 1×2
     6     5
```

```
% Create new vector consisting only of entries of p less than 4
val2 = p(p<=4)
```

```
val2 = 3
```

```
% Overwrite all entries whose value is greater than 3
v(v>3) = 7
```

```
v = 1×3
     7     7     2
```

```
% Delete entries from v based on logical indexing
v(p<6) = []
```

```
v = 7
```

Note: in all of these cases, **we modify the above vectors** or **create subvectors** without explicitly determing which elements fulfill some criteria before the operation. In this sense, the relational operator use is similar in spirit to while loops, in which we do not know ahead of time how many iterations to perform.

Examples with matrices

```
% Create example matrices
M = [3,7,6;5,4,2]
```

```
M = 2×3
     3     7     6
     5     4     2
```

```
A = [1,8,0;5,9,2]
```

```
A = 2×3
     1     8     0
     5     9     2
```

When using logical matrices, note the output is a ***vector***. It does not make sense to display the output as a matrix because some elements may not be present (all elements corresponding to `false`).

```
res = M(A>2)
```

```
res = 3×1
     5
     7
```

```
     4
```

Remember. Logical indexing must be performed with logical arrays.

```
%T = M(A)
out = M(logical(A))
```

```
out = 5×1
     3
     5
     7
     4
     2
```

```
% Two more examples
G = A(M==A)
```

```
G = 2×1
     5
     2
```

```
vals = A(M==7 | A==9)
```

```
vals = 2×1
     8
     9
```

In addition to **returning vectors** based upon logical indexing, matrix elements can be overwritten or redefined using logical indexing.

```
% Redefine matrices
M = [3,7,6;5,4,2]
```

```
M = 2×3
     3     7     6
     5     4     2
```

```
A = [1,8,0;5,9,2]
```

```
A = 2×3
     1     8     0
     5     9     2
```

```
% Set all elements less than 4 to be 9.
M(M<4) = 99
```

```
M = 2×3
    99     7     6
     5     4    99
```

```
% Specify element positions in M corresponding to those in which either A <
% 2 or A > 7 to equal 5
M(A<2 | A>7) = 5
```

```
M = 2×3
     5     5     5
     5     5    99
```

```
% Delete elements from A that are greater than 5
A(A>5) = []
```

```
A = 1×4
     1     5     0     2
```

Mathematical operations using logical arrays

Logical arrays formed by relational criteria may be used to selectively perform mathematical operations.

```
% Define vectors
v = [6,5,2]
```

```
v = 1×3
     6     5     2
```

```
p = [9,5,3]
```

```
p = 1×3
     9     5     3
```

```
% Create a vector with nonzero elements defined by the relational operator
res = (v>5).*p
```

```
res = 1×3
     9     0     0
```

```
% Note the importance of precedence!
res2 = v>5.*p
```

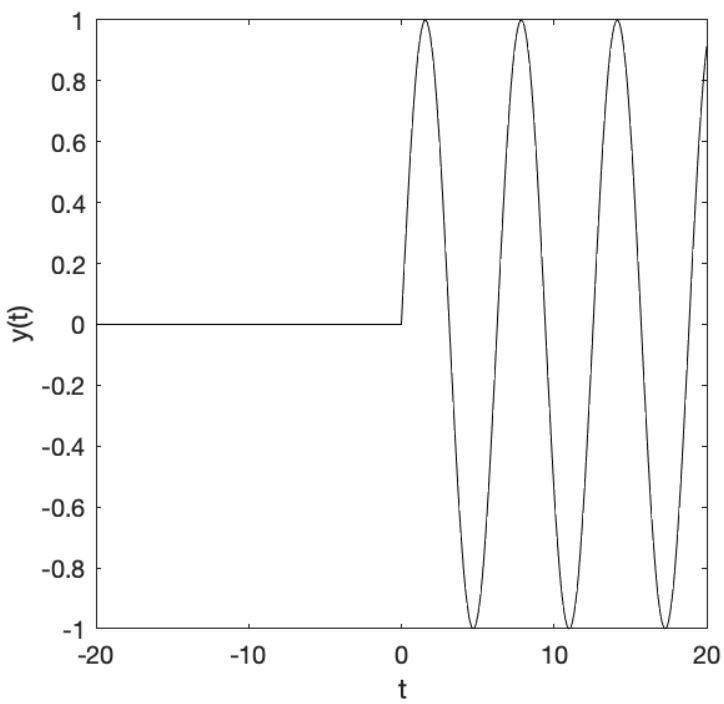
```
res2 = 1×3 logical array
   0   0   0
```

Example. In signal processing applications, we are often interested in *causal signals*, *i.e.* those that are zero for all times $t < 0$. Consider the causal $\sin(t)$ function.

$$y(t) = \begin{cases} \sin(t) & t \geq 0 \\ 0 & t < 0 \end{cases}$$

We can plot this by using the element-wise multiplication of a logical vector multiplied by the sine function.

```
% time vector
t = -20:.1:20;
% Produce a plot of the causal signal y(t)
plot(t,sin(t).*(t>=0),'k-')
% Define plotting parameters
xlabel('t')
ylabel('y(t)')
axis square % The plot window becomes square
ylim([-1,1]) % Set the y limits
```

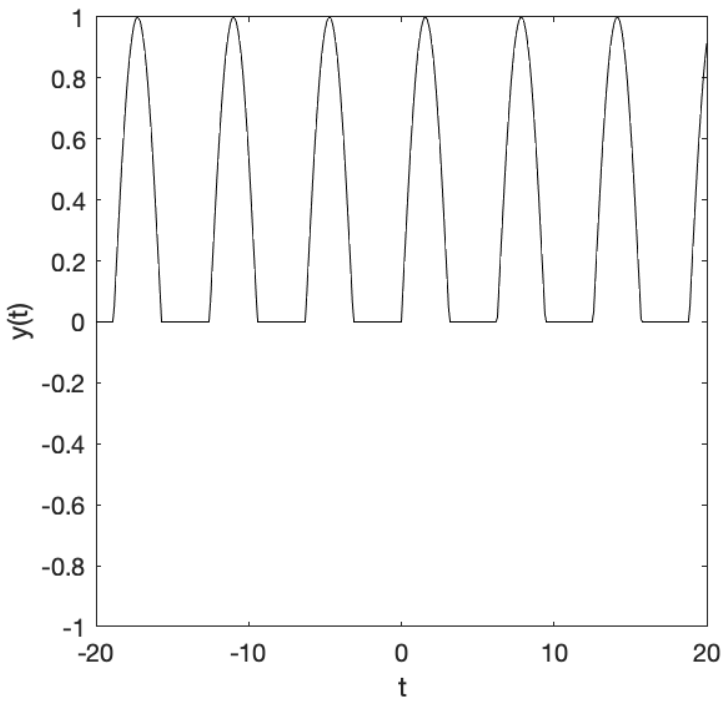


Example. Say we want to produce a half wave rectification of sine function.

$$y(t) = \begin{cases} \sin(t) & \sin(t) > 0 \\ 0 & \sin(t) \leq 0 \end{cases}$$

```
% time vector
t = -20:.1:20;
```

```
% Produce a plot of the causal signal y(t)
plot(t,sin(t).*(sin(t)>0),'k-')
% Define plotting parameters
xlabel('t')
ylabel('y(t)')
axis square % The plot window becomes square
ylim([-1,1]) % Set the y limits
```



Note: in both of the cases, multiplying the function by a logical array of the same size was a quick and efficient way to specify regions to be zero.

Built-in logical functions

- `any()` takes in an array or logical array and returns a scalar and logical `true` if any array values are `true`.

```
x = (1:4)>3
```

```
x = 1×4 logical array
   0   0   0   1
```

```
% TRUE if any element is true
any(x)
```

```
ans = logical
   1
```

- `all()` takes in an array or logical array and returns a scalar and logical `true` only if <u>all</u> values are `true`

```
x = (1:4)>3
```

```
x = 1×4 logical array
   0   0   0   1
```

```
% TRUE if all elements are true
all(x)
```

```
ans = logical
   0
```

- `isscalar()` takes an input and returns `true` if that input is a scalar.

```
x = 1:4
```

```
x = 1×4
     1     2     3     4
```

```
% TRUE if any element is true
isscalar(x)
```

```
ans = logical
   0
```

- `isempty()` takes an input and returns `true` if that input is an empty set (`[]`).

```
x = 1:4
```

```
x = 1×4
     1     2     3     4
```

```
% TRUE if any element is true
isempty(x)
```

```
ans = logical
   0
```

- `isnan()` takes an input and returns `true` if that input is not a number (`NaN`).

```
x = nan(1,3)
```

```
x = 1×3
   NaN   NaN   NaN
```

```
% Element-wise evaluation -> a corresponding index is 1 if that entry is
% NaN
isnan(x)
```

```
ans = 1×3 logical array
   1   1   1
```

Note: MATLAB has many other built-in logical functions. These can be found at:

https://www.mathworks.com/help/matlab/ref/is.html

Changing the shape of arrays

We have seen a few built-in functions for reshaping arrays, such as using the transpose (`'`). As a reminder, the transpose function rotates a matrix over its main diagonal, as seen here. This works for vectors and matrices.

```
% Create matrix
z = magic(3)
```

```
z = 3×3
     8     1     6
     3     5     7
     4     9     2
```

```
z'
```

```
ans = 3×3
     8     3     4
     1     5     9
     6     7     2
```

Note: transpose is not defined for >2D arrays in MATLAB.

Rotating and flipping arrays

MATLAB permits array rotations using the `rot90` and reflections using `fliplr`, `flipud`,and `flip`.

```
% Display matrix
z
```

```
z = 3×3
     8     1     6
```

```
     3     5     7
     4     9     2
```

```
% Rotate matrix 90 degrees counter clockwise
rot90(z)
```

```
ans = 3×3
     6     7     2
     1     5     9
     8     3     4
```

```
% Flip matrix "up and down," that is the last row becomes the first
flipud(z)
```

```
ans = 3×3
     4     9     2
     3     5     7
     8     1     6
```

```
% This is equivalent to reversing the row order, but requires less code
z(end:-1:1,:)
```

```
ans = 3×3
     4     9     2
     3     5     7
     8     1     6
```

```
% Flip matrix "left and right," that is the last column becomes the first
fliplr(z)
```

```
ans = 3×3
     6     1     8
     7     5     3
     2     9     4
```

```
% This is equivalent to reversing the column order, but requires less code
z(:,end:-1:1)
```

```
ans = 3×3
     6     1     8
     7     5     3
     2     9     4
```

```
% Flip the matrix along the specified dimension. This could work for higher
% order arrays
% 1: flip rows
```

```
% 2: flip columns
flip(z,1)
```

```
ans = 3×3
     4     9     2
     3     5     7
     8     1     6
```

Reshaping arrays

An array may be reshaped using the `reshape()`, which requires that the product of the reshaped dimensions equals the product of the original array dimensions.

```
z
```

```
z = 3×3
     8     1     6
     3     5     7
     4     9     2
```

```
z2 = reshape(z,1,9)
```

```
z2 = 1×9
     8     3     4     1     5     9     6     7     2
```

Note: the indices of the initial and reshaped matrix are determined from single index ("unwrapped") subscripts

TOPIC 10: Revisiting arrays, pt. 3: improving computational efficiency

ENGR105, 10/6/20

Logical arrays

Reminder: logical indexing and reshaping arrays

Last time, we talked about how logical arrays can be used to index vectors and matrices, incorporated into mathematical statements, and how arrays can be reshaped and restructured.

- We can perform **relational** and **logical operations on arrays** to create **logical vectors** for use in logical indexing or logical array math.

```
% Create a test arrays
a = 1:7
```

```
a = 1×7
     1     2     3     4     5     6     7
```

```
% Use logical indexing to select all elements greater than 3
a(a>3)
```

```
ans = 1×4
     4     5     6     7
```

```
% Use array math to set all elements that are NOT greater than 3 to 0.
a.*(a>3)
```

```
ans = 1×7
     0     0     0     4     5     6     7
```

- There are useful built-in logical functions for evaluating if **none, some, or all** entries in a logical vector are `true` or `false`.
-

```
%b is a logical vector so only has 0 and 1
b = a > 1
```

```
b = 1×7 logical array
   0   1   1   1   1   1   1
```

```
% Evaluate if at least one entry is true
any(b)
```

```
ans = logical
   1
```

```
% Evaluate if all entries are true
all(b)
```

```
ans = logical
   0
```

```
% Evaluate if no entries are true
all(-b)
```

```
ans = logical
   0
```

- We can also **restructure** and **rotate** arrays using built-in functions.

```
% Reshape array of elements larger than 3 into a 2x2 array.
b = reshape(a(a>3),2,2)
```

```
b = 2×2
     4     6
     5     7
```

```
% Rotate matrix 90 degrees ccw
rot90(b)
```

```
ans = 2×2
     6     7
     4     5
```

```
% Reverse the rows
flipud(b)
```

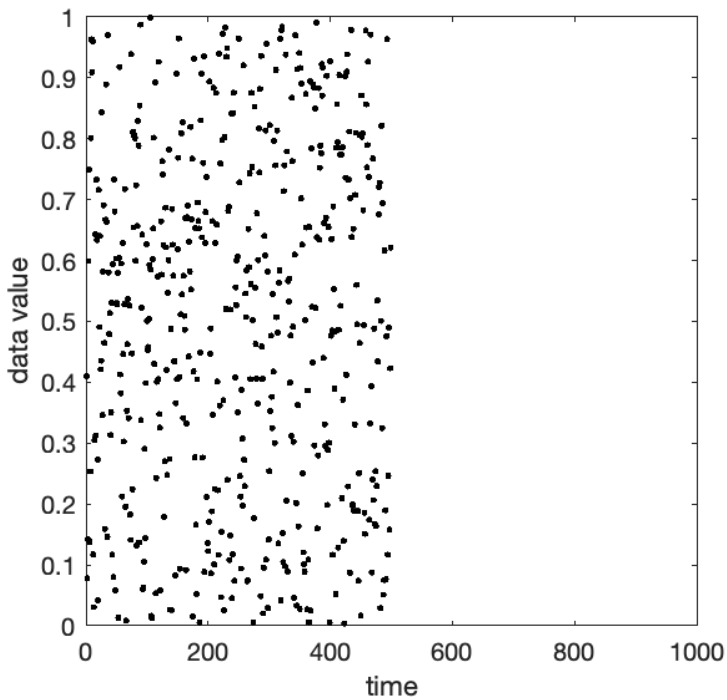
```
ans = 2×2
     5     7
     4     6
```

```
% Reverse the columns
fliplr(b)
```

```
ans = 2×2
     6     4
     7     5
```

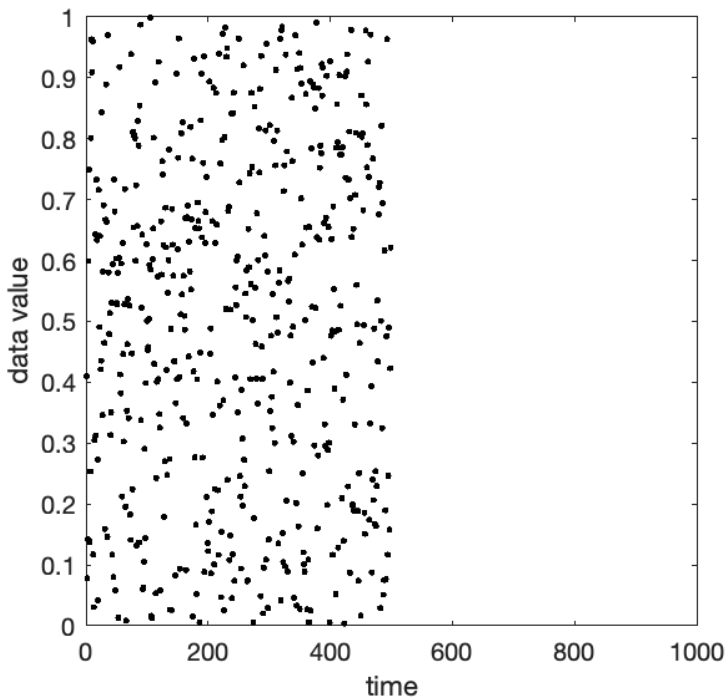
Plotting example. Assume we collected data points at many different time points and we wish to plot only a subset of these.

```
% Measured time points
t = 1:1000;
% Data points where each entry is the data point measured at the
% corresponding entry in the time vector.
dataPts = rand(1,length(t));

% Plot data
%plot for times less that 500
plot(t(t<500),dataPts(t<500),'k.')
axis square
ylim([0,1])         % y limit
xlim([0,length(t)]) % x limit
xlabel('time')      % data label for x axis
ylabel('data value')    % data label for y axis
```



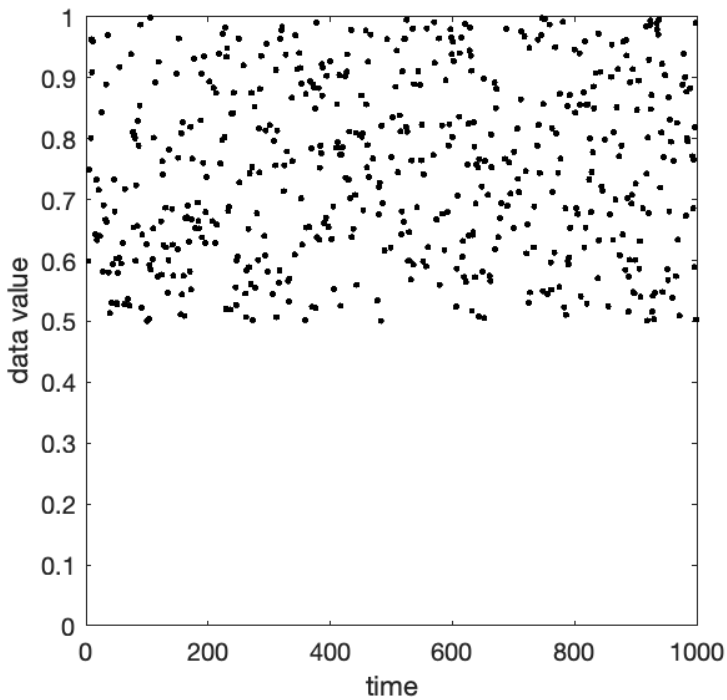
- Q1: plot data from times **less than 500 -> logical indexing**

```
% Plot data at t < 500
plot(t(t<500),dataPts(t<500),'k.')

axis square
ylim([0,1])         % y limit
xlim([0,length(t)]) % x limit
xlabel('time')      % data label for x axis
ylabel('data value')    % data label for y axis
```



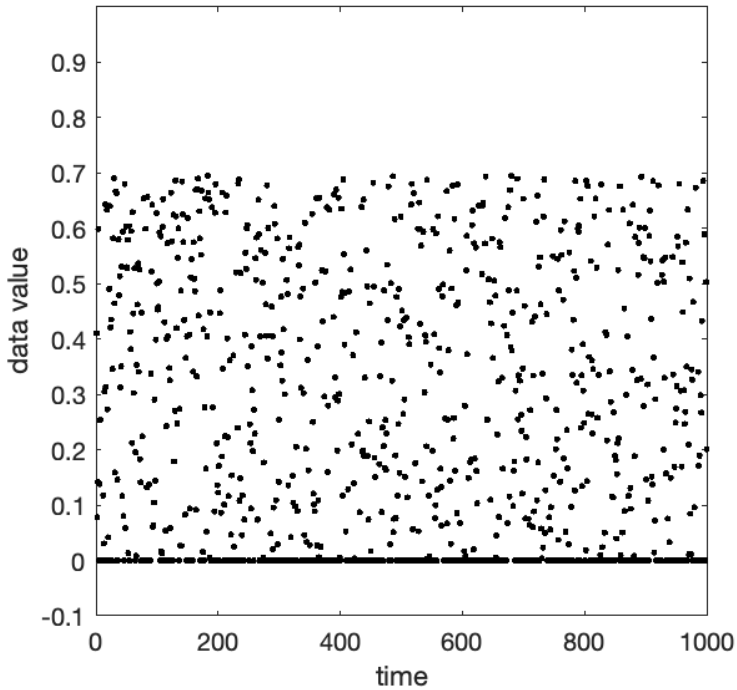
- Q2: plot data points with **values greater than 0.5 -> logical indexing**

```
% Plot data with values greater than 0.5
plot(t(dataPts>0.5),dataPts(dataPts>0.5),'k.')

axis square
ylim([0,1])         % y limit
xlim([0,length(t)]) % x limit
xlabel('time')      % data label for x axis
ylabel('data value')    % data label for y axis
```



- Q3: plot data points with with values **greater than 0.7 set to 0 -> logical math**

```
% Plot data with values greater than 0.7 set to 0

plot(t,dataPts.*(dataPts<=0.7),'k.')
axis square
ylim([-0.1,1])         % y limit
xlim([0,length(t)]) % x limit
xlabel('time')      % data label for x axis
ylabel('data value')    % data label for y axis
```



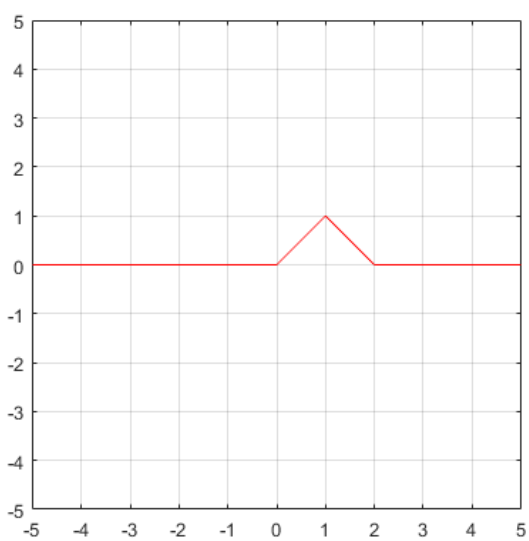
<u>NOTE</u>: in all of these example, <u>note the agreement</u> between the "x" and "y" inputs into the `plot()` function.

Question: produce the following piece-wise signal.

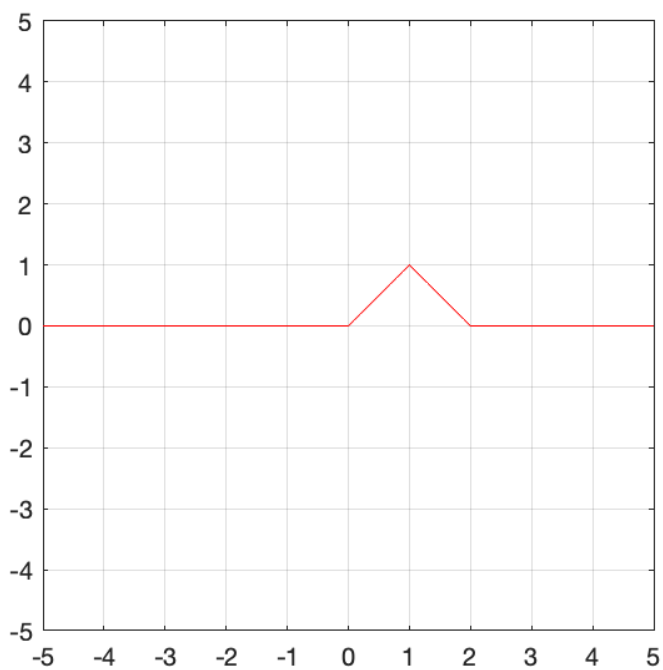
Relevant UPenn Course: BE 301: Signals and Systems

Relevant UPenn Course: MEAM 321 Dynamic Systems and Control

Relevant UPenn Course: CBE 360 Chemical Process Control



```
% Create the time vector
t = -5:0.1:5;
% Create the signal vector
x = t.*(t>=0).*(t<1) + (2-t).*(t>=1).*(t<2);

% Plot
plot(t,x,'r-')
axis square
ylim([-5,5])
xlim([-5,5])
grid on   % Display the grid
xticks(-5:5) % Explicitly define the x tick marks
yticks(-5:5)
```



Logical arrays vs. `if-else` statements

Example #1. The bonus for barkers selling peanuts at a baseball game is set at $0.02/pack if at least 100 packs were sold, $0.04/pack of at least 200 packs were sold, and $0.06/pack if at least 300 packs were sold. Calculate the bonus for four barkers who sold individually sold 57, 299, 161, and 429 packs of peanuts at a game.

```
% Create vector, each entry corresponds to a different seller's packs sold
packsSold = [57, 299, 161, 429];
```

So far, our strategy to accomplish this would be to create `if-else` statements nested in a loop.

```
bonus = zeros(1,4);
% Calculate bonuses.
for jj = 1:length(packsSold)

    % If >=100 and <200 packs sold
    if packsSold(jj)>=100 && packsSold(jj)<200
        bonus(jj) = .02*packsSold(jj);

    % If >=200 and <300 packs sold
    elseif packsSold(jj)>=200 && packsSold(jj)<300
        bonus(jj) = .04*packsSold(jj);

    % If >300 packs sold
    elseif packsSold(jj)>=300
        bonus(jj) = .06*packsSold(jj);
    end
end
disp(bonus)
```

```
         0   11.9600    3.2200   25.7400
```

A more concise (and easy to understand) way to calculate this would be to use logical array operations and math to perform all of the above calculations in one MATLAB statement.

```
% Calculate bonus using logical indexing
%make a logic based function that only multiplies agains thtings that were
%true
bonus2 = ...
    0.02*packsSold.*(packsSold>= 100 & packsSold<200) + ...
    0.04*packsSold.*(packsSold>= 200 & packsSold<300) + ...
    0.06*packsSold.*(packsSold>= 300);
disp(bonus2)
```

```
         0   11.9600    3.2200   25.7400
```

In this example, we add together three logical vectors (corresponding to each of the bonus conditions) that are multiplied by the corresponding dollar bonus.

Example #2. Employees of a business are paid at a rate of $19/hr and receive overtime (time and a half) above 40 hours/week. Calculate the pay for employees who worked 45, 21, 19, 61, and 39 hours last week.

```
%define verctor containing employee hours
employeeHours = [45, 21, 19, 61, 39];
%calculate pay in one statement
eployeePay = 10*employeeHours + 19*0.5*(employeeHours-40).*(employeeHours>40)
```

```
eployeePay = 1×5
```

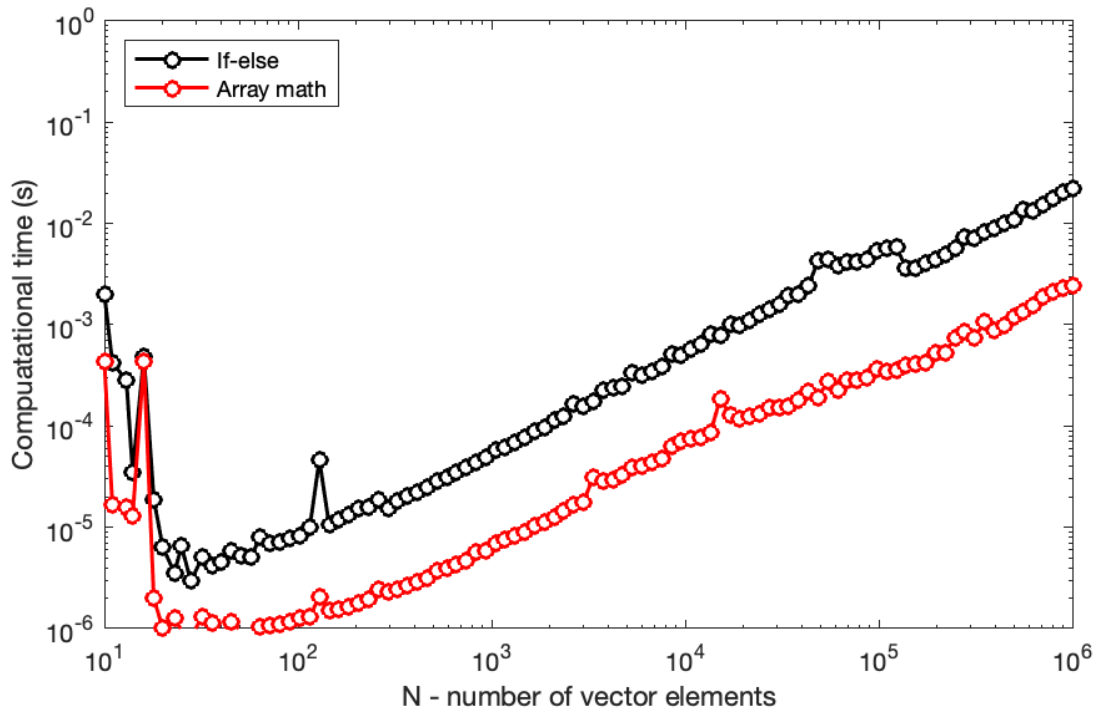
```
   497.5000  210.0000  190.0000  809.5000  390.0000
```

```
%at the end there is a logical that only multiplies if there is more than
%40 hours
```

Example #3. Instead of four people selling peanuts, you have N barkers selling packs of peanuts at sporting events in your multi-national event management corporation. Compare the computational time using if-else statements and logical arrays for determining the bonus pay each barker should receive, as a function of N, the number of barkers. Assume employees can sell 1 - 500 packs at a given event.

```
% Vector containing different numbers of barkers. We will calculate the
% time to calculate the bonus for increasing N using if-else vs. logical arrays
N = round(logspace(1,6,100));

% Preallocate vectors to store computational time
t_ifElse = zeros(1,length(N));
t_logicalMath = zeros(1,length(N));

% Calculate computational time for vector of different numbers of barkers
for k = 1:length(N)

    % Generate vector containing packs sold. Each entry corresponds to the
    % packs sold by a different employee.
    packsSold = randi(500,[1,N(k)]);

    % Calculate time using conditional statement
    tic;
    % Preallocate bonus array
    bonus1 = zeros(1,length(packsSold));

    % Calculate bonus received for each employee
    for jj = 1:length(packsSold)

        % If >=100 and <200 packs sold
        if packsSold(jj)>=100 && packsSold(jj)<200
            bonus(jj) = .02*packsSold(jj);

        % If >=200 and <300 packs sold
        elseif packsSold(jj)>=200 && packsSold(jj)<300
            bonus(jj) = .04*packsSold(jj);
```

```
        % If >300 packs sold
        elseif packsSold(jj)>=300
            bonus(jj) = .06*packsSold(jj);
        end
    end
    t_ifElse(k) = toc;


    % Calculate time using logical array operations and math
    tic;
    % Calculate bonus in one statement
    bonus2 = ...
        0.02*packsSold.*(packsSold>=100 & packsSold<200) + ...
        0.04*packsSold.*(packsSold>=200 & packsSold<300) + ...
        0.06*packsSold.*(packsSold>=300);
    t_logicalMath(k) = toc;
end
% Define dimensions of plot
figure('Position',[0,0,500,300]);
% Plot the functions
plot(N,t_ifElse,'ko-',N,t_logicalMath,'ro-
','LineWidth',1.5,'MarkerFaceColor','auto')
% Graphical parameters
set(gca, 'XScale', 'log')   % "log" axes - x
set(gca, 'YScale', 'log')   % "log" axes - y
set(gca,'XMinorTick','on')
xlim([10^1,10^6]);  % range of x values on plot
ylim([10^-6,10^0]); % range of y values on plot
xlabel('N - number of vector elements');
ylabel('Compuatational time (s)');
legend('If-else','Array math','Location','northwest')
```



<u>Takeaway</u>: creating simple statements using logical arrays can be more efficient and easier to understand.

Always preallocate arrays

As we saw with vectors, preallocation is the process of creating an array of the final size (usually padded with zeros) before modifying each element of the array. **Consider the computational time associated with creating an NxN multiplication table** using for loops for different numbers of N.

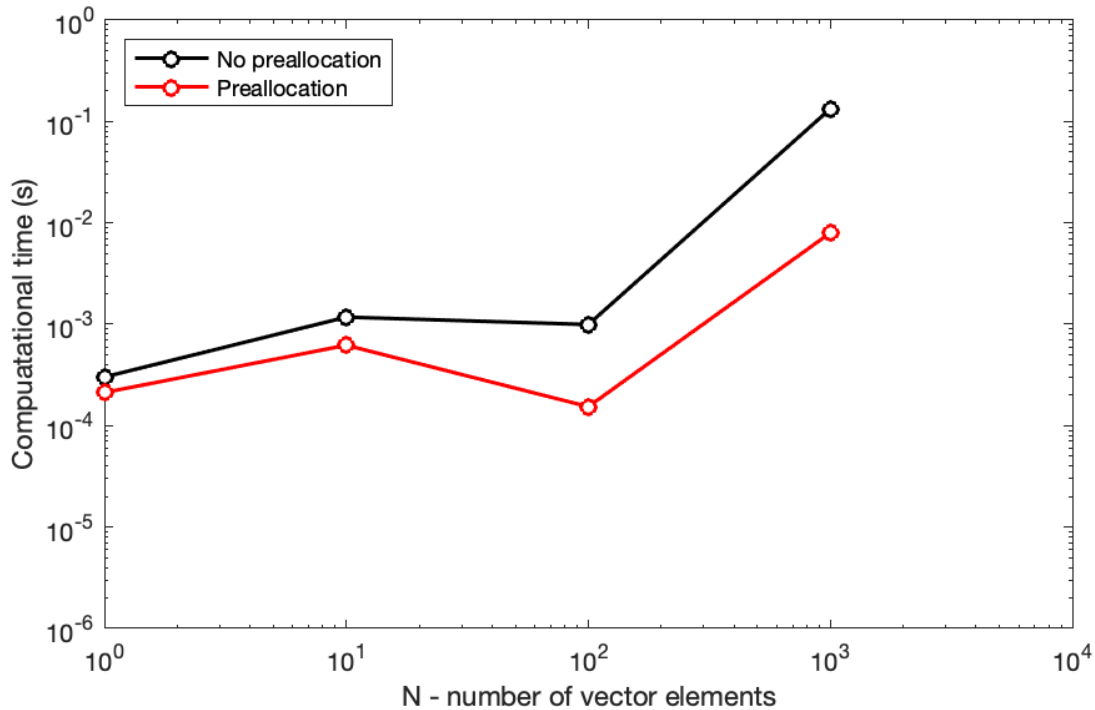
That is, what we want to create:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \cdots \\ 2 & 4 & 6 & 8 & 10 & \cdots \\ 3 & 6 & 9 & 12 & 15 & \cdots \\ 4 & 8 & 12 & 16 & 20 & \cdots \\ 5 & 10 & 15 & 20 & 25 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

```
% Determine range of N
N = logspace(0,3,4);

% Preallocate vectors to store computational time
t_no = zeros(length(N),1);
t_yes = zeros(length(N),1);

% For each number of elements considered
```

```
for k = 1:length(N)

    % Calculate time without preallocation
    tic;
    % For each row
    for jj = 1:N(k)
        % For each column
        for mm = 1:N(k)
            % Each entry is the produce of the rows and columns of the
            % multiplication table
            multTable(jj,mm) = jj*mm;
        end
    end
    t_no(k) = toc;

    % Calculate time without preallocation
    tic;
    % Preallocate array
    multTable = zeros(N(k),N(k));

    % For each row
    for jj = 1:N(k)
        % For each column
        for mm = 1:N(k)
            % Each entry is the produce of the rows and columns of the
            % multiplication table
            multTable(jj,mm) = jj*mm;
        end
    end
    t_yes(k) = toc;

    % Delete multiplication table so that it can be created from scratch on
    % the next loop
    clear multTable
end
% Define dimensions of plot
figure('Position',[0,0,500,300]);
% Plot the functions
plot(N,t_no,'ko-',N,t_yes,'ro-','LineWidth',1.5,'MarkerFaceColor','auto')
% Graphical parameters
set(gca, 'XScale', 'log')   % "log" axes - x
set(gca, 'YScale', 'log')   % "log" axes - y
set(gca,'XMinorTick','on')
xlim([10^0,10^4]);  % range of x values on plot
ylim([10^-6,10^0]); % range of y values on plot
xlabel('N - number of vector elements');
```

```
ylabel('Compuatational time (s)');
legend('No preallocation','Preallocation','Location','northwest')
```



<u>Side note</u>: the above multiplication table can be create **much more efficiently** using **matrix multiplication** instead of nested for loops.

```
multTable = (1:10)'*(1:10)
```

```
multTable = 10×10
     1     2     3     4     5     6     7     8     9    10
     2     4     6     8    10    12    14    16    18    20
     3     6     9    12    15    18    21    24    27    30
     4     8    12    16    20    24    28    32    36    40
     5    10    15    20    25    30    35    40    45    50
     6    12    18    24    30    36    42    48    54    60
     7    14    21    28    35    42    49    56    63    70
     8    16    24    32    40    48    56    64    72    80
     9    18    27    36    45    54    63    72    81    90
    10    20    30    40    50    60    70    80    90   100
```

Sparse arrays

Sparse matrices -- large arrays with few nonzero entries -- are encountered in a broad set of engineering disciplines, with applications including solving PDEs, search algorithms, atomistics, and FEA.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.8 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Consider a sparse matrix of dimensions 1000 x 2000 elements - that correlates to 2 million data points. Assume that this matrix contains non-zero values at the following indices:

- (row 3, column 4) = 15
- (row 100, column 1500) = 5
- (row 1000, column 2000) = 9

This matrix may be established using a conventional matrix indexing:

```
A = zeros(1000,2000);
A(3,4) = 15;
A(100,1500) = 5;
A(1000,2000) = 9;
```

Sparse matrices can be stored efficiently as a matrix of 0's with nonzero values stored by their row, column, and value – this can save significant memory space. In the above example, this matrix could also be established using a sparse matrix syntax:

```
B = sparse([3,100,1000],...
    [4,1500,2000],...
    [15,5,9],1000,2000);
```

Or, the original non-sparse matrix `A` could also be converted to a sparse matrix:

```
B = sparse(A);
```

The sparse matrix offers considerable savings for memory. Information about each variable can be depicted using the `whos` command.

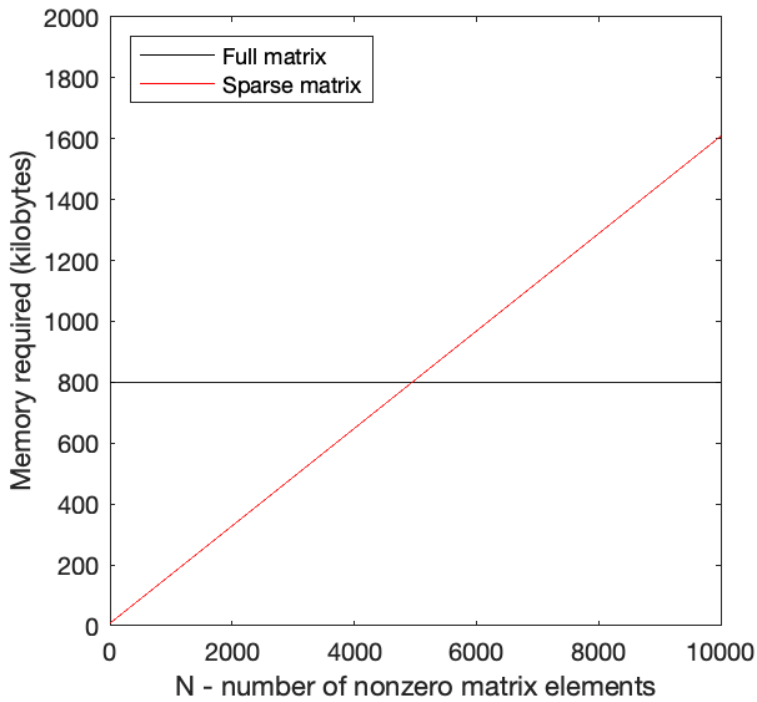
```
whos A B
```

```
  Name         Size                 Bytes  Class     Attributes

  A         1000x2000            16000000  double
  B         1000x2000               16056  double    sparse
```

In this case, variable A requires 16 MB of memory to create the **full matrix**, whereas variable B (**the sparse array**) only requires 16 kB! Dramatic savings in memory can be observed when most of the elements are zero.

At what number of nonzero elements do sparse matrices lose this advantage? Let's plot the memory associated with a matrix variable for different numbers of nonzero elements in a 100x100 matrix.

```
% Define a matrix intially with all zeros
A = zeros(100,100);

% Preallocate vector to store memory
mem_Full = zeros(1,numel(A));
mem_Sparse = zeros(1,numel(A));

% Repeat for all elements
for k = 1:numel(A)
    A(k) = randi(5);

    A_sparse = sparse(A);

    matrixInfo = whos('A','A_sparse');
    mem_Full(k) = matrixInfo(1).bytes;
    mem_Sparse(k) = matrixInfo(2).bytes;
end
% Plot the functions
plot(1:numel(A),mem_Full/100,'k-',1:numel(A),mem_Sparse/100,'r-')

% Graphical parameters
axis square
xlim([0,10^4]);  % range of x values on plot
ylim([0,2*10^3]); % range of y values on plot
xlabel('N - number of nonzero matrix elements');
ylabel('Memory required (kilobytes)');
legend('Full matrix','Sparse matrix','Location','northwest')
```



In addtion to memory, a similar dramatic savings in computational time when working with sparse arrays can be observed by using `tic` and `toc`, as we have done previously.

Note: if you need to convert a sparse matrix to a full matrix, this can be accomplished using `full()`.

TOPIC 11: Revisiting arrays, pt. 4; function handles

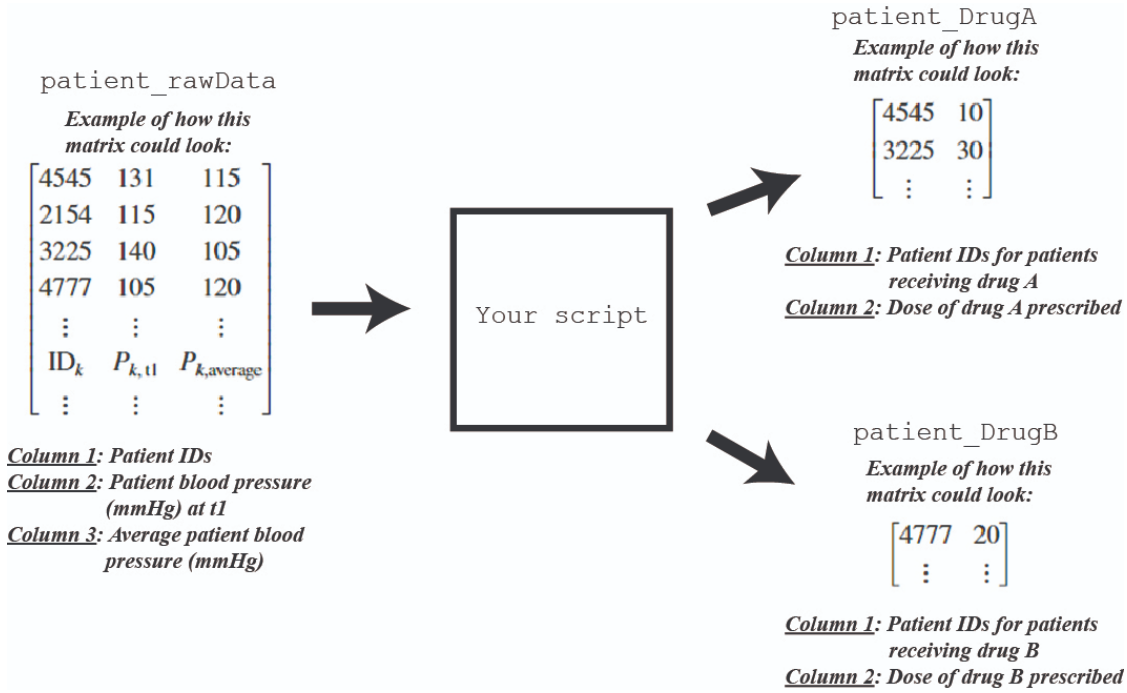
ENGR105, 10/8/20

What makes a good computational solution:

1. **Correct**: it works for all (or at least a specific set) of conditions
2. **Clear**: it is well-commented, concise, easy to understand, readable
3. **Computationally efficient**: it performs a task quickly while minimizing any unnecessary operations

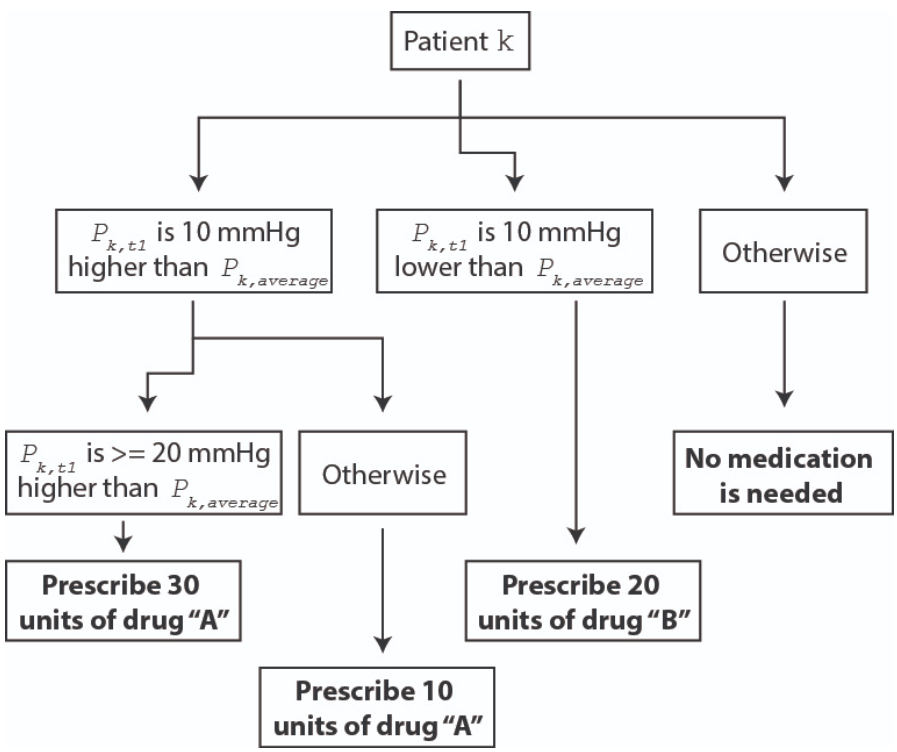
Practice Problem: process engineering in a medical environment

As a process engineer in a hospital, your task is to produce a computational framework for assigning doses of drugs to patients based upon their current (relative to prior) blood pressure. As described in the following schematic, your script should extract information from a starting matrix, perform some computations, and produce two output matrices.



In this problem, the input is an **Nx3 matrix**, where **N is the number of patients with the aforementioned illness** (where N changes daily). Call this matrix: **patient_rawData**

Furthermore, assume each patient in the database has the same medical condition: that patient's **systolic blood pressure (relative to that patient's baseline)** determines if that patient should receive **drug A, drug B,** or **no medication**. The decision tree describing how doses are assigned to *each* patient is below:



<u>Goal</u>: Create a script to convert **patient_rawData** into <u>two</u> new matrices: one matrix (**patient_DrugA**) for all patients receiving **drug A**, and another matrix (**patient_DrugB**) for all patients receiving **drug B**.

- Both **patient_DrugA** and **patient_DrugB** should have two columns. As with **patient_rawData**, each row in **patient_DrugA** and **patient_DrugB** should correspond to a unique patient.
- For both **patient_DrugA** and **patient_DrugB**, the <u>first column</u> should contain the patient IDs and the <u>second column</u> should contain the doses of drug to be administered to the corresponding patients.
- **patient_DrugA** should only contain patients who should be assigned Drug A and **patient_DrugB** should only contain patients who should be assigned Drug B.
- If a patient should not be prescribed either drug, then they should **<u>NOT</u>** show up in either **patient_DrugA** or **patient_DrugB.**

Pre-solution code: before solving this problem, let's create a "test" patient_raw Data matrix:

```
% Create a test matrix of containing simulated patient information
N = 10; % Number of patients

% Input matrix containing patient information
patient_rawData = [(1:N)',randi(50,[N,2])+100];
disp(patient_rawData)
```

```
     1   139   138
     2   134   137
     3   142   116
     4   101   120
     5   139   119
     6   149   125
     7   141   141
     8   122   130
     9   112   133
    10   135   109
```

Solution 1

- **Define output matrices as empty initially**
- **Add the appropriate patient ID and dose (determined from code) to "next row" of output matrices**

```
% Step 1: Define output arrays - we will populate these as we sequentially
% go through each for loop.

patient_DrugA = [];
patient_DrugB = [];


% Step 2: For each patient, compare their blood pressure at time t1 to
% their average blood pressure to determine if they should receive meds
```

```
% Use a FOR loop because we know how many iterations to perform -> one for
% each row of patient_rawData because each row in patient_rawData
% corresponds to one patient
for k = 1:size(patient_rawData, 1)

    % NOTE: the columns contain the following information:
    % patient_rawData(k,1): four-digit patient ID
    % patient_rawData(k,2): P_t1: blood pressures of patients at time t1
    % patient_rawData(k,3): P_avg: average recent patient blood pressures

    % if P_t1 - P_avg >= 10, determine if a dose of 10 or 30 should be
    % applied
    if patient_rawData(k,2) - patient_rawData(k,3) >= 10

    % if P_t1 - P_avg >= 20 -> assign to patient_DrugA and presribe a dose
    % of 30
        if patient_rawData(k,2) - patient_rawData(k,3) >= 10

            % Add a row to patient_DrugA, with the first entry being the
            % patient ID and the second being a dose of 30
            patient_DrugA(end+1,1:2) = [patient_rawData(k,1), 30];


        % if P_t1 - P_avg >= 10 -> assign to patient_DrugA and presribe a dose
        % of 10
        else
            % Add a row to patient_DrugA, with the first entry being the
            % patient ID and the second being a dose of 10
            patient_DrugA(end+1,1:2) = [patient_rawData(k,1), 10];
        end


    % if P_t1 - P_avg < 10 -> assign to patient_DrugB and presribe a dose
    % of 20
    elseif patient_rawData(k,2) - patient_rawData(k,3) <= -10

        % Add a row to patient_DrugA, with the first entry being the
        % patient ID and the second being a dose of 10
        patient_DrugB(end+1, 1:2) = [patient_rawData(k,1), 20];

    end
end

disp(patient_DrugA)
```

```
     3    30
     5    30
     6    30
    10    30
```

```
disp(patient_DrugB)
```

```
     4    20
     9    20
```

Common issues

- **Array indexing.** Be careful with the indexing inside of the `for` loop. This will define which entries of the output matrices are populated.
- **`If-elseif` statements vs. nested `if` statements:** in general, try to avoid nested `if` statements. These are fine if they are clear, but my experience has been that simpler `if-elseif` statements can more concisely convey a piece of code.
- **For loop syntax.** The index of the `for` loop does not need to be defined prior to the for loop. Also, specifying the loop to be performed for the loop index equaling one number is not valuable -- we want to specify a vector over which the index may be repeated.
- **For loops vs. while loops**. When should you use one vs. the other? If you use a while loop when a for loop is more appropriate, it may cause unintended confusion.
- **Built-in array functions**. What are the advantages and disadvantages of different ones? When should you use `length()` vs. `size()`. In general, I am against memorizing functions. However, these two are extremely useful if used correctly.

In other words, just solving a problem is not the only thing that matters. It also matters *how* you solve the problem.

Solution 2

- **Preallocate output matrices with the same number of rows as `patient_rawData`**
- **Add the appropriate patient ID and dose (determined from code) to the next row using a counter**
- **Delete all empty (ununsed) rows**

```
% Step 1: Define output arrays - we will populate these as we sequentially
% go through each for loop.

patient_DrugA = zeros(size(patient_rawData,1),2);
patient_DrugB = zeros(size(patient_rawData,1),2);;


%create counters to keep track or rows for the patients
rowA = 1;
rowB = 1;

% Step 2: For each patient, compare their blood pressure at time t1 to
% their average blood pressure to determine if they should receive meds
```

```
% Use a FOR loop because we know how many iterations to perform -> one for
% each row of patient_rawData because each row in patient_rawData
% corresponds to one patient
for k = 1:size(patient_rawData, 1)

    % NOTE: the columns contain the following information:
    % patient_rawData(k,1): four-digit patient ID
    % patient_rawData(k,2): P_t1: blood pressures of patients at time t1
    % patient_rawData(k,3): P_avg: average recent patient blood pressures

    % if P_t1 - P_avg >= 10, determine if a dose of 10 or 30 should be
    % applied
    if patient_rawData(k,2) - patient_rawData(k,3) >= 10

    % if P_t1 - P_avg >= 20 -> assign to patient_DrugA and presribe a dose
    % of 30
        if patient_rawData(k,2) - patient_rawData(k,3) >= 10

            % Add a row to patient_DrugA, with the first entry being the
            % patient ID and the second being a dose of 30
            patient_DrugA(end+1,1:2) = [patient_rawData(k,1), 30];


        % if P_t1 - P_avg >= 10 -> assign to patient_DrugA and presribe a dose
        % of 10
        else
            % Add a row to patient_DrugA, with the first entry being the
            % patient ID and the second being a dose of 10
            patient_DrugA(end+1,1:2) = [patient_rawData(k,1), 10];
        end


    % if P_t1 - P_avg < 10 -> assign to patient_DrugB and presribe a dose
    % of 20
    elseif patient_rawData(k,2) - patient_rawData(k,3) <= -10

        % Add a row to patient_DrugA, with the first entry being the
        % patient ID and the second being a dose of 10
        patient_DrugB(end+1, 1:2) = [patient_rawData(k,1), 20];

    end
end

disp(patient_DrugA)
```

```
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     3    30
     5    30
     6    30
    10    30
```

```
disp(patient_DrugB)
```

```
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     0     0
     4    20
     9    20
```

Solution 3

- **Use logical indexing to directly create the output matrices**
- **This is the "vectorization" equivalent**

```
% Create matrix for patients recieiveing drug A

%create submatrix consisting only of patients that need Drug A
patient_DrugA = patient_rawData(patient_rawData(:,2) - patient_rawData(:,3) >=
10,:);

%Use logival array indexing to determine the dose of drugs each patient
%should recive. Create a new column to clculate doses based on the
%patient's blood pressure and avg blood pressure

patient_DrugA(:,4) = 10 + 20*(patient_DrugA(:,2) - patient_DrugA(:,3) >= 20);

%Delete the unnecessary columns -> we do not want columns corresponding to
patient blood pressure or the avg blood pressure.
```

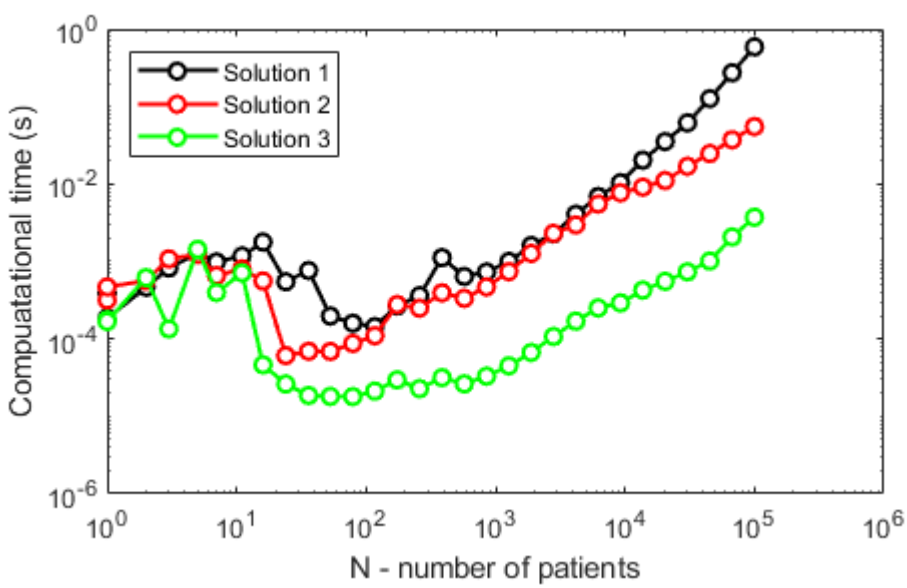
```
patient_DrugA(:,2:3) = [];


%Step 2: Create matrix for patients recieving drub B

%Create a column vector consisting of only patients that will need drub B
patient_DrugB = patient_rawData(patient_rawData(:,2) - patient_rawData(:,3) <=
10,1);

%use logival array indexing to deremine the dose of drugs each person
%should receive. Create a new column containing the dose 20 -- all
patient_DrubB(:,2) = 20;

disp(patient_DrugA)
```

```
     3    30
     5    30
     6    30
    10    30
```

```
disp(patient_DrugB)
```

```
     1
     2
     4
     7
     8
     9
```

For fun, we can **compare the computational time for each of these solutions** using the approach we have discussed previous. To conserve space, only the output is shown here.



Takeaways

- **Clear comments and pseudocode** go a long way towards helping you understand a problem and develop a clear, robust solution.
- Avoid nested if statements, if possible.
- Understanding a decision tree is critical for developing an if statement, but also for producing a solution using logical arrays.
- **Pay particular attention to the *form*** of the output.
- **Vectorized solutions (e.g. using logical arrays) are simpler, faster, and easier to read.**

Increasing computation efficiency with arrays (continued from last time):

Always preallocate arrays

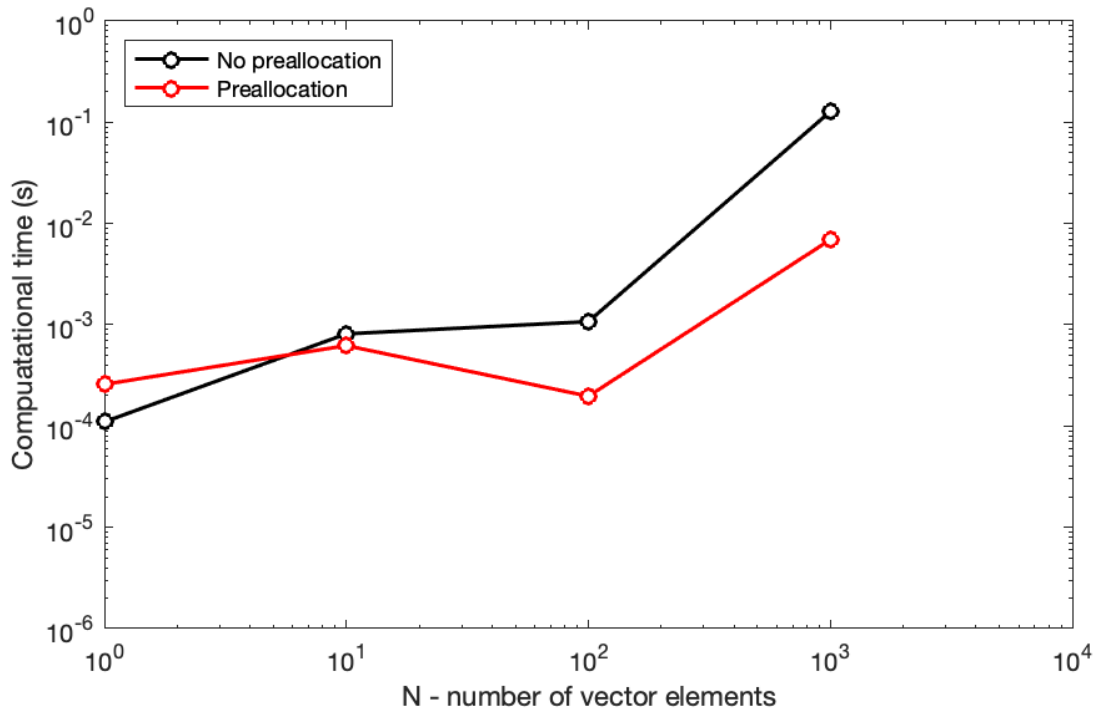
As we saw with vectors, preallocation is the process of creating an array of the final size (usually padded with zeros) before modifying each element of the array. **Consider the computational time associated with creating an NxN multiplication table** using `for` loops for different numbers of N.

That is, what we want to create:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & \cdots \\ 2 & 4 & 6 & 8 & 10 & \cdots \\ 3 & 6 & 9 & 12 & 15 & \cdots \\ 4 & 8 & 12 & 16 & 20 & \cdots \\ 5 & 10 & 15 & 20 & 25 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

```
% Determine range of N
N = logspace(0,3,4);

% Preallocate vectors to store computational time
t_no = zeros(length(N),1);
t_yes = zeros(length(N),1);

% For each number of elements considered
for k = 1:length(N)

    % Calculate time without preallocation
    tic;
    % For each row
    for jj = 1:N(k)
        % For each column
        for mm = 1:N(k)
            % Each entry is the produce of the rows and columns of the
            % multiplication table
            multTable(jj,mm) = jj*mm;
        end
    end
    t_no(k) = toc;
```

```
    % Calculate time without preallocation
    tic;
    % Preallocate array
    multTable = zeros(N(k),N(k));

    % For each row
    for jj = 1:N(k)
        % For each column
        for mm = 1:N(k)
            % Each entry is the produce of the rows and columns of the
            % multiplication table
            multTable(jj,mm) = jj*mm;
        end
    end
    t_yes(k) = toc;

    % Delete multiplication table so that it can be created from scratch on
    % the next loop
    clear multTable
end
% Define dimensions of plot
figure('Position',[0,0,500,300]);
% Plot the functions
plot(N,t_no,'ko-',N,t_yes,'ro-','LineWidth',1.5,'MarkerFaceColor','auto')
% Graphical parameters
set(gca, 'XScale', 'log')   % "log" axes - x
set(gca, 'YScale', 'log')   % "log" axes - y
set(gca,'XMinorTick','on')
xlim([10^0,10^4]);  % range of x values on plot
ylim([10^-6,10^0]); % range of y values on plot
xlabel('N - number of vector elements');
ylabel('Compuatational time (s)');
legend('No preallocation','Preallocation','Location','northwest')
```



<u>Side note</u>: the above multiplication table can be created **much more efficiently** using **matrix multiplication** instead of nested for loops.

```
multTable = (1:10)'*(1:10)
```

```
multTable = 10×10
     1     2     3     4     5     6     7     8     9    10
     2     4     6     8    10    12    14    16    18    20
     3     6     9    12    15    18    21    24    27    30
     4     8    12    16    20    24    28    32    36    40
     5    10    15    20    25    30    35    40    45    50
     6    12    18    24    30    36    42    48    54    60
     7    14    21    28    35    42    49    56    63    70
     8    16    24    32    40    48    56    64    72    80
     9    18    27    36    45    54    63    72    81    90
    10    20    30    40    50    60    70    80    90   100
```

Sparse arrays

Sparse matrices -- large arrays with few nonzero entries -- are encountered in a broad set of engineering disciplines, with applications including solving PDEs, search algorithms, atomistics, and FEA.

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.8 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Consider a sparse matrix of dimensions 1000 x 2000 elements - that correlates to 2 million data points. Assume that this matrix contains non-zero values at the following indices:

- (row 3, column 4) = 15
- (row 100, column 1500) = 5
- (row 1000, column 2000) = 9

This matrix may be established using a conventional matrix indexing:

```
A = zeros(1000,2000);
A(3,4) = 15;
A(100,1500) = 5;
A(1000,2000) = 9;
```

Sparse matrices can be stored efficiently as a matrix of 0's with nonzero values stored by their row, column, and value – this can save significant memory space. In the above example, this matrix could also be established using a sparse matrix syntax:

```
B = sparse([3,100,1000],...
    [4,1500,2000],...
    [15,5,9],1000,2000);
```

Or, the original non-sparse matrix `A` could also be converted to a sparse matrix:

```
B = sparse(A);
```

The sparse matrix offers considerable savings for memory. Information about a variable can be depicted using the `whos` command.

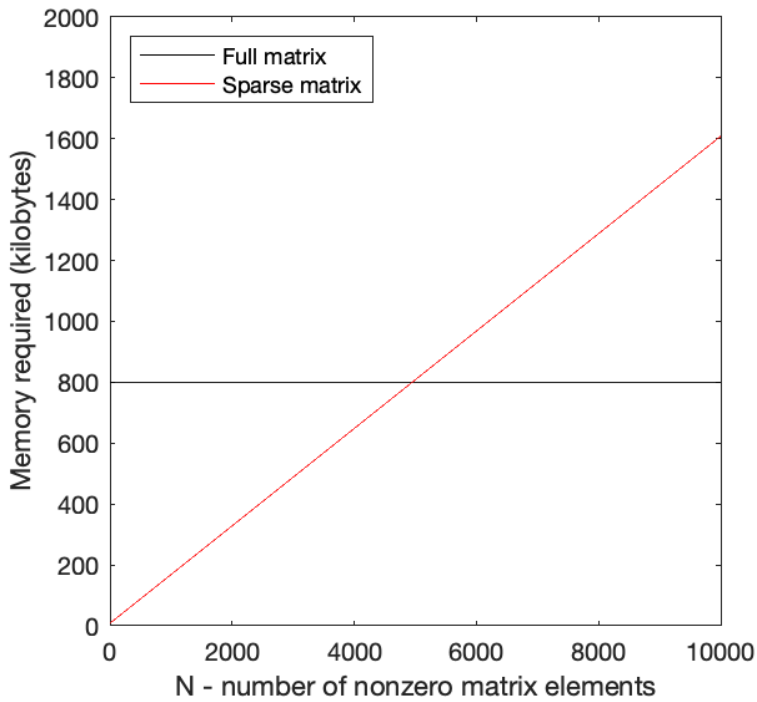
```
whos A B
```

```
  Name         Size                 Bytes  Class     Attributes

  A         1000x2000            16000000  double
  B         1000x2000               16056  double    sparse
```

In this case, variable A requires 16 MB of memory to create the **full matrix**, whereas variable B (**the sparse array**) only requires 16 kB! Dramatic savings in memory can be observed when most of the elements are zero.

At what number of nonzero elements do sparse matrices lose this advantage? Let's plot the memory associated with a matrix variable for different numbers of nonzero elements in a 100x100 matrix.

```
% Define a matrix intially with all zeros
A = zeros(100,100);

% Preallocate vector to store memory
mem_Full = zeros(1,numel(A));
mem_Sparse = zeros(1,numel(A));

% Repeat for all elements
for k = 1:numel(A)
    A(k) = randi(5);

    A_sparse = sparse(A);

    matrixInfo = whos('A','A_sparse');
    mem_Full(k) = matrixInfo(1).bytes;
    mem_Sparse(k) = matrixInfo(2).bytes;
end
% Plot the functions
plot(1:numel(A),mem_Full/100,'k-',1:numel(A),mem_Sparse/100,'r-')

% Graphical parameters
axis square
xlim([0,10^4]);  % range of x values on plot
ylim([0,2*10^3]); % range of y values on plot
xlabel('N - number of nonzero matrix elements');
ylabel('Memory required (kilobytes)');
legend('Full matrix','Sparse matrix','Location','northwest')
```



Since sparse variables need to also store information about the nonzero indices, these variables will require more space when more entries are full. In addtion to memory, a similar dramatic savings in computational time when working with sparse arrays can be observed by using `tic` and `toc`, as we have done previously.

Note: if you need to convert a sparse matrix to a full matrix, this can be accomplished using `full()`.

Revisiting functions

As we have seen so far in this class, functions are **named assemblages of code** that perform specific tasks. By encapsulating sections of code in functions, we can call the function to efficiently repeat the enclosed tasks, while potentially changing any number of inputs.

Functions:

- Take in inputs (from none to many)
- Return outputs (from none to many)
- Are created with a function declaration (at the end of the LiveScript document or in a separate .m file)

```
function myFunction

function out1 = myFunction(in1,in2,in3)

function [out1,out2] = myFunction(in1,in2,in3)

function myFunction(in1,in2,in3)
```

In terms of precedence, commas in a function declaration can be thought of as having the lowest precedence. That is, commas separate inputs, even if the inputs themselves are complicated expressions.

```
A = 1:4
```

```
A = 1×4
     1     2     3     4
```

```
B = A > 2
```

```
B = 1×4 logical array
   0   0   1   1
```

```
% Sum over all rows
sum(((A>2) | A == 1),1)
```

```
ans = 1×4
     1     0     1     1
```

Consider the function `square_add1`, below, which has 2 inputs and 1 output. This function is stored at the end of this LiveScript document.

```
function res1 = square_add1(a,b)
    % Example function. In this case there are two inputs and one
    % output.
    res1 = a^2 + b^2 + 1;
end
```

Functions are properly called by matching the form of the function declaration in your script.

```
val1 = 1;
val2 = 2;

% Call function
x = square_add1(val1,val2)
```

```
x = 6
```

<u>Note</u>: the definition of inputs `a`, `b`, and `c` are unique to the function itself and **can't be seen by the memory workspace** from which the function was invoked.

A common struggle when learning about functions is (unintentionally) **not capturing all of the outputs when invoking the functions**. Consider the function `addMultiply`.

```
function [res1,res2] = addMultiply(a,b,c)
     % Example function
     res1 = a+b+c;
     res2 = a*b*c;
```

```
end
```

The following would **only capture the first output** of `addMultiply` (`res1` in the function declaration) and store it in `ans` (the default variable when no assignment is described) as no output assignment is declared.

```
addMultiply(val1,val2,val3)
```

If only one output is desired, use a **tilde** to **suppress the undesirable output.** The tilde in this case says "this first output is defined/required by the function, but I don't want it…I just want the second output"

```
[~,b] = addMultiply(val1,val2,val3)
```

Function handles

Functions can be passed as variables and referenced/invoked by using this new variable. A variable that has been assigned to a function is referred to as a <u>function handle</u>. Consider our previous function.

```
function res1 = square_add1(a,b)
    % Example function. In this case there are two inputs and one
    % output.
    res1 = a^2 + b^2 + 1;
end
```

To create a function handle, use the `@` symbol. This indicates that the variable is a function handle and not a data type. The following is an example of assigning the function `square_add1` to the function handle a

```
% Create a function handle
f1 = @square_add1;
% Use the function handle. Note the scalar inputs are passed to the
% function in the same way as before.
y = f1(1,2)
```

```
y = 6
```

<u>Important feature</u>: function handles enable us to pass a function as an input into another function.

```
function res2 = operateOnKnownValues(fxn)
    % This function takes an input function and prescribes to set
inputs: 1
    % and 2
    res2 = fxn(1,2);

end
```

```
% Create function handle for new function
f2 = @operateOnKnownValues
```

```
f2 = function_handle with value:
```

```
    @operateOnKnownValues
```

```
% Input square_add1 into new function
z = f2(@sum)
```

```
z = 1
```

You can create function handles for **built-in functions** in addition to custom functions.

```
% Create a vecotr
A = 1:4
```

```
A = 1×4
     1     2     3     4
```

```
% Create function handle for sum() function
f1 = @sum;
f1(A)
```

```
ans = 10
```

Example. Create function handles for the following two functions that are created at the end of this document.

```
function res3 = exampleA(fxn,A)
    % Example function: input 1 is a function, input 2 is an array
    res3 = fxn(A) + 1;

end
```

```
% Function handle for example A
f2 = @exampleA
```

```
f2 = function_handle with value:
    @exampleA
```

```
function res4 = exampleB(fxn,A)
    % Example function: input 1 is a function, input 2 is an array
    res4 = fxn(A) - 1;

end
```

```
% Function handle for example B
f3 = @exampleB
```

```
f3 = function_handle with value:
    @exampleB
```

```
% Call example function, input sum and A. The following two are equivalent
```

```
out1 = f3(@sum,A)
```

```
out1 = 9
```

```
out2 = f2(f1,A)
```

```
out2 = 11
```

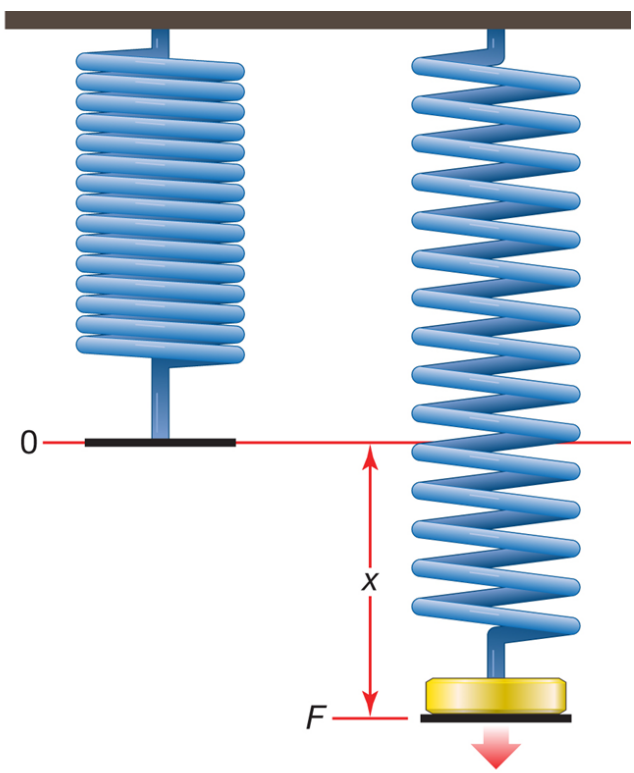
```
% NOTE: an error will occur if you input a function and not a function
% handle
% out3 = f2(sum,A)
```

Example: force on a spring

In your physics, chemistry, and engineering courses, you will likely encounter Hooke's law, which relates the displacement of the end of a spring (i.e. the stretching or compression of the spring) to the force required to cause that deformation:

$$F = k\,x$$

This equation describes a linear dependence of the force on the material deformation. An elastic body or material for which this equation can be assumed is said to be linear-elastic or **Hookean**.



Consider the function `springForce1`, which computes the force on a Hookean spring based on its displacement

```
function force = springForce1(delta)
    % Calculate the force required to cause a deformation of delta
```

```
    % Spring constant = 0.5
    force = 0.5*delta;
end
```

The function can be assigned to a function handle and invoked with an input for `delta` from the command line, a script, or another function

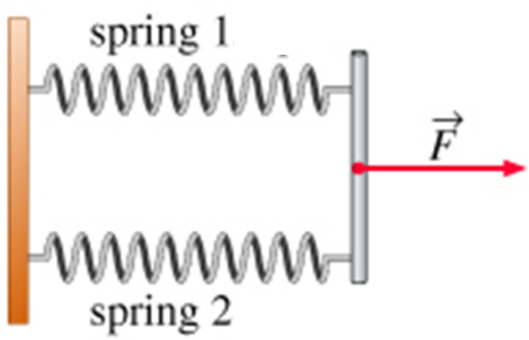
```
% Create function handle
a = @springForce1;
% Call function
myForce = a(4)
```

```
myForce = 2
```

However, **nonlinear springs are also common**, as even the most linear spring can be deformed enough such that a nonlinear relationship between force and deformation becomes apparent. Consider force-displacement relationship for another spring.

```
function force = springForce2(delta)
    % Calculate the force required to cause a deformation of delta
    % in a nonlinear spring
    force = .2*delta^2 + .5*delta;
end
```

Using function handles, we could easily create a modular representation of two springs in parallel.



In this case, the net force is determined as the sum of the contributions from each spring, which could be encoded into a separate function.

```
function force = parallelSprings(spring1,spring2,delta)
    % Force for two springs loaded in parallel config.
    % spring1 = handle to force/deflection eqn for spring 1
    % spring2 = handle to force/deflection eqn for spring 2
    % delta = displacement of springs

    force = spring1(delta) + spring2(delta);
end
```

```
% Evaluate the total force when spring 1 and 2 are arranged in parallel
```

```
force = parallelSprings(@springForce1,@springForce1,5)
```

```
force = 5
```

Note: in this way, we use functions to create a framework in which we can easily change different components.

The following functions are used to introduce function handles.

TOPIC 12: Function handles and recursion

ENGR105, 10/13/20

Reminder: function handles

Functions can be passed as variables and referenced/invoked by using this new variable. A variable that has been assigned to a function is referred to as a function handle. Function handles may be created with **user-defined** or **built-in** functions.

```
% Create function handle with product function
a = @prod;
a([3,2,1,2])
```

```
ans = 12
```

```
b = @mod;
% Create function handle with modulus function
b(3,2)
```

```
ans = 1
```

Function handles are **useful** because these enable us to **pass one function into another function**. Consider the following function.

```
function fxn1(f,x)
    % Example function
    output = f(x);
```

```
    disp(strcat('The output of this function is ',num2str(output)));
end
```

```
% Create a function handle
f1 = @sum;
% Define a vector
c = 1:4
```

```
c = 1×4
     1     2     3     4
```

```
% Call function that takes the handle and vector as inputs
fxn1(f1,c)
```

```
The output of this function is...10
```

Where else have we seen function handles? **We saw them when solving systems of nonlinear equations (<u>Topic 3 / Lecture 4</u>).**

```
% Determine solution 1
sol_1 = fsolve(@eqnSet,[0,0])
```

Here, `eqnSet` contained a set of nonlinear equations to be solved:

```
function F = eqnSet(x)
    % System of two non linear equations
    % F(1) and F(2): "equation" 1 and 2, respectively
    % x(1) and x(2): variable 1 and 2, respectively
    F(1) = x(2)-2*x(1)^2+3;
    F(2) = x(2)^2-x(1);
end
```

How does `fsolve` take in a function handle to solve the set of equations? Let's consider a more familiar example.

Practice problem: a root-finding numerical solver

Create a function that uses the Newton-Raphson method to find the root of any arbitrary (well-behaved) input function. **Apply this function** to find the root of the following equation, where $g(t)$ could represent a *temporal concentration profile*, signal, or other *time-dependent function*.

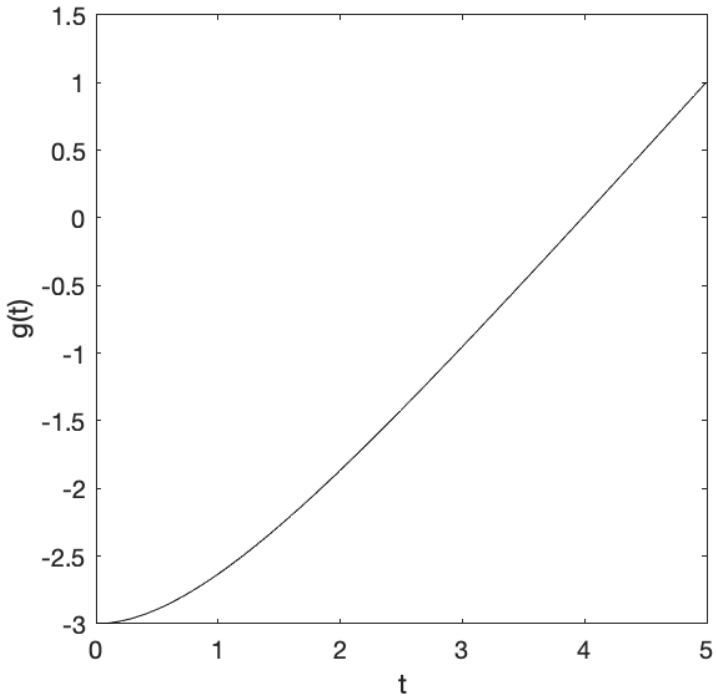
$$g(t) = t + e^{-t} - 4$$

<u>Checking in</u>: what does it mean to find the *root of an equation*? It means solving for when that function is **equal to zero**.

% Time vector

```
t = 0:0.1:5;       % Time vector
g = t + exp(-t)-4; % Function g(t)
plot(t,g,'k-')
```

```
axis square
xlabel('t')
ylabel('g(t)')
```



Start by remembering our systematic approach.

PRO:

1. **P**lan, **P**repare, and **P**seudocode
2. w**R**ite **R**ough (increments of code) and **R**efine
3. **O**ptimize and improve

Step 1: Plan, prepare, pseudocode

Recall from earlier, the Newton-Raphson method is an iterative algorithm for finding a root (based on an initial guess) of any well-behaved function. The Newton-Raphson equation is given by the following:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

In **Topic 7 / Lecture 8**, we found the root of the following $f(x)$ by creating a script that applies the Newton-Raphson method.

$$f(x) = x^4 + 2(x-2)^2 - 20$$
$$f'(x) = 4x^3 + 4(x-2)$$

```
% Script for using the Newton-Raphson method to solve for f(x)

% Step 1: specify initial guess
xn = 5; % initial guess
xprev = inf; % Variable to store previous xn value -> ensure it is
           % not xn to enter while loop

% Step 2: iterate until xn -> xn+1
while abs(xn - xprev) > 10^-5

    % Step 3: evaluate function at xn
    fn = xn^4 + 2*(xn-2)^2 - 20;
    % Step 3a: evaluate derivative at xn
    f_prime_n = 4*xn^3 + 4*(xn-2);

    % Instead of calculating the "next" x value, store the previous xn
    % value
    xprev = xn;

    % Step 3b: update xn to next value for the iteration
    xn = xn - fn/f_prime_n;
end
disp(xn)
```

```
2.1141
```

Step 2: w**R**ite **R**ough (increments of code) and **R**efine (to code a new application)

```
%Determine root of original f(x)
rootF = newtonSolve(@originalEqn, 5, 10^-5)
```

```
rootF = 2.1141
```

Apply this function to the problem here:

```
rootG = newtonSolve(@originalEqn, 5, 10^5)
```

```
rootG = 3.7832
```

```
newEqn(rootG)
```

```
ans = 1×2
   -0.1940    0.9773
```

We have thus created our first function that works as a numerical solver!

Anonymous functions

Function handles can also be used to store **anonymous functions**. Anonymous functions are **not "only" the built-in function** or **a custom function stored in a `.m` file**, but are associated with a variable. These are useful for smaller/repeatable computations that don't demand a full function and accessibility from multiple functions. Anonymous functions may take in one to multiple inputs, can only return one output, and begin with a list of input variables `@(var1,var2,...,varN)`.

- Example operating on a **scalar**:

```
% Anonymous function of a single input operating on a
% scalar.
myFunc = @(x) x^2 + 2*x + 1;
res1 = myFunc(1)
```

```
res1 = 4
```

- Example on a **vector**:

```
% Anonymous function of a single input operating on a
% vector.
myFunc = @(x) x.^2 + 2.*x + 1;
res2 = myFunc([1:4])
```

```
res2 = 1×4
     4     9    16    25
```

- Example with **multiple inputs**:

```
% Anonymous function of a multiple inputs operating on
% scalars.
%x and y are the inputs
myFunc = @(x,y) x^2 + 2*y +1;
res3 = myFunc(1,3)
```

```
res3 = 8
```

```
 % Anonymous function of a multiple inputs operating on
% vectors.
myFunc = @(x,y) x.^2 + 2.*y +1;
res4 = myFunc([1,2,2,1],4:7);
```

- Example using **built-in function**

```
mod2 = @(x) mod(x,2);
```

```
mod2(3)
```

```
ans = 1
```

Note: this is different from a "simple" function handle based only on a built-in function. Here, we define a function handle that only takes *one* input. Previously, the function handle must take *two* inputs.

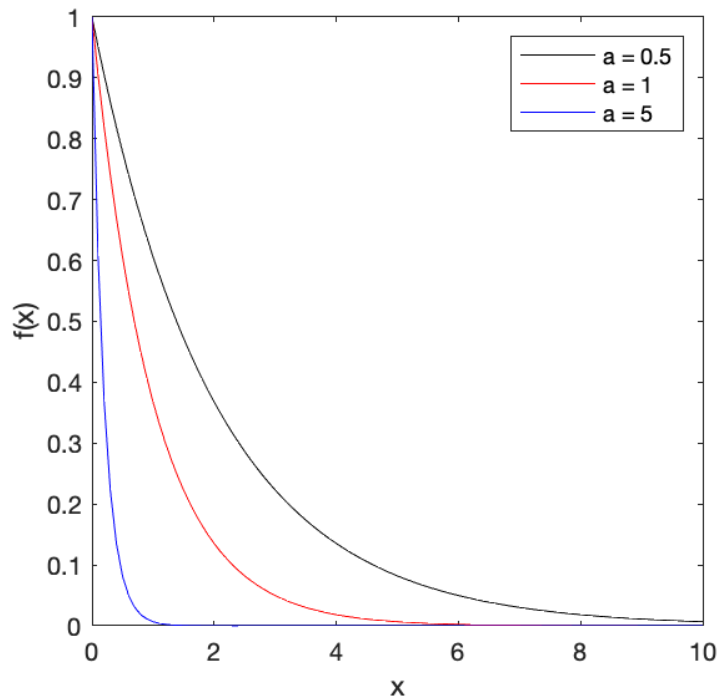
Example. Plot the following equation, $f(x)$, over the domain $x \in [0, 10]$ for $a = 0.5,\ 1,\ \text{and}\ 5$.

$$f(x) = e^{-a\,x}$$

```
% Plot x vector
x = 0:0.1:10;
% Define function handle
f = @(a) exp(-a*x)
```

```
f = function_handle with value:
    @(a)exp(-a*x)
```

```
% Plot function for each a
plot(x,f(0.5),'k-')
hold on     % plot on the same set of axis
plot(x,f(1),'r-')
hold on     % plot on the same set of axis
plot(x,f(5),'b-')
hold off    % turn "off" plotting on the same set of axis

axis square
legend('a = 0.5','a = 1', 'a = 5')
xlabel('x')
ylabel('f(x)')
```



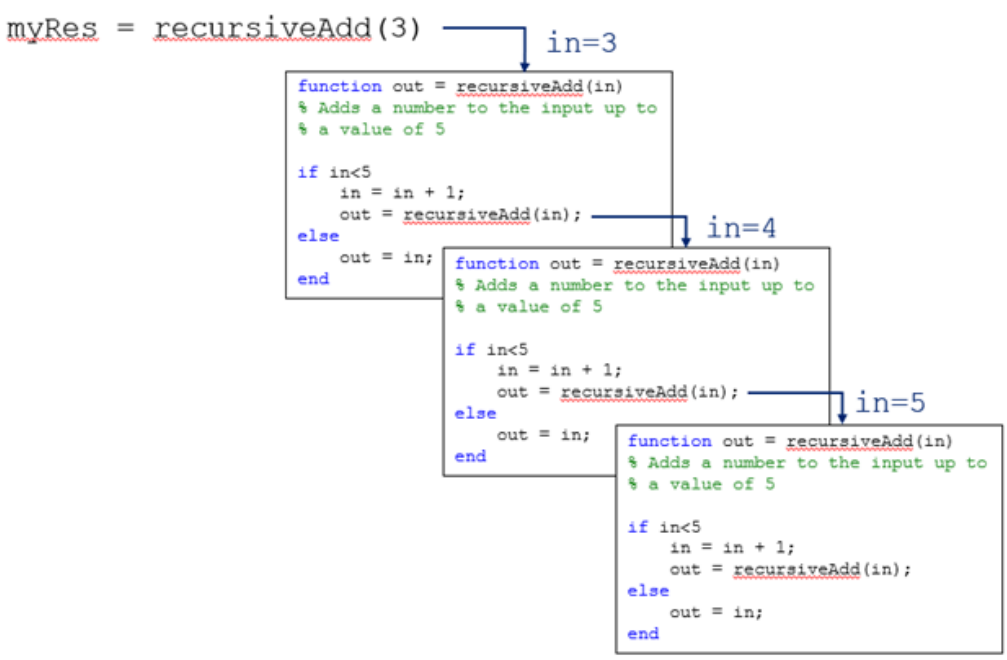
<u>Note</u>: anonymous functions (and function handles) can help reduce the frequency of recurring errors.

Recursion

To *repeat tasks* in a computational environment, one could either use loops (as we have seen before) or call a **recursive function**. Recursive functions call themselves. When a function calls itself, it "opens" a new implementation of the function that is distinct from the original invocation. Consider the function `recursiveAdd`, defined at the end of this LiveScript document.

```
function out = recursiveAdd(in)
    % Adds a number to the input up to
    % a value of 5

    if in<5
        in = in + 1;
        out = recursiveAdd(in);    % Recursive call
    else
        out = in;     % the base conditionstops the recursion
    end
end
```

The following schematic indicates what will happen when this is called:



Once the logical statement is not satistifed, each invocation of the function will output the same entity.

```
% Call the recursive function
x = recursiveAdd(3)
```

```
x = 5
```

Example: Revisiting the fibonacci sequence

Earlier in this course, we created code to calculate either the first n entries of the Fibonacci sequence or all elements less than some integer. For example, consider the latter, in which we can use a `while` loop to calculate all elements of the Fibonacci sequence less than 50.

```
% Max value of entries in Fibonacci sequence
n = 50;
% Define first two entries
x(1) = 0;
x(2) = 1;
% Iterate as long as the last element is less than 100.
while x(end) < n
    % If the last element is less that 100, add the next element of the
    % Fibonacci sequence to the vector
     x(end+1)=x(end)+x(end-1);
end
x(end)=[];  % Remove last element
disp(x)
```

```
     0     1     1     2     3     5     8    13    21    34
```

The same process can be accomplished using **recursion**. Consider the following function.

```
function x = recursiveFib(x,maxVal)
    % Function using recursion to create a Fibonacci sequence vector

    % If the last element is less that maxVal, add the next element of
the
    % Fibonacci sequence to the vector
    if  x(end-1) + x(end) < maxVal
        % Calculate next entry
        x(end+1) = x(end)+x(end-1);
        % Use recursion to repeat the process
        x = recursiveFib(x,maxVal);
    end
end
```

```
% Calculate Fibonacci sequence vector containing elements less than 50
x = recursiveFib(0:1,50)
```

```
x = 1×10
     0     1     1     2     3     5     8    13    21    34
```

Recursion can be useful:

- **Not all languages support loops** (e.g. Haskell)
- Loops may, in some cases, require more complex code/indexing that leads to a *higher probability of errors.*

Recursion can have **limitations**:

- First, recursive solutions may involve extensive overhead because they use function calls.
- Each time you make a call you use up some of your memory allocation. If the recursion is deep – that is, if there is a large number of recursive calls – then you may run out of memory.

Example: calculate the factorial of a number

Consider calculating the factorial of a number:

$$fact(n) = \begin{cases} 1 & n = 0 \\ n \cdot fact(n-1) & n > 0 \end{cases}$$

We can solve this problem using either recursion or loops in MATLAB. Consider the following recursive function.

```
function y = fact(n)
    % Recursive function for determining the factorial

    % Specify the number we are interested in solving the factorial for
    y = n;

    % Specify the "lower limit" in the factorial calculation
```

```
    if n == 0
        y = 1;
    else
        % We multiply by all the integers before the input,
        % one at a time...
        y = y*fact(n-1);
    end
end
```

The factorial of a number may be calculated by calling this **recursive function**. However, "out-of-memory" errors will arise for **large input numbers**.

```
n = 50000;
% Call recursive function
x = fact(n);
```

This issue is avoided by using a simple `for` loop.

```
fact_n = 1;
for m = 1:n
    fact_n = fact_n*m;
end
```

Takeaway: recursion is another tool that, in the right context, will yield simple, easy-to-troubleshoot code.

TOPIC 13: Plotting in MATLAB

ENGR105, 10/15/20

An introductory note about data

As scientists and engineers, you will work a lot with data: acquiring it, processing it, visualizing it. An advantage of MATLAB is that many data visualization capabilities are built into MATLAB. We will start by importing sensor data.

```
% Load the data we just collected
rawData = load('sensorData5.mat')
```

```
rawData = struct with fields:
    Position: [42×6 timetable]
```

The imported data is an *object*, a data structure. The table that we want can be specified by selecting the `Position` field.

```
% Timetable containing our collected sensor data
rawData = rawData.Position
```

```
rawData = 42×6 timetable
           Timestamp            latitude    longitude    altitude    speed    course
hacc
    ________________________    ________    _________    ________    _____    ______
______

    15-Oct-2020 09:03:04.624     39.952      -75.191      13.347     0.462    187.39
35.374
    15-Oct-2020 09:03:05.624     39.952      -75.191       13.07      0.29     203.6
26.742
    15-Oct-2020 09:03:06.624     39.952      -75.191       13.07      0.29     203.6
26.742
    15-Oct-2020 09:03:07.624     39.952      -75.191      13.045     0.245    203.49
21.802
    15-Oct-2020 09:03:08.624     39.952      -75.191      13.053     0.277    209.58
25.921
    15-Oct-2020 09:03:09.624     39.952      -75.191      13.035     0.275    211.23
43.667
    15-Oct-2020 09:03:10.625     39.952      -75.191      13.038     0.275    208.77
61.217
    15-Oct-2020 09:03:11.625     39.952      -75.191      13.033     0.261    212.36
26.538
    15-Oct-2020 09:03:12.625     39.952      -75.191      13.039     0.257    212.19
27.775
    15-Oct-2020 09:03:13.625     39.952      -75.191      13.041     0.248    212.04
29.138
    15-Oct-2020 09:03:14.624     39.952      -75.191      13.042     0.242    211.56
30.59
    15-Oct-2020 09:03:15.760     39.952      -75.191      13.041     0.237    209.36
32.166
    15-Oct-2020 09:03:31.167     39.952       -75.19      13.242         0         0
65
    15-Oct-2020 09:03:37.127     39.952       -75.19      13.441         0         0
65
    15-Oct-2020 09:03:50.359     39.952       -75.19      13.441         0         0
73.948
    15-Oct-2020 09:03:50.364     39.952       -75.19      13.441         0         0
73.948
```

Note: a field is essentially a parameter for the object. This parameter can contain some setting or it can contain useful data.

When working with data, it is **extemely important** to understand what ***class*** (or data type) you are working with.

- `myData` is `timetable` class
- `myData.Timestamp` is `datetime` class
- `myData.speed` is `double` class
- `'hello world'` is `char` class

Important: when working with data, part of **STEP 1** (**P**lanning, **p**rep, and **p**seudocode) is always getting your data into the correct format. For now, we will simply convert the above data table into a matrix containing all numbers (of class `double`).

Note on coordinates systems: one can convert latitude and longitude differences to distance using knowledge of coordinate systems. For the purposes here, we will use a "rule of thumb" from the USGS: At 38 degrees North latitude:

- One degree of **latitude** equals approximately 364,000 feet (69 miles), one minute equals 6,068 feet (1.15 miles), and one-second equals 101 feet.
- One-degree of **longitude** equals 288,200 feet (54.6 miles), one minute equals 4,800 feet (0.91 mile), and one second equals 80 feet.

```
% Step 1: convert the Timestamp information into "seconds past start time"
% Convert date time vector into numbers representing the corresponding minutes
t = minute(rawData.Timestamp) + second(rawData.Timestamp)/60;
t = (t - t(1))*60;      % t is now a vector in which entries are in seconds

% Step 2: Convert positional information to relative positions
% Convert both to relative positions
lat_1 = (rawData.latitude - rawData.latitude(1));
long_1 = (rawData.longitude - rawData.longitude(1));

% Both are in units of "degrees" -> convert to units of meters
lat_2 = lat_1*364000*0.3048;
long_2 = long_1*288200*0.3048;

% Convert altitude to relative z position, in meters
alt = rawData.altitude - rawData.altitude(1);

% Step 2: Create data matrix
myData = [t,long_2,lat_2,alt,rawData.speed];
disp(myData)
```

```
         0         0         0         0    0.4620
    1.0000   -0.8784   -1.1095   -0.2770    0.2900
    2.0000   -0.8784   -1.1095   -0.2770    0.2900
    3.0000   -0.8784   -2.2189   -0.3020    0.2450
    4.0000   -0.8784   -3.3284   -0.2940    0.2770
    5.0000   -0.8784   -3.3284   -0.3120    0.2750
    6.0010   -0.8784   -3.3284   -0.3090    0.2750
    7.0010   -1.7569   -4.4379   -0.3140    0.2610
    8.0010   -1.7569   -4.4379   -0.3080    0.2570
    9.0010   -1.7569   -4.4379   -0.3060    0.2480
   10.0000   -1.7569   -4.4379   -0.3050    0.2420
   11.1360   -1.7569   -4.4379   -0.3060    0.2370
   26.5430   41.2864  -36.6126   -0.1050         0
   32.5030   44.8001  -38.8315    0.0940         0
   45.7350   64.1257  -53.2547    0.0940         0
   45.7400   64.1257  -53.2547    0.0940         0
   51.9090   39.5295  -44.3789    9.7340         0
   55.3610   35.1373  -44.3789   10.1320         0
   55.7870   45.6785  -52.1452    9.8350         0
   58.1060   47.4354  -57.6925    9.7520         0
   58.5210   47.4354  -58.8020    9.6310         0
   64.7160   54.4629  -66.5683    9.2280         0
```

```
   70.0000   32.5020  -52.1452    2.8660    0.2960
   71.0000   32.5020  -52.1452    2.8660    0.2960
   72.0000   30.7452  -55.4736   -1.4850    0.3580
   73.0000   27.2314  -54.3641   -1.5480    0.3580
   74.0000   27.2314  -56.5831   -1.5190    0.4710
   74.9990   26.3530  -59.9115   -1.5550    0.6290
   75.9990   26.3530  -62.1304   -1.5930    0.6060
   76.9990   25.4746  -63.2399   -1.6090    0.6060
   77.9990   23.7177  -65.4588   -1.6680    0.1980
   78.9990   23.7177  -65.4588   -1.6410    0.1990
   79.9990   25.4746  -65.4588   -1.6610    0.1510
   80.9990   22.8393  -67.6778   -1.7470    0.3580
   81.9990   23.7177  -71.0062   -1.7350    0.3990
   82.9990   24.5961  -71.0062   -1.7520    0.3720
   83.9990   24.5961  -72.1157   -1.7830    0.1140
   84.9990   22.8393  -72.1157   -1.8370    0.2040
   85.9990   21.9608  -73.2252   -1.8570    0.2630
   86.9990   21.0824  -73.2252   -1.8410    0.2690
   87.9990   20.2040  -73.2252   -1.8140    0.1450
   88.9990   20.2040  -74.3346   -1.7820    0.2290
```

Note: always include in your script (or function) explicit information about what each row or column of a matrix corresponds to. I do this by commenting out portions of code

```
% myData(:,1): time, seconds
% myData(:,2): longitude (x value), meters relative to start.
% Positive x: move east; negative x: move west
% myData(:,3): latitude (y value), meters relative to start
% Positive y: move north; negative y: move south
% myData(:,4): altitude (z value), meters relative to start
% myData(:,5): speed, meters/second
```

Plotting in 2D

Helpful resources:

- **List of plotting functions in MATLAB**

https://www.mathworks.com/help/matlab/creating_plots/types-of-matlab-plots.html

- **Axis settings**

https://www.mathworks.com/help/matlab/ref/axis.html

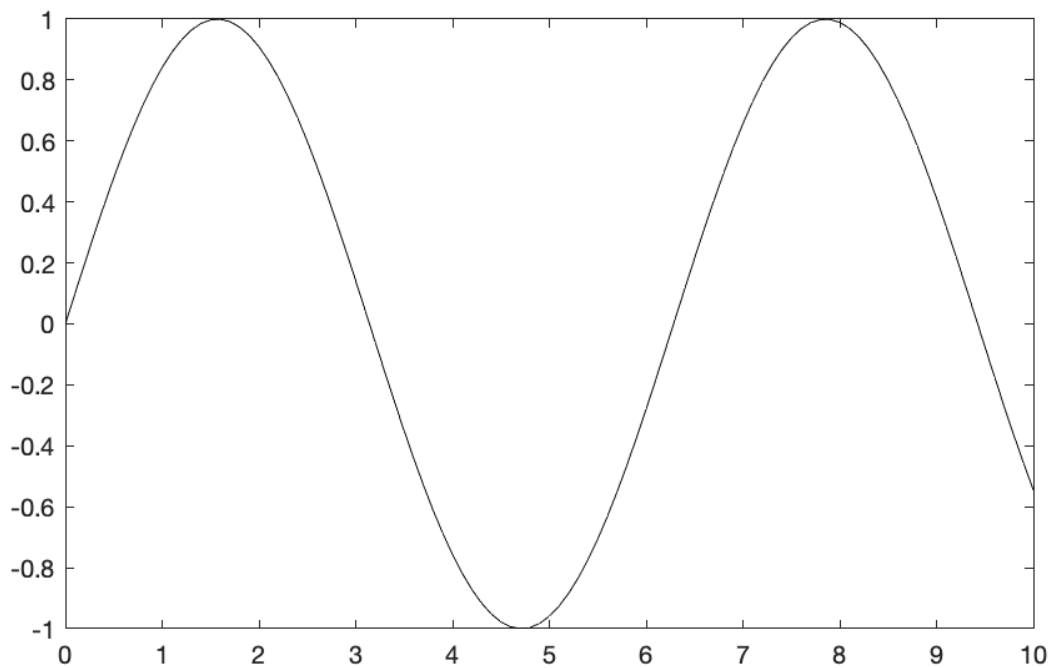
- The `help` documentation for all of these functions.

`plot()` - the "workhorse" plotting function

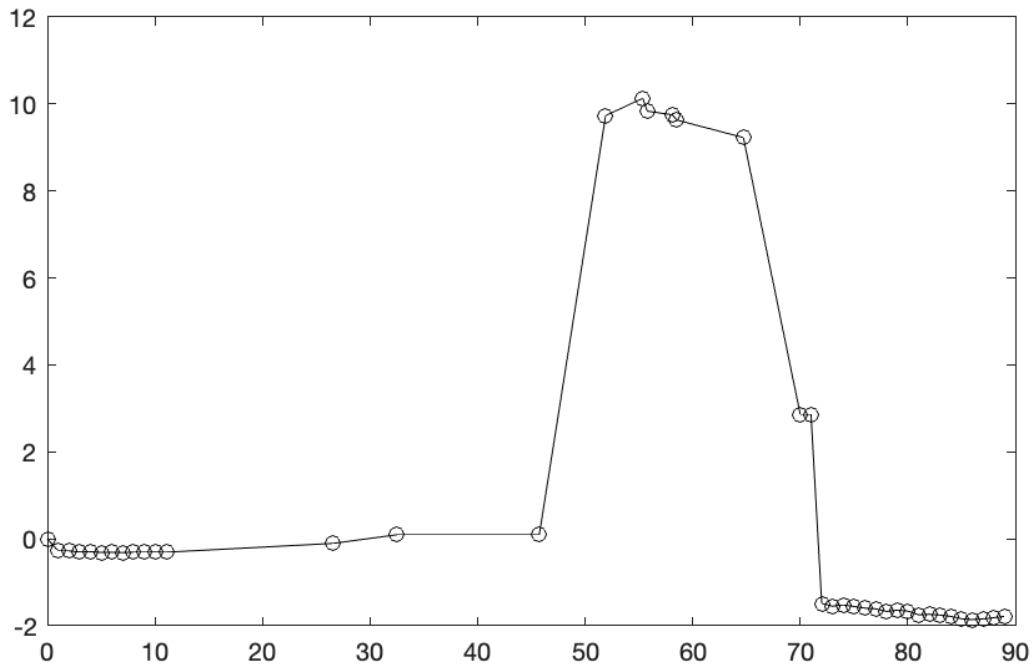
- `plot()` requires **input vectors of equal length**

```
% Independent variable
t = 0:0.1:10;
```

```
% Function (dependent variable) is sin(t)
f = sin(t);
% Plot the function: adjust the "line specification" to change the output
plot(t,f,'k-')
```

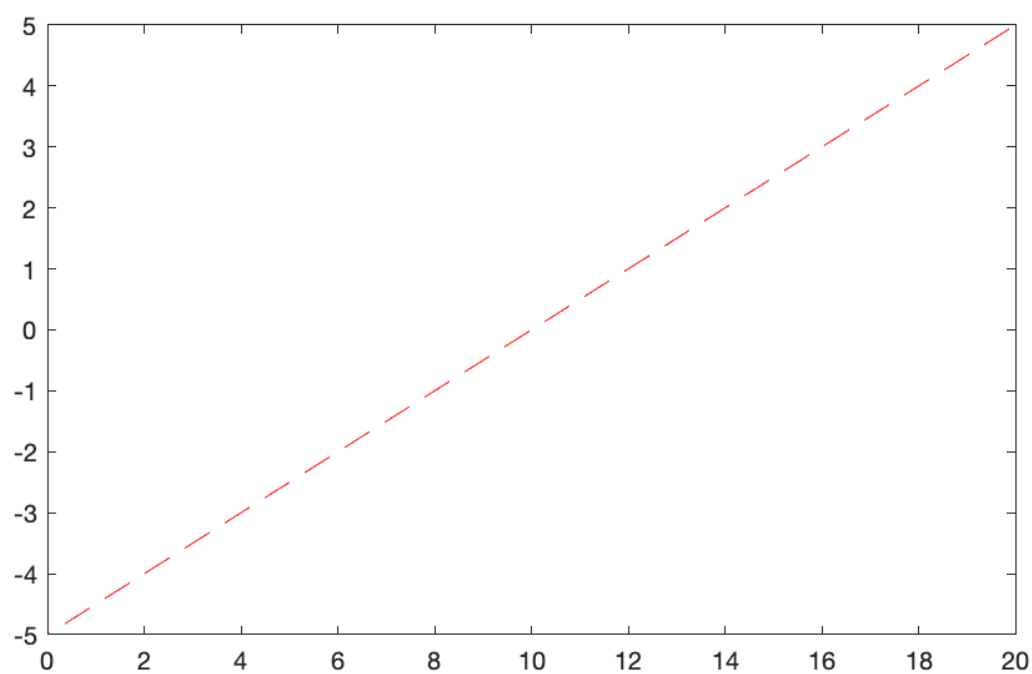



```
% time
t1 = myData(:,1);
% z position
y1 = myData(:,4);
% Plot the function: adjust the "line specification" to change the output
plot(t1,y1,'ko-');
```



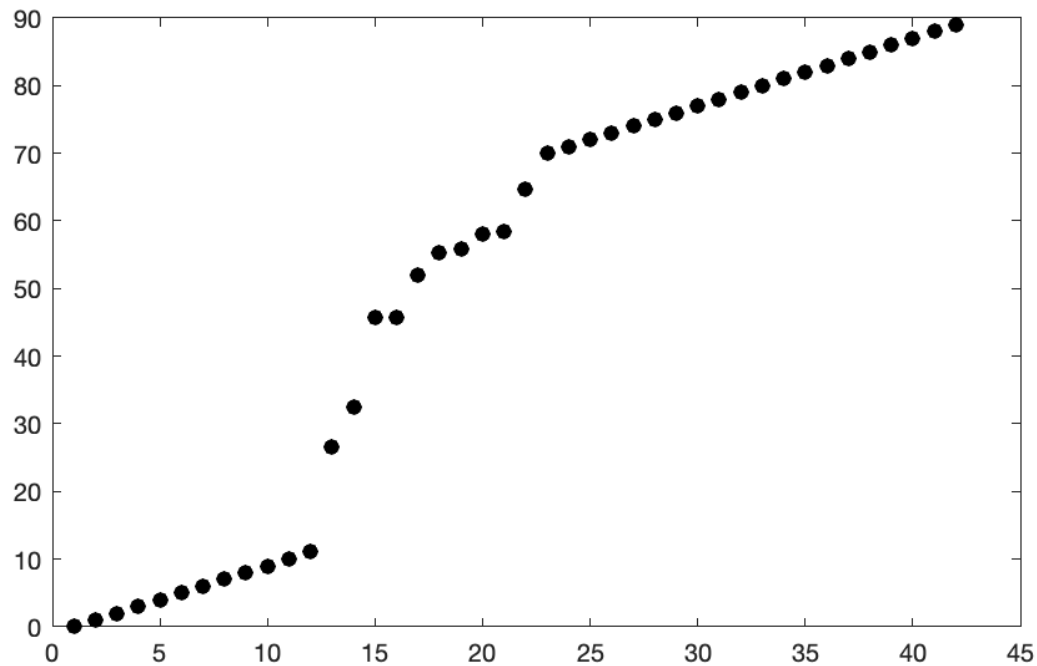
Plotting straight lines. Straight lines can be plotted by specifying as little as two points - the beginning and end of the line

```
plot([0 20],[-5 5],'r--')
```



Plotting one vector. Most 2D plot commands can be called with a single vector, which is often useful for debugging.

```
plot(myData(:,1),'ko','MarkerFaceColor','k')
```

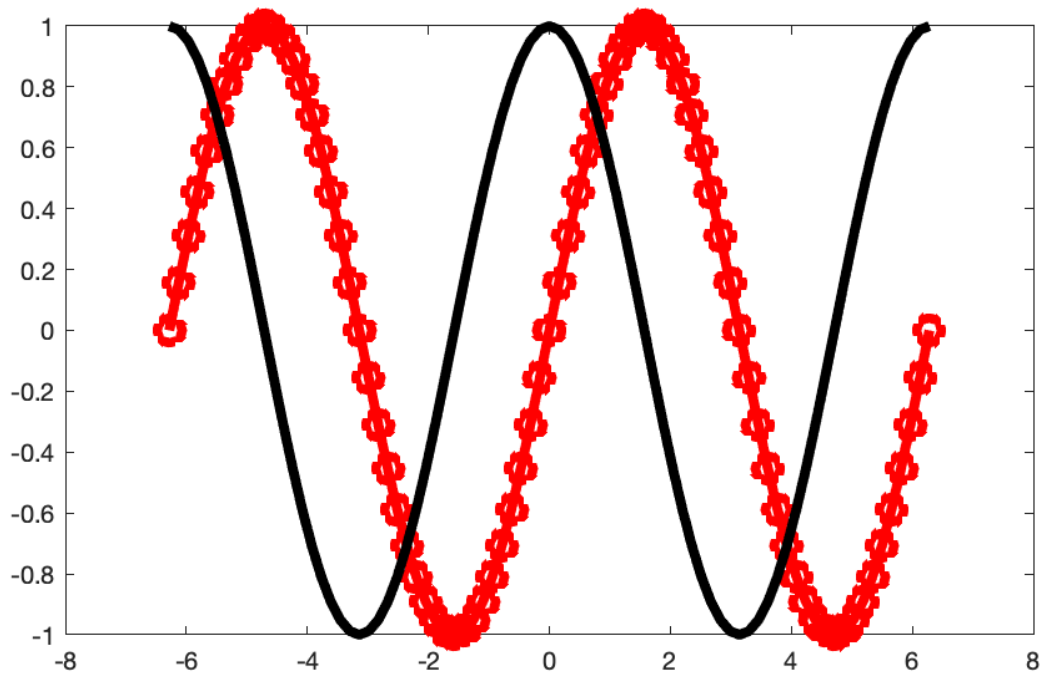


In this case, the x values indicate the **index number**.

Overlaying plots

Using one `plot` function call. Multiple plots may be **overlaid** by **specifying multiple vectors** for most built-in plotting functions.

```
x = -2*pi:pi/20:2*pi;
y1 = sin(x);
y2 = cos(x);
plot(x,y1,'ro-',x,y2,'k-','LineWidth',4,'MarkerSize',10)
```

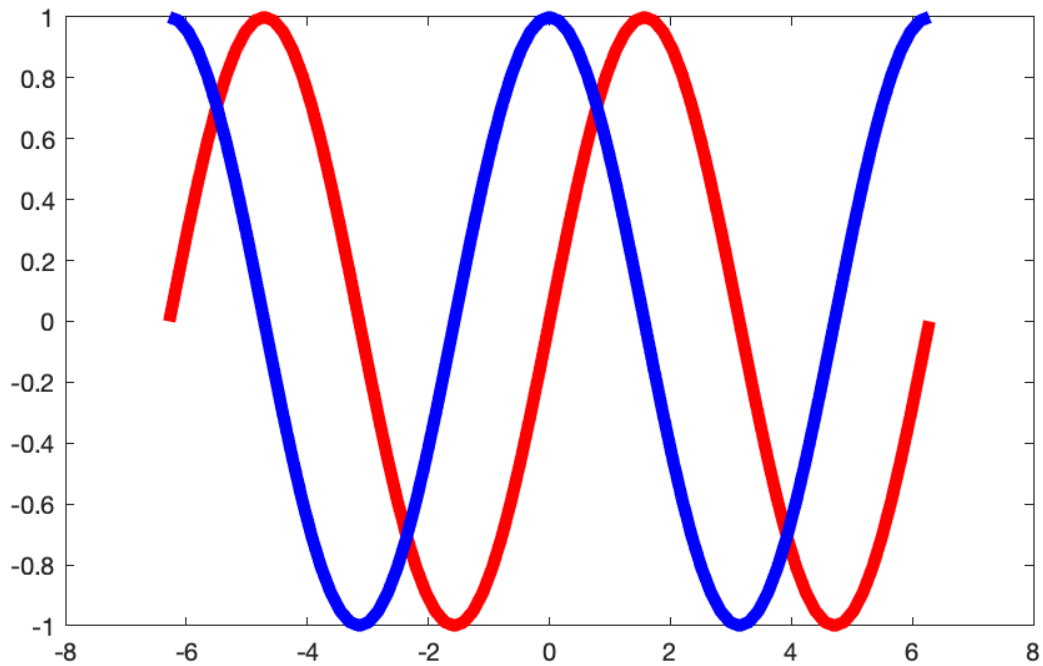


Here, the "x" input must be specified for each plot instance. In the above, MATLAB automatically chooses the plotting colors unless otherwise specified. Plot colors may be specified for each vector.

Note: line specifications such as `LineWidth` invoked within the plot function will be applied to **all** data series when using this method.

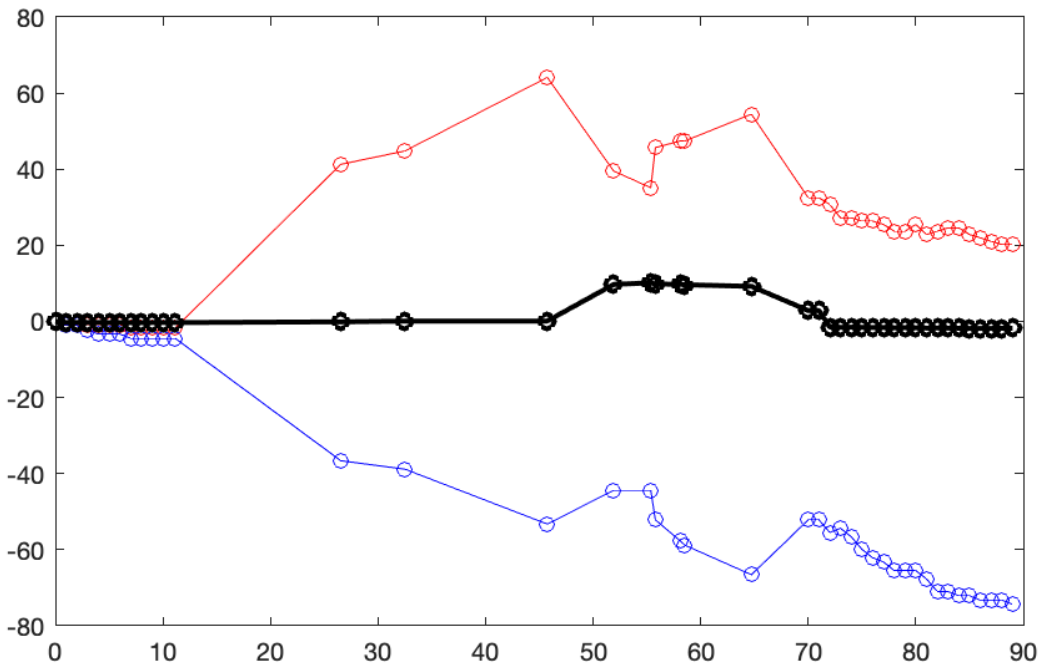
Using `hold on/off`. Multiple plots may also be overlaid by using the `hold on` and `hold off` commands

```
x = -2*pi:pi/20:2*pi;
y1 = sin(x);
y2 = cos(x);
plot(x,y1,'r-','LineWidth', 5)
hold on
plot(x,y2,'b-','LineWidth', 5)
hold off
```



`hold on` is invoked after the first plot command and indicates that more data will be attributed to the existing axes; `hold off` "closes out" `hold on`.

```
% x position vs. time
plot(myData(:,1), myData(:,2),'ro-');
hold on
% y position vs. time
plot(myData(:,1), myData(:,3),'bo-');
hold on
% z position vs. time
plot(myData(:,1),myData(:,4),'ko-','LineWidth',2)
hold off
```



<u>IMPORTANT NOTE</u>: if you are **not** using a LiveScript file, any additional plots will completely overwrite the axes if you do not use `hold on/off`. hold on is only needed once, though I often call it after every plot for clarity.

Specifying plot attributes inline

The canonical plot call is:

```
plot(x,y,'LineSpecifiers', ...
    'PropertyName',PropertyValue)
```

Line specifiers

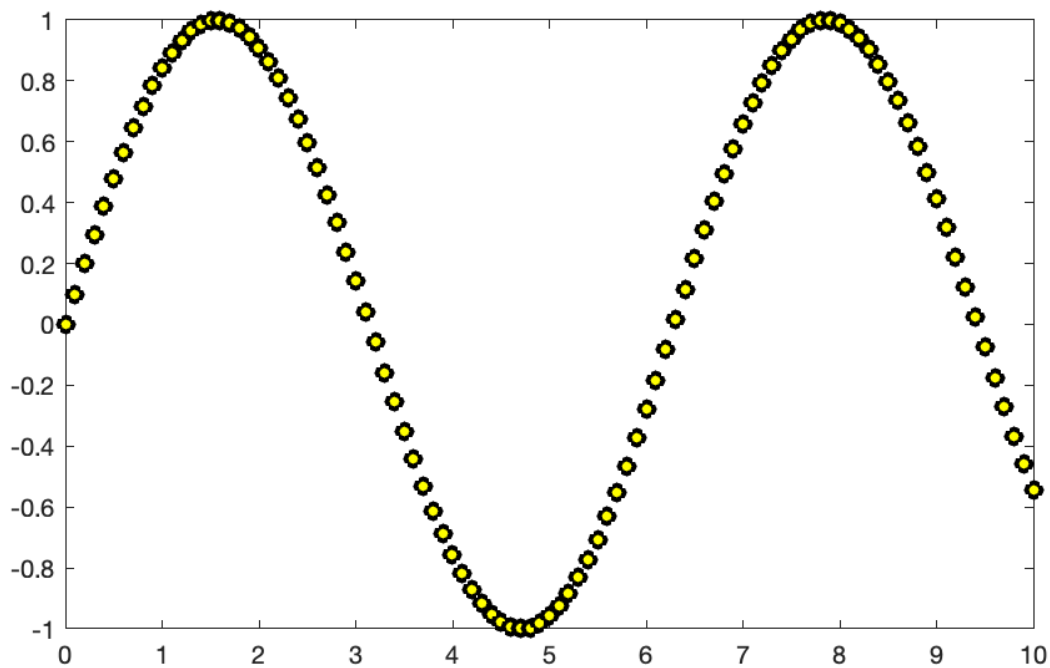
- Line style: solid, dash, dot, dash-dot
- Line color: red, green, blue, cyan, etc.
- Marker: plus, circle, asterisk, etc.

Property names

- LineWidth
- MarkerSize
- MarkerEdgeColor
- MarkerFaceColor

```
% Independent variable
t = 0:.1:10;
% Function (dependent variable) is sin(t)
```

```
f = sin(t);
% Plot the function: adjust the "line specification" to change the output
plot(t,f,'ko','MarkerFaceColor','y','LineWidth',2)
```



Note: the default plot attributes are blue lines, though this can be changed by adjusting the above inputs. Please see the **help** documentation for `plot` and adjust the above plot.

Can you produce the above with:

- Red circles
- Red circles with lines between markers
- Black circles with connected lines
- Green dots
- Magenta circles with dashed lines between markers
- Black plot line set to a 10 pt width
- Circles with a green face color, red edge color, and 2 pt edge width

Colors

MATLAB includes many predefined colors:

- r = Red
- g = Green
- b = Blue
- c = Cyan
- m = Magenta
- y = Yellow

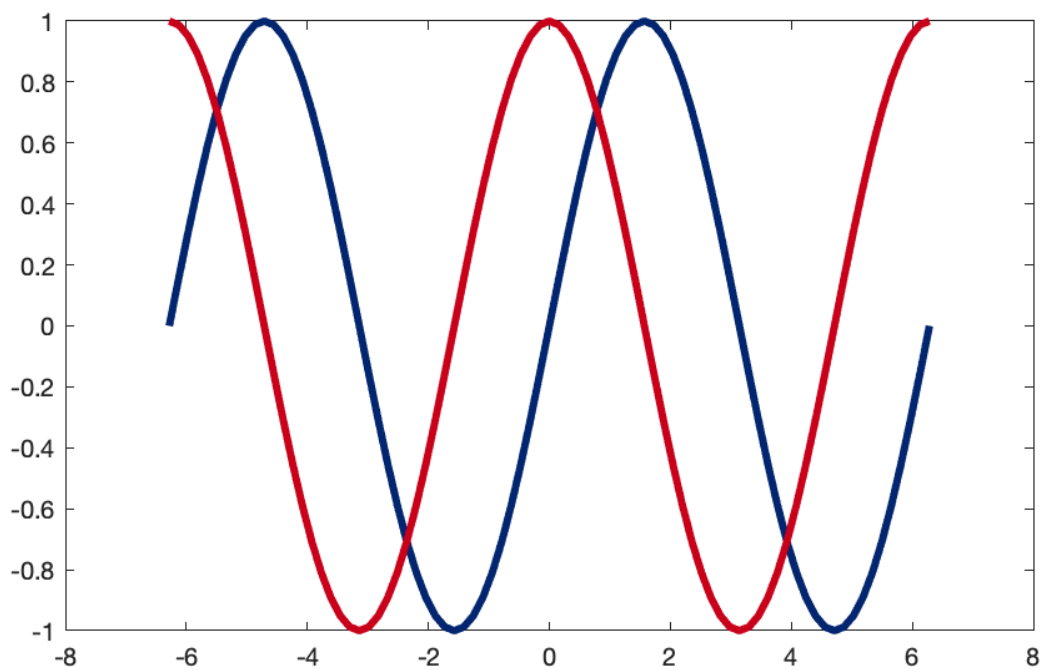
- k = Black
- w = White

Colors may also be specified using the `Color` (or other appropriate) property and specifying the **3-digit RGB color set.**

- RGB colors typically range from 0 to 255 in most computer systems (languages), however, **MATLAB takes in color values rescaled from 0 to 1.**

Example. Consider Penn blue (RGB color code 1, 37, 10) and red (RGB color code 149, 0, 26)

```
% x vector
x = -2*pi:pi/20:2*pi;
% Two different y functions
y1 = sin(x);
y2 = cos(x);
plot(x,y1,'Color',[1,37,110]./255,'LineWidth',3);
hold on
plot(x,y2,'Color',[200,0,26]./255,'LineWidth',3);
hold off
```

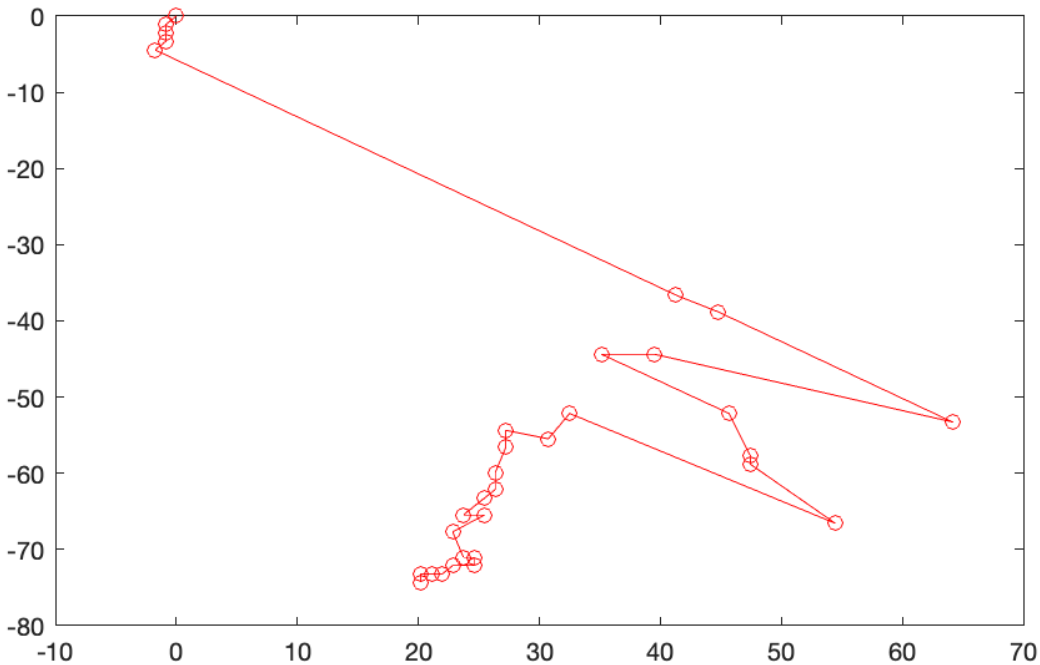


Basic axis formatting

Consider our example data -- we want to view our path, only considering motion in the x/y.

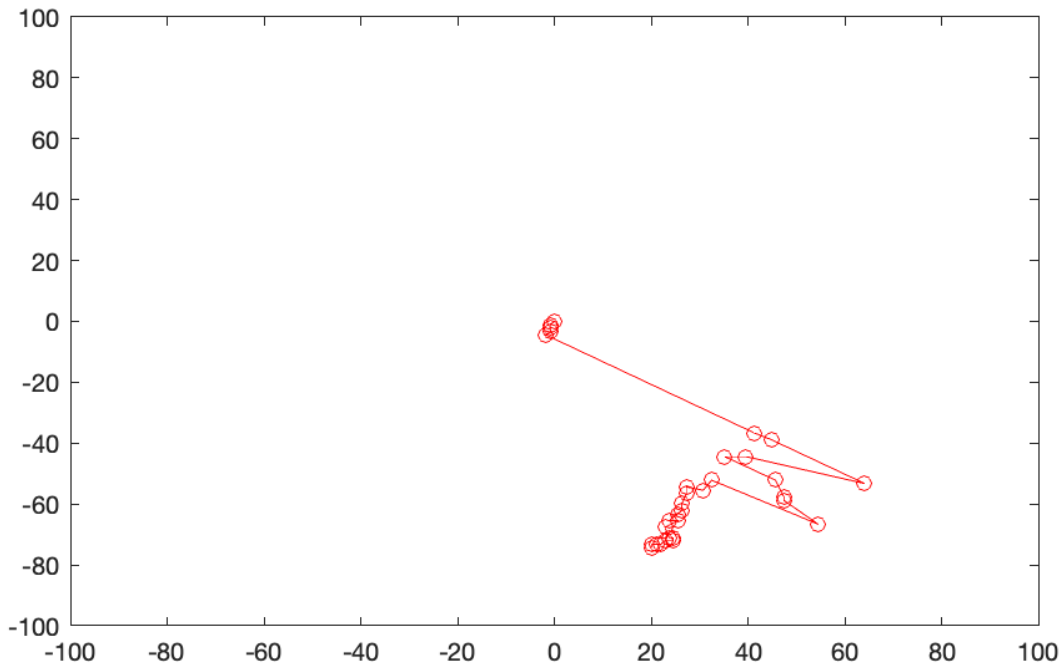
```
% x position vs. y position
```

```
plot(myData(:,2), myData(:,3),'ro-')
```



- A specific **domain and range** may be specified using `xlim` or `ylim`

```
xlim([-100,100]) % set x limit
ylim([-100,100]) % set y limit
```

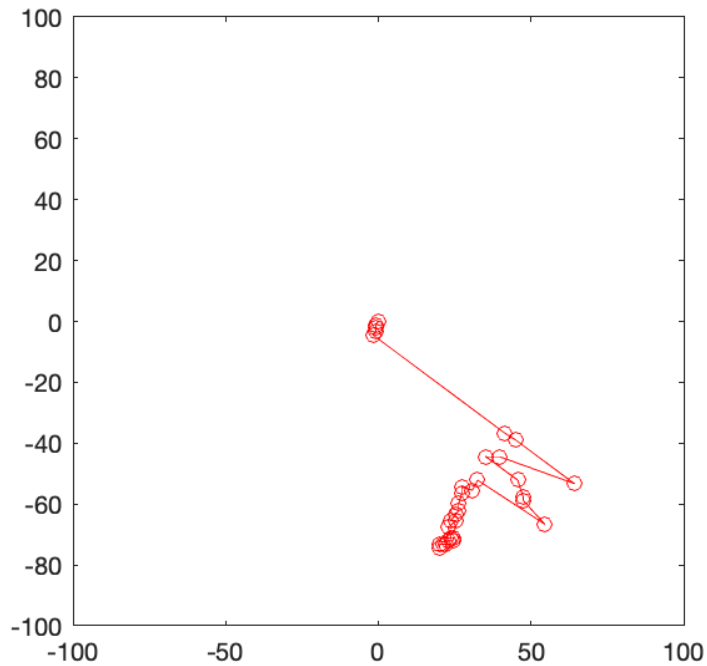


- The `axis` command offers many predefined axis scaling/sizing operations

Useful documentation: https://www.mathworks.com/help/matlab/ref/axis.html

```
 axis auto       % Automatically choose all axis limits
 axis tight      % Fit the axes box tightly around the data by setting the
axis limits equal to the range of the data.
 axis equal      % Use the same length for the data units along each axis
 axis square     % Use axis lines with equal lengths
 axis off        % Turn off axes
 axis ij         % Reverse y axis direction
```

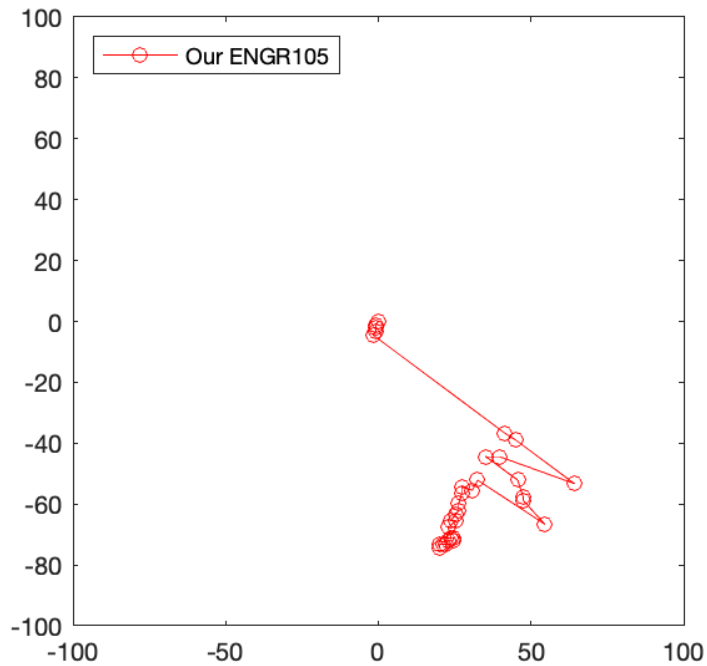
```
axis square %change shape of axis
```



<u>Note</u>: specifying a parameter like this operates on the previous plot. Information about the previous plot is stored as the **current axis** and **current figure**.

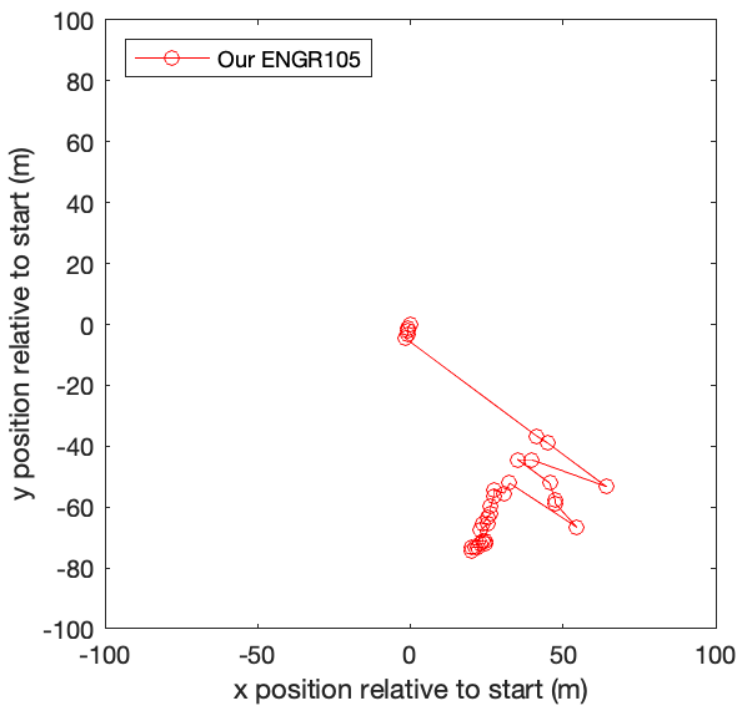
- **Legends** may be added and the position of the legend specified

```
legend('Our ENGR105', 'Location', 'NorthWest');
```



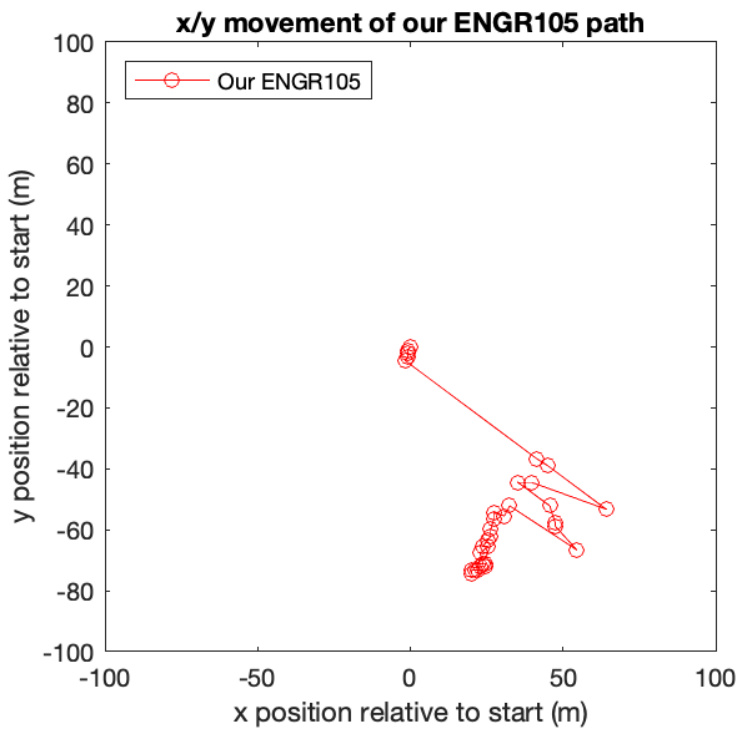
- **Axis labels and titles** may be placed on the graph

```
xlabel('x position relative to start (m)')
ylabel('y position relative to start (m)')
```



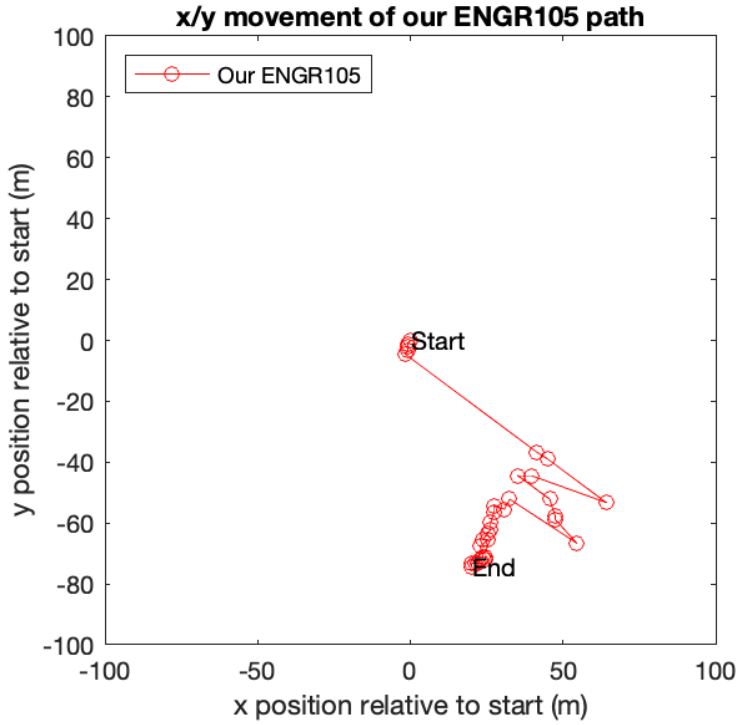
- We can add a **title**

```
title('x/y movement of our ENGR105 path')
```



- **Text may be placed on a plot** using the `text` command

```
text(myData(1,2), myData(1,3), 'Start')
text(myData(end,2), myData(end,3), 'End')
```

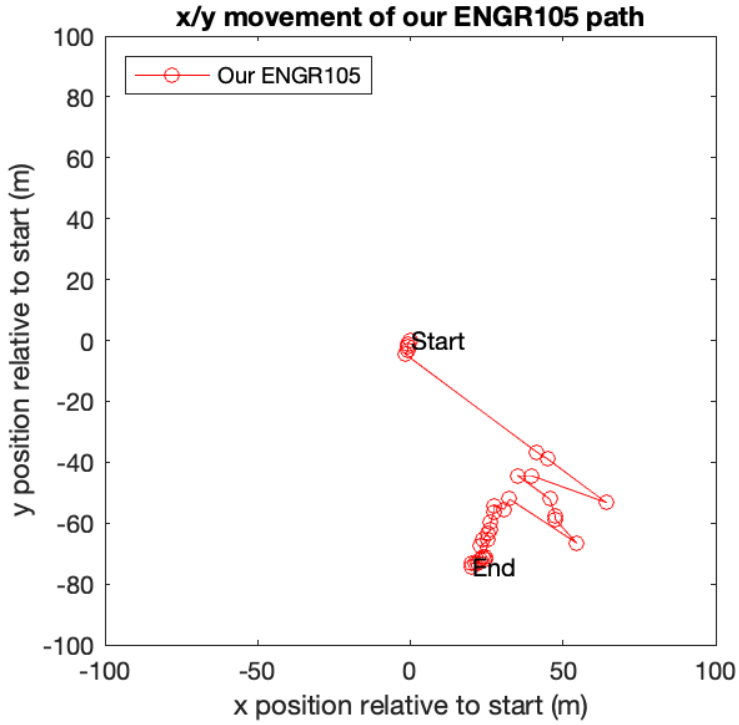


Important note regarding LiveScript files

We are using LiveScript files because these are a really convenient way to combine text and code, ultimately because we want these lecture notes to be an accessible resource for you.

However, when plotting in a `.mlx` file, all figures are embedded in the document (which is what we want). When you are working with a script file (`.m`) or at the command line, figures will "pop up" when called. **The following code will create a plot in a new figure window.**

```
% Display figure in its own window
set(gcf,'Visible','on');
```



Plots as objects

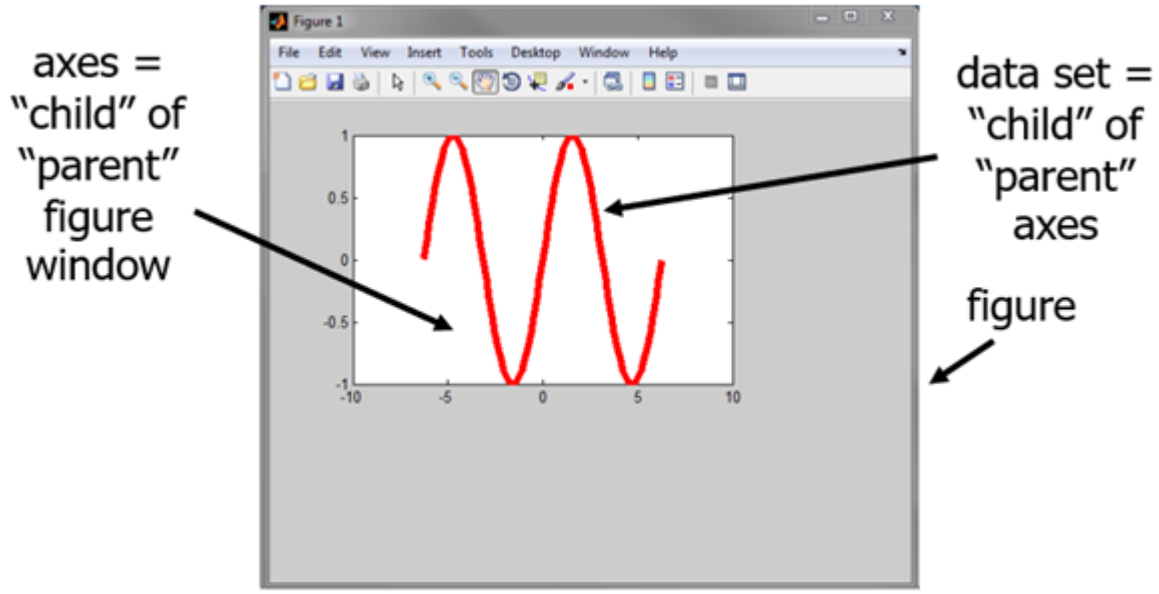
- Functions can be stored as **function handles**.
- Plots and graphics are inherently stored as **graphic handles**.

While these are not technically the same thing, they are both ways to store (at least temporarily) and access something as/from an (*explicitly* or *implicitly* defined) variable.

First go: Characteristics of the plot may be modified by referring to the **current axes or figure -- the one that is active.**

- `gca`: get current axes
- `gcf`: get current figure

The following is an example of a "typical" figure produced in the command line or a script file.



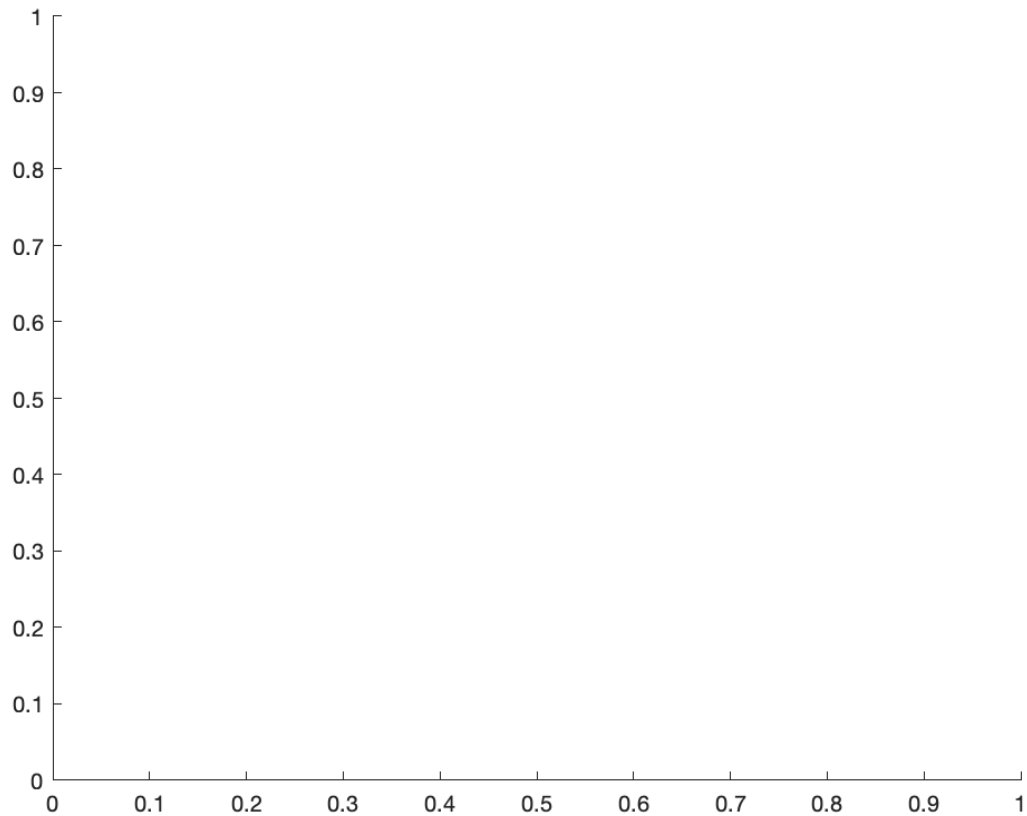
- A second plot command executed (without specifying `hold on`) will overwrite the axes from a first plot.

<u>Important</u>: When working at the command line or in a .m file, new figures can be created using the `figure` function, which will establish a new parent figure window. In a LiveScript document, **all figures are set to "not visible" by default and you see the axes**.

```
figure
```

When invoking the `axes()` command, MATLAB creates a "blank" set of axes. If invoked outside a LiveScript document, a new figure window will also be created to contain it.

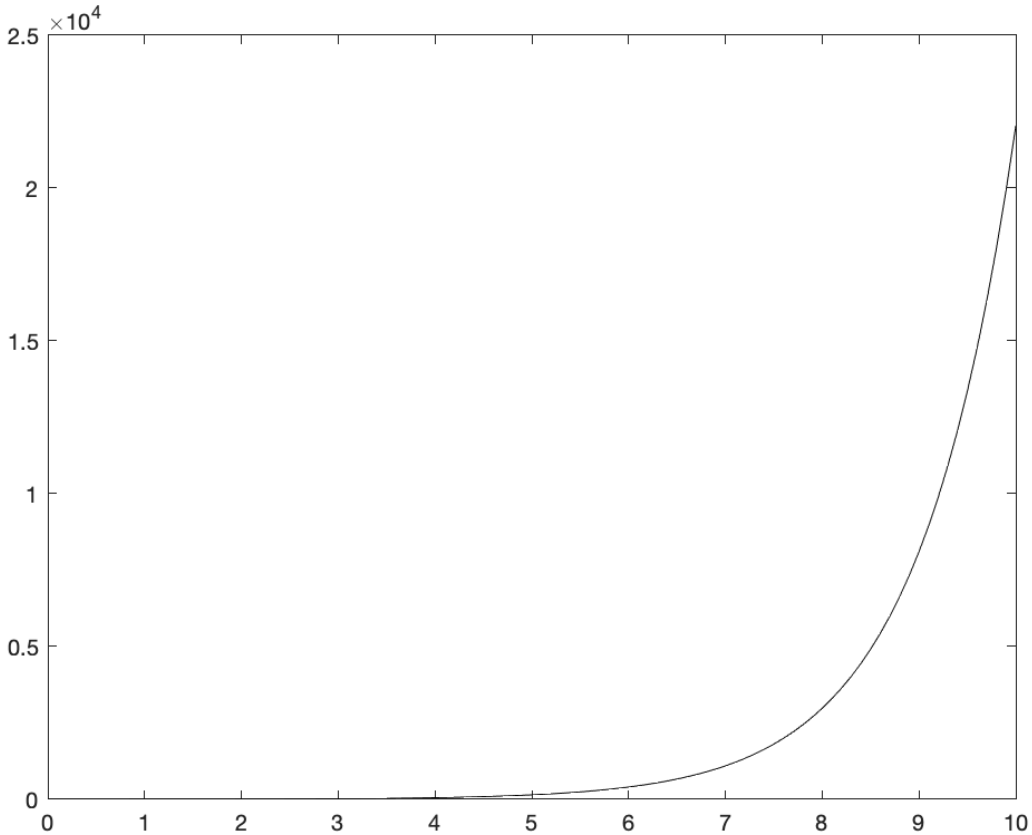
```
axes
```



Getting axes or figure properties

Current figure or axis parameters may be viewed by using the `get` function. **This is often a helpful way to understand what potential parameters may be modified.**

```
% Create new plot of f(t) = exp(t)
t = 0:0.1:10;
f = exp(t);
plot(t,f,'k-')
get(gca)
```




```
                ALim: [0 1]
            ALimMode: 'auto'
          AlphaScale: 'linear'
            Alphamap: [1×64 double]
   AmbientLightColor: [1 1 1]
        BeingDeleted: off
                 Box: on
            BoxStyle: 'back'
          BusyAction: 'queue'
       ButtonDownFcn: ''
                CLim: [0 1]
            CLimMode: 'auto'
      CameraPosition: [5 12500 17.3205]
  CameraPositionMode: 'auto'
        CameraTarget: [5 12500 0]
    CameraTargetMode: 'auto'
      CameraUpVector: [0 1 0]
  CameraUpVectorMode: 'auto'
     CameraViewAngle: 6.6086
 CameraViewAngleMode: 'auto'
            Children: [1×1 Line]
            Clipping: on
       ClippingStyle: '3dbox'
               Color: [1 1 1]
          ColorOrder: [7×3 double]
     ColorOrderIndex: 2
          ColorScale: 'linear'
            Colormap: [256×3 double]
```

```
                  ContextMenu: [0×0 GraphicsPlaceholder]
                    CreateFcn: ''
                 CurrentPoint: [2×3 double]
              DataAspectRatio: [5 12500 1]
          DataAspectRatioMode: 'auto'
                    DeleteFcn: ''
                    FontAngle: 'normal'
                     FontName: 'Helvetica'
                     FontSize: 10
                 FontSizeMode: 'auto'
                FontSmoothing: on
                    FontUnits: 'points'
                   FontWeight: 'normal'
                    GridAlpha: 0.1500
                GridAlphaMode: 'auto'
                    GridColor: [0.1500 0.1500 0.1500]
                GridColorMode: 'auto'
                GridLineStyle: '-'
             HandleVisibility: 'on'
                      HitTest: on
                InnerPosition: [0.1300 0.1100 0.7750 0.8150]
                 Interactions: [1×1
matlab.graphics.interaction.interface.DefaultAxesInteractionSet]
                Interruptible: on
      LabelFontSizeMultiplier: 1.1000
                        Layer: 'bottom'
                       Layout: [0×0 matlab.ui.layout.LayoutOptions]
                       Legend: [0×0 GraphicsPlaceholder]
               LineStyleOrder: '-'
          LineStyleOrderIndex: 1
                    LineWidth: 0.5000
               MinorGridAlpha: 0.2500
           MinorGridAlphaMode: 'auto'
               MinorGridColor: [0.1000 0.1000 0.1000]
           MinorGridColorMode: 'auto'
           MinorGridLineStyle: ':'
                     NextPlot: 'replace'
              NextSeriesIndex: 2
                OuterPosition: [0 0 1 1]
                       Parent: [1×1 Figure]
                PickableParts: 'visible'
           PlotBoxAspectRatio: [1 0.7892 0.7892]
       PlotBoxAspectRatioMode: 'auto'
                     Position: [0.1300 0.1100 0.7750 0.8150]
           PositionConstraint: 'outerposition'
                   Projection: 'orthographic'
                     Selected: off
           SelectionHighlight: on
                   SortMethod: 'childorder'
                     Subtitle: [1×1 Text]
           SubtitleFontWeight: 'normal'
                          Tag: ''
                      TickDir: 'in'
                  TickDirMode: 'auto'
         TickLabelInterpreter: 'tex'
                   TickLength: [0.0100 0.0250]
                   TightInset: [0.0321 0.0393 0.0103 0.0369]
                        Title: [1×1 Text]
      TitleFontSizeMultiplier: 1.1000
              TitleFontWeight: 'bold'
     TitleHorizontalAlignment: 'center'
```

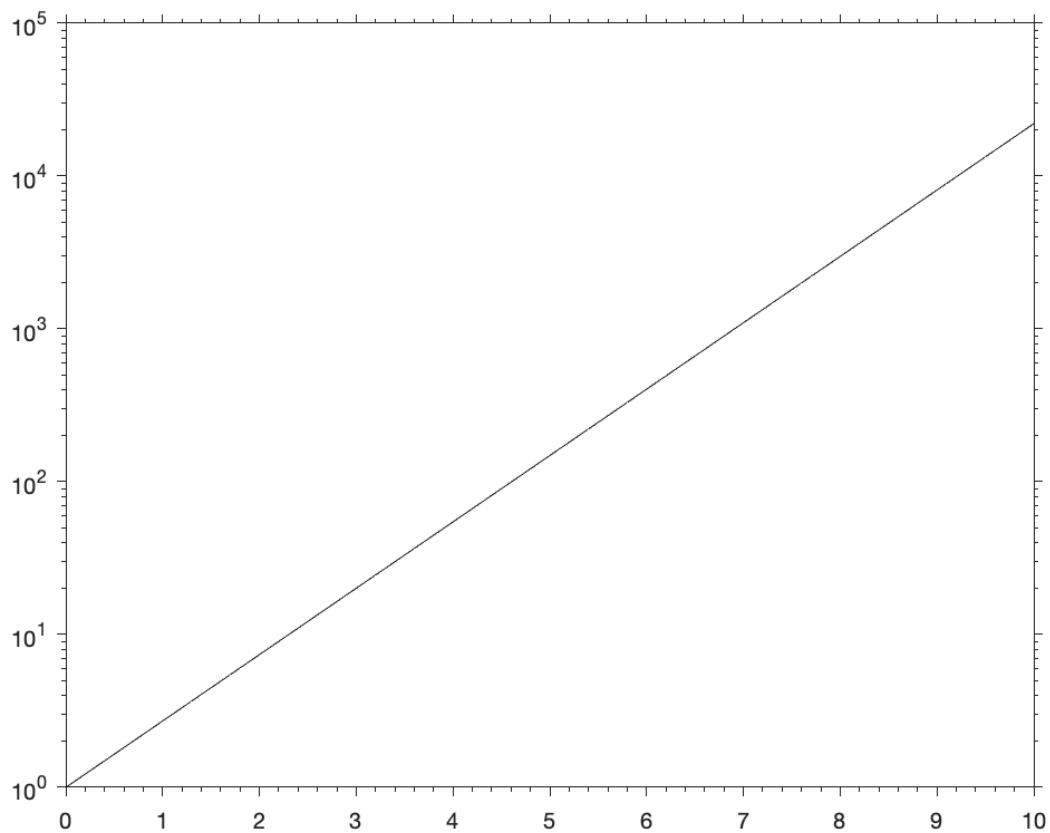
```
                 Toolbar: [1×1 AxesToolbar]
                    Type: 'axes'
                   Units: 'normalized'
                UserData: []
                    View: [0 90]
                 Visible: on
                    XAxis: [1×1 NumericRuler]
            XAxisLocation: 'bottom'
                   XColor: [0.1500 0.1500 0.1500]
               XColorMode: 'auto'
                     XDir: 'normal'
                    XGrid: off
                   XLabel: [1×1 Text]
                     XLim: [0 10]
                 XLimMode: 'auto'
               XMinorGrid: off
               XMinorTick: off
                   XScale: 'linear'
                    XTick: [0 1 2 3 4 5 6 7 8 9 10]
               XTickLabel: {11×1 cell}
           XTickLabelMode: 'auto'
       XTickLabelRotation: 0
                XTickMode: 'auto'
                    YAxis: [1×1 NumericRuler]
            YAxisLocation: 'left'
                   YColor: [0.1500 0.1500 0.1500]
               YColorMode: 'auto'
                     YDir: 'normal'
                    YGrid: off
                   YLabel: [1×1 Text]
                     YLim: [0 25000]
                 YLimMode: 'auto'
               YMinorGrid: off
               YMinorTick: off
                   YScale: 'linear'
                    YTick: [0 5000 10000 15000 20000 25000]
               YTickLabel: {6×1 cell}
           YTickLabelMode: 'auto'
       YTickLabelRotation: 0
                YTickMode: 'auto'
                    ZAxis: [1×1 NumericRuler]
                   ZColor: [0.1500 0.1500 0.1500]
               ZColorMode: 'auto'
                     ZDir: 'normal'
                    ZGrid: off
                   ZLabel: [1×1 Text]
                     ZLim: [-1 1]
                 ZLimMode: 'auto'
               ZMinorGrid: off
               ZMinorTick: off
                   ZScale: 'linear'
                    ZTick: [-1 0 1]
               ZTickLabel: ''
           ZTickLabelMode: 'auto'
       ZTickLabelRotation: 0
                ZTickMode: 'auto'
```

Setting axes and figure properties

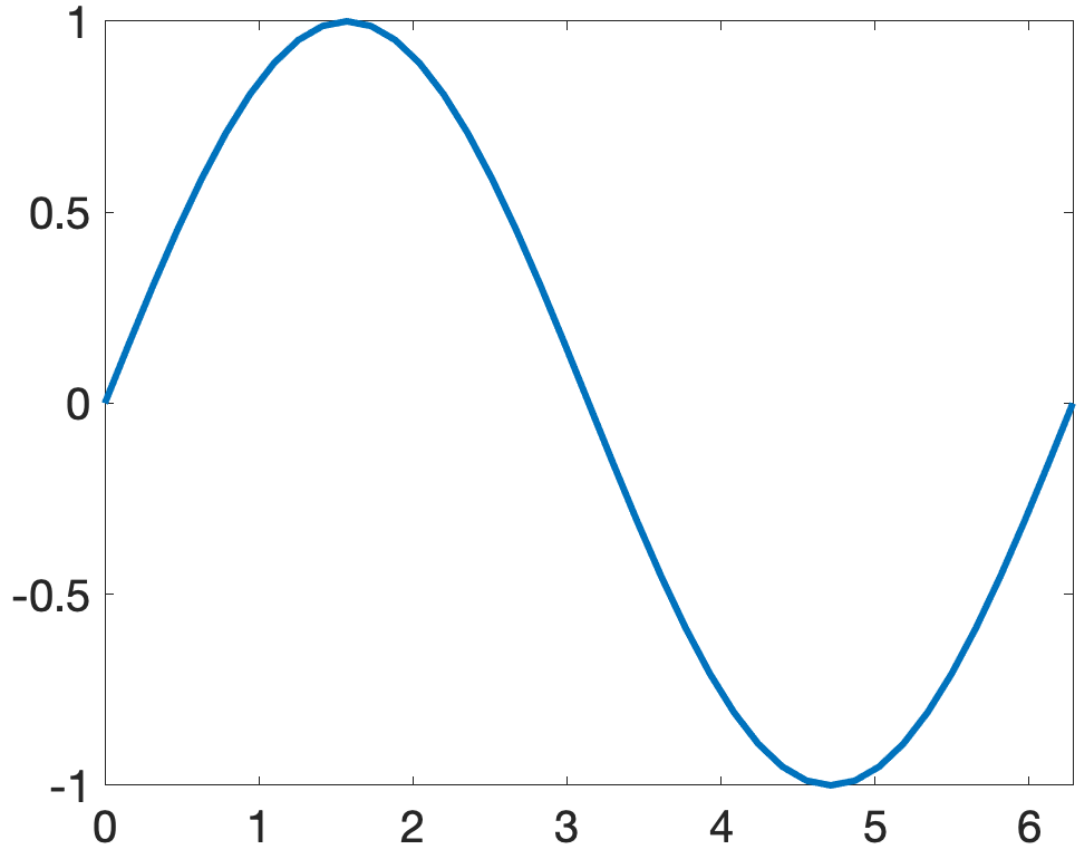
- `gca` or `gcf` may be combined with the `set` command to modify a plot characteristic.
- The general form for the `set` command is

```
set(<ref to handle>,<property name>,<property value>)
```

```
% Change some parameters
set(gca,'YScale','log')
set(gca,'XMinorTic','on')
set(gca,'TickDir','out')
```




```
x = 0:pi/20:2*pi;
y = sin(x);
plot(x,y,'LineWidth',3)
% Increase the font size of the figure
set(gca,'FontSize',20)
```



Note: the plot that was produced most recently or is active is stored as `gca` or `gcf`. In the above case, `set` operates on that set of axes.

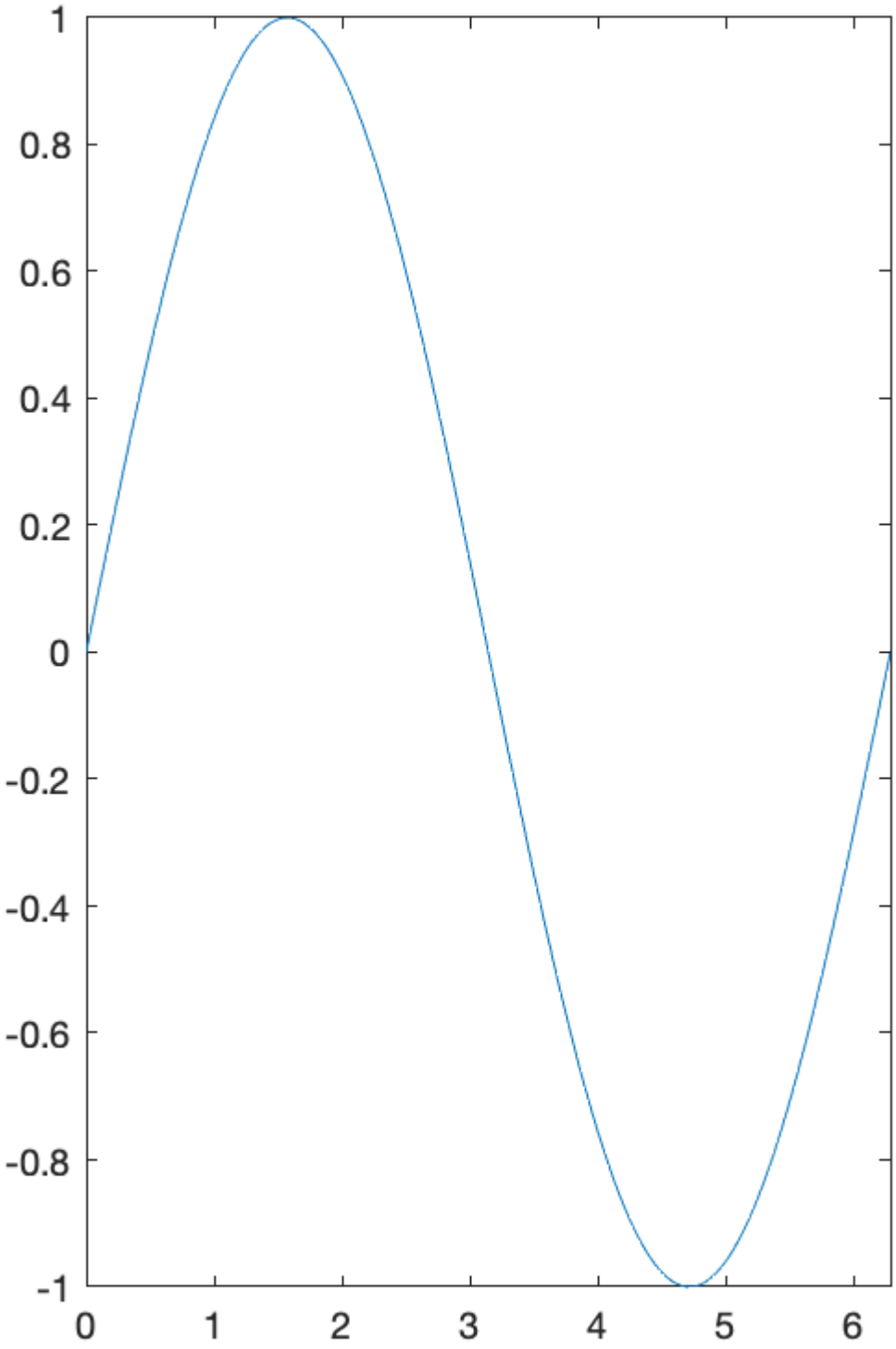
Graphics handles

In the same way that we created function handles to store functions in a variable, you can **create graphics handles** to store graphics in a variable.

```
% Example handles
myHandle1 = plot(x,y);
myFig = figure;
h = axes;
foo = axes;
```

```
% Example graphics handle use
x = 0:pi/100:2*pi;
y = sin(x);

myFigure = figure;
plot(x,y)

% Set the background of the figure
```

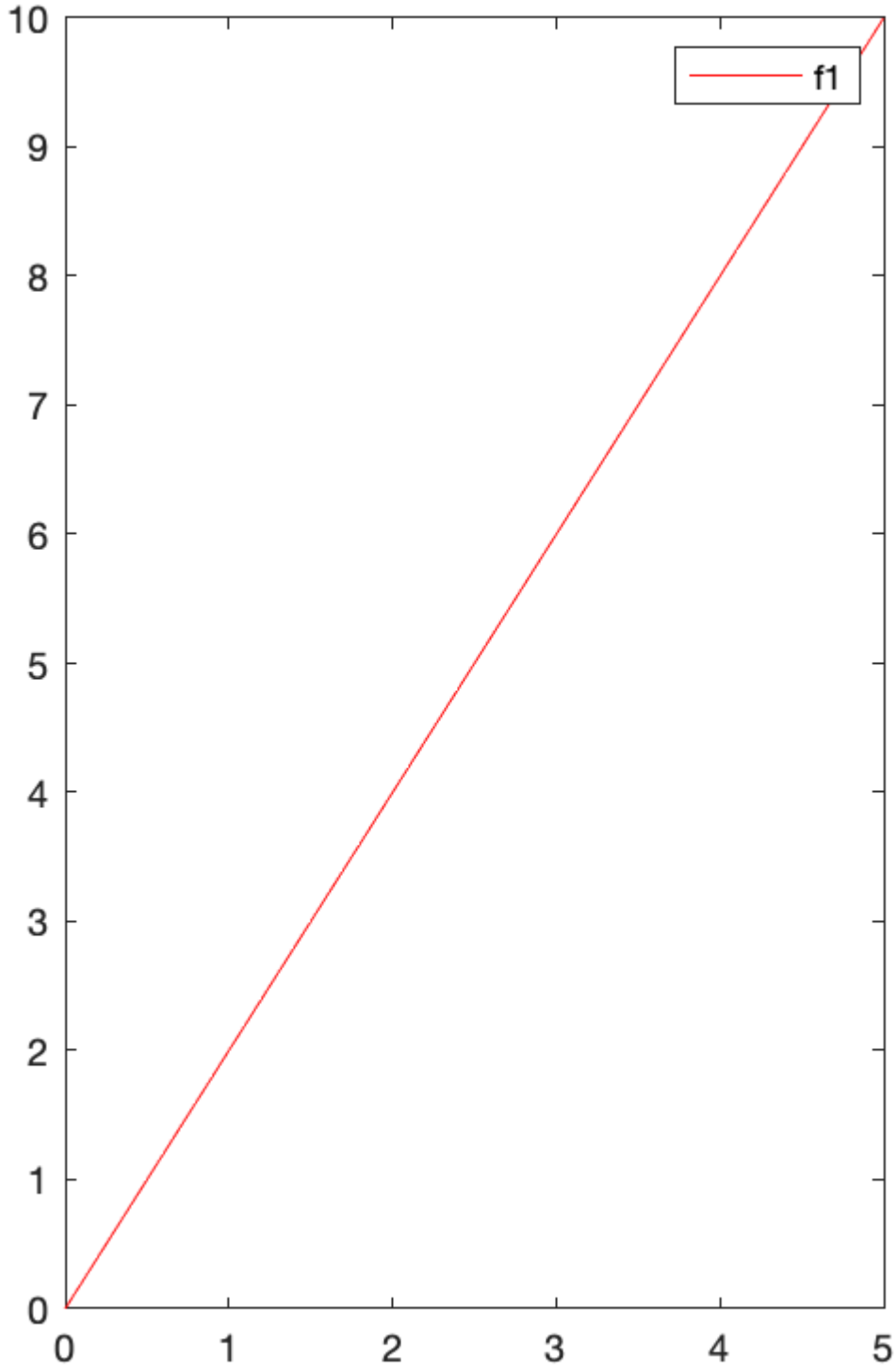
```
% to white
set(myFigure,'Color','w')

% Take the default plot position and
% reduce the width by 1/2
pos = get(myFigure,'Position');
pos(3) = 0.5*pos(3);
set(myFigure,'Position',pos);
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

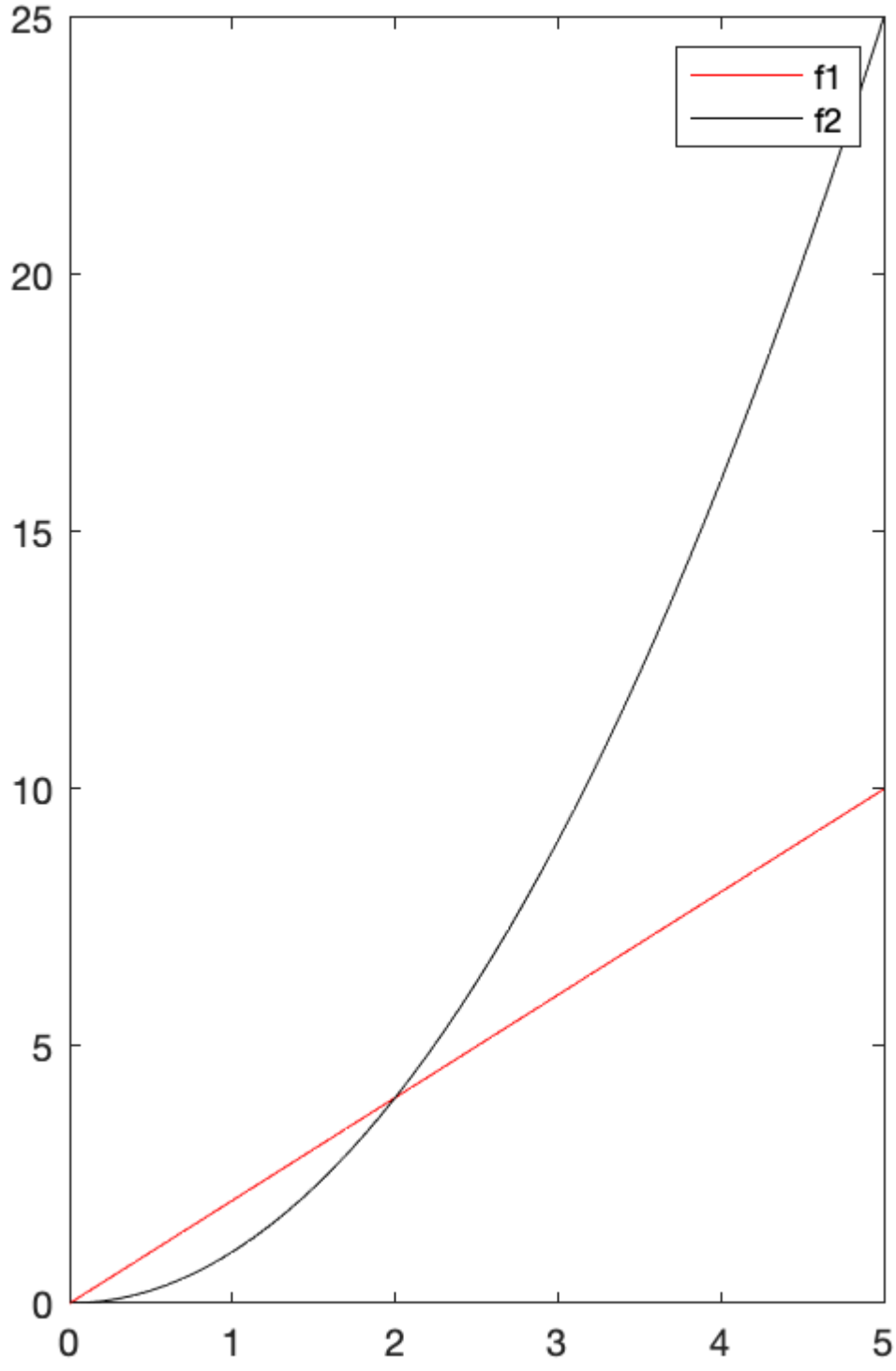
Question 1: my legend / plot / graphics object isn't working. What do I do?

- **Step 1**: Double check code for errors
- **Step 2**: Close all graphics objects and figures (`close all`)
- **Step 3**: consult documentation and Google (mathworks Q/A)

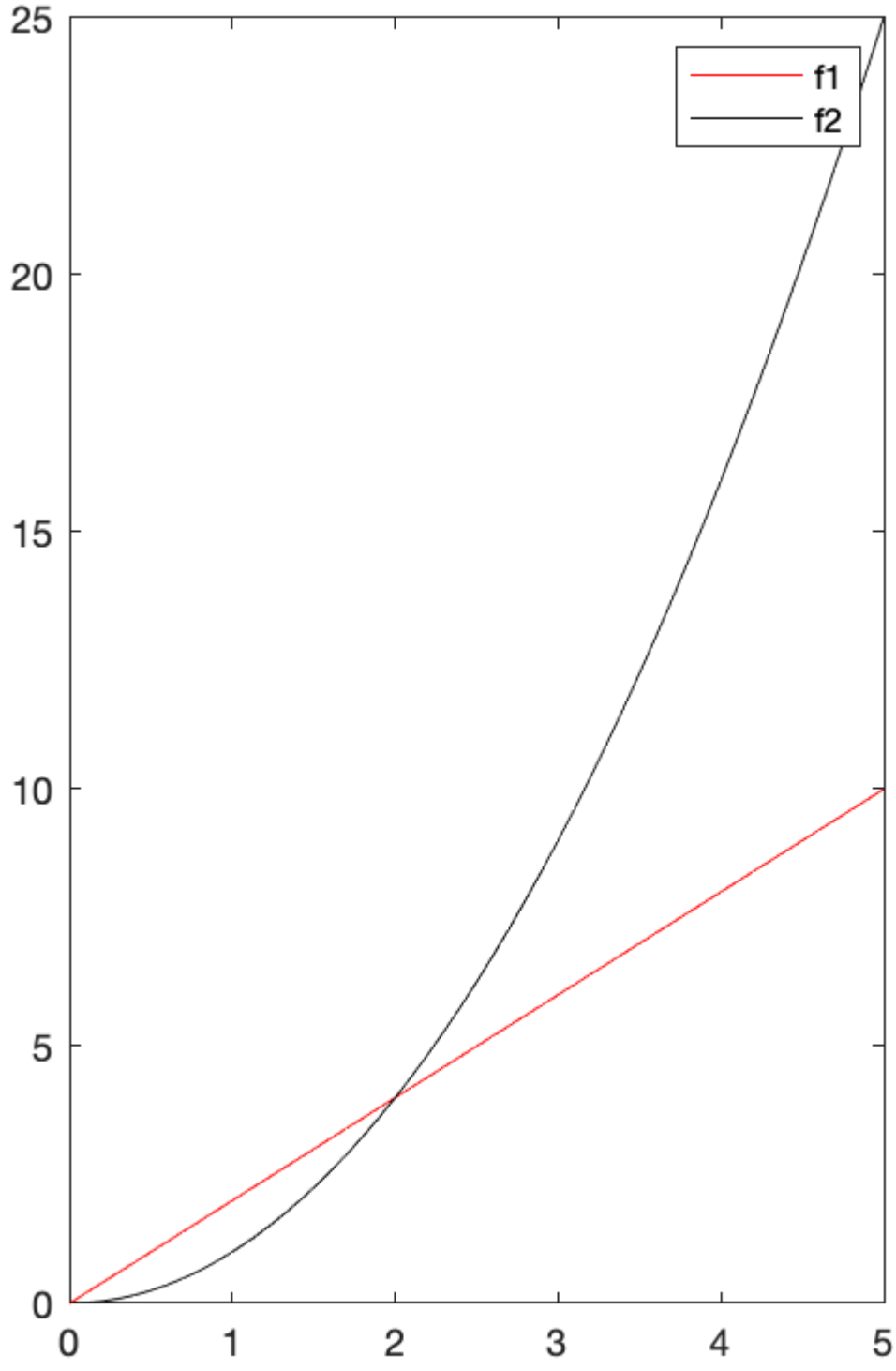
```
% This is the "usual" way we have added legends
t = 0:0.1:5;
f1 = 2*t;
f2 = t.^2;
plot(t,f1,'r-')
legend('f1')
```


f1
10
9
8
7
6
5
4
3
2
1
0
0
1
2
3
4
5

```
% Create graphics handles
h1 = plot(t,f1,'r-');
hold on
h2 = plot(t,f2,'k-');
hold off

% Create a legend by inputting. NOTE: this will change handle properties in
% h1 and h2!
h3 = legend([h1 h2],{'f1','f2'});
```


25
20
15
10
5
0
0
1
2
3
4
5
f1
f2

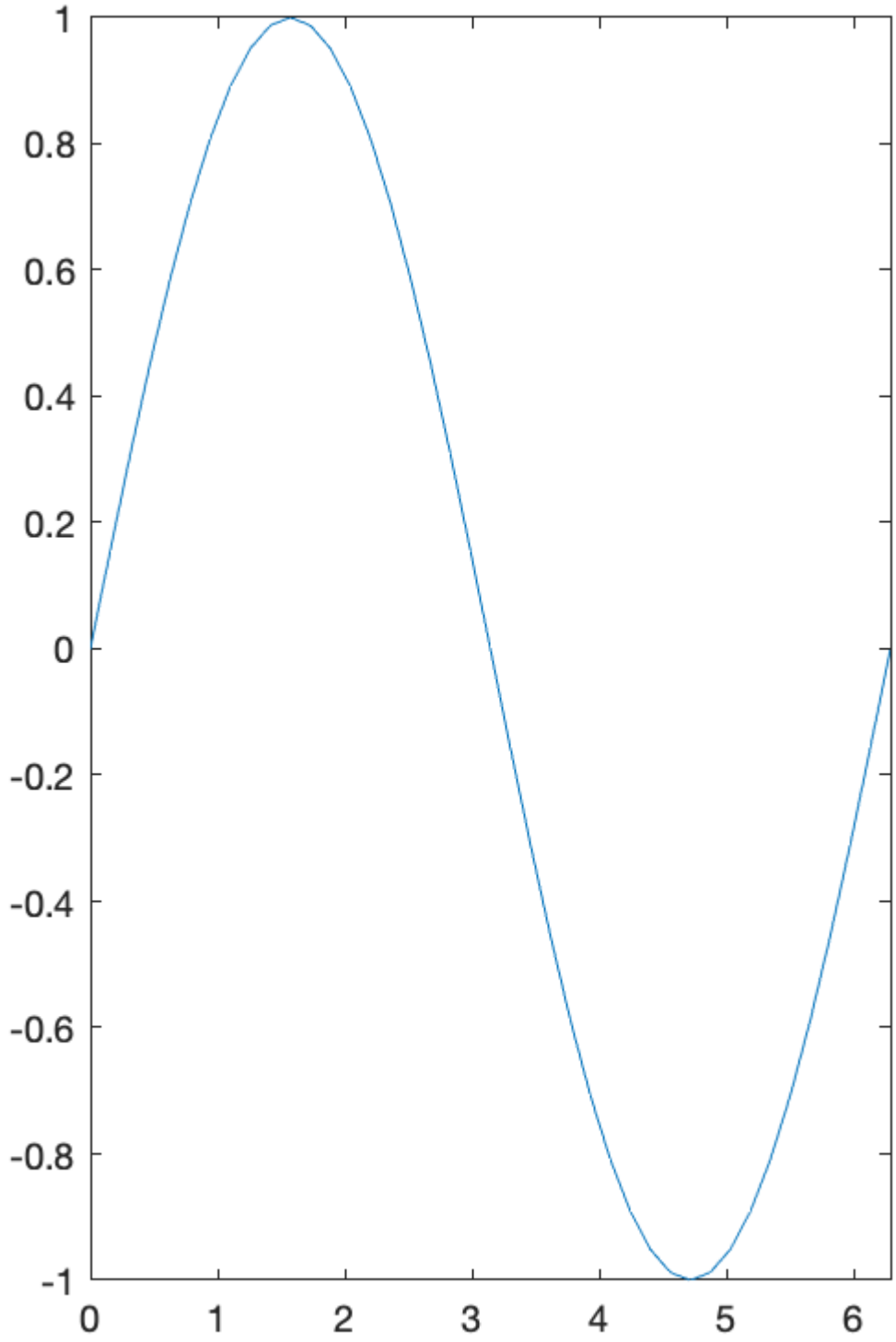
```
% Modify the the handle associated with each plot
plot(t,f1,'r-', 'DisplayName', 'f1');
hold on
plot(t,f2,'k-','DisplayName','f2');
hold off
legend
```


25
20
15
10
5
0
0
1
2
3
4
5
f1
f2

<u>Takeaway</u>: all of these are doing the same thing. If one doesn't work, try one of the alternatives.

Question 2: what happens when you specifiy a line spec for a plot?

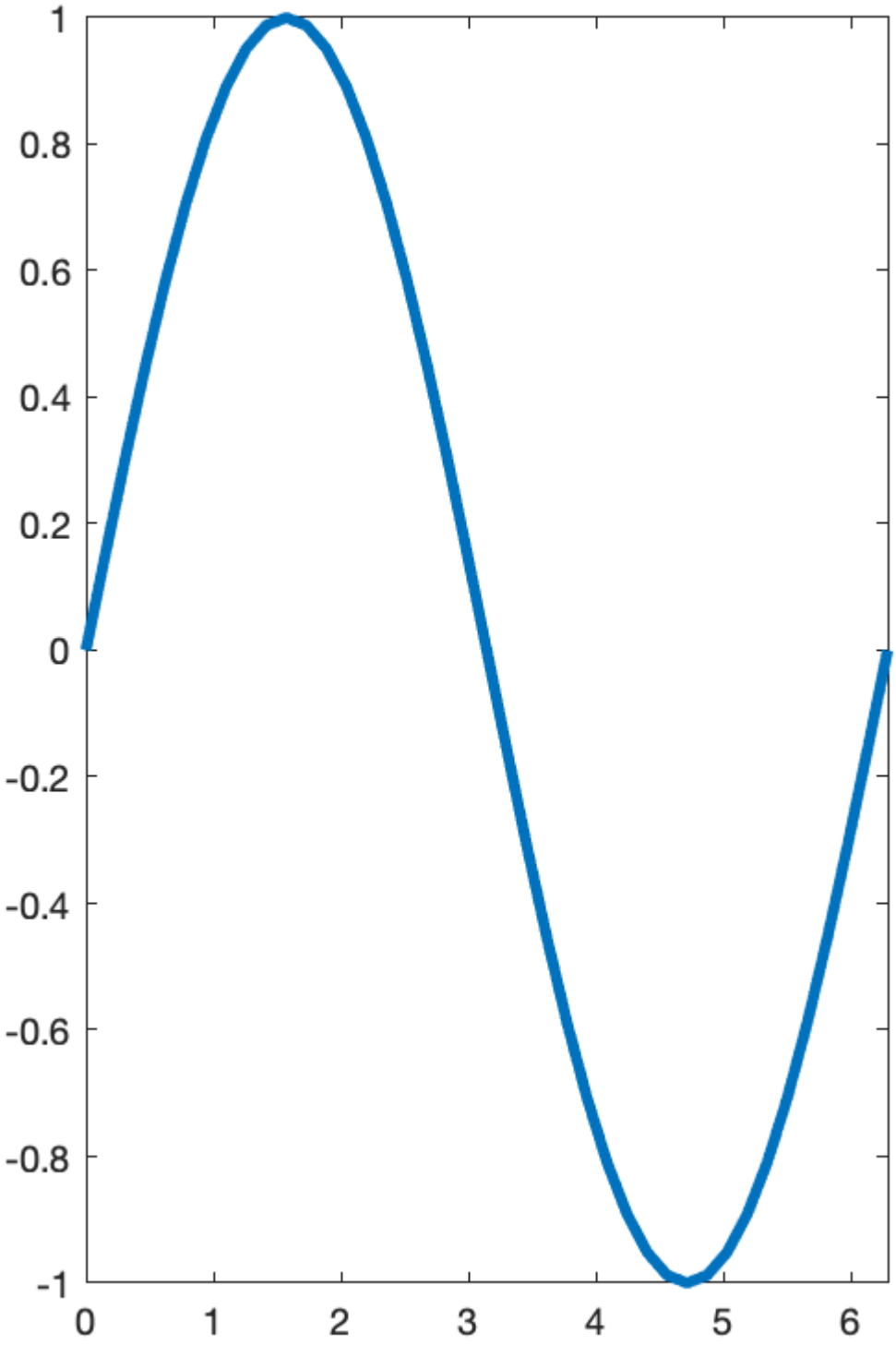
```
x = 0:pi/20:2*pi;
y = sin(x);
h1 = plot(x,y);
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

```
get(h1, 'LineWidth')
```

```
ans = 0.5000
```

```
h2 = plot(x,y,'LineWidth',3);
```


1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

```
get(h2, 'LineWidth')
```

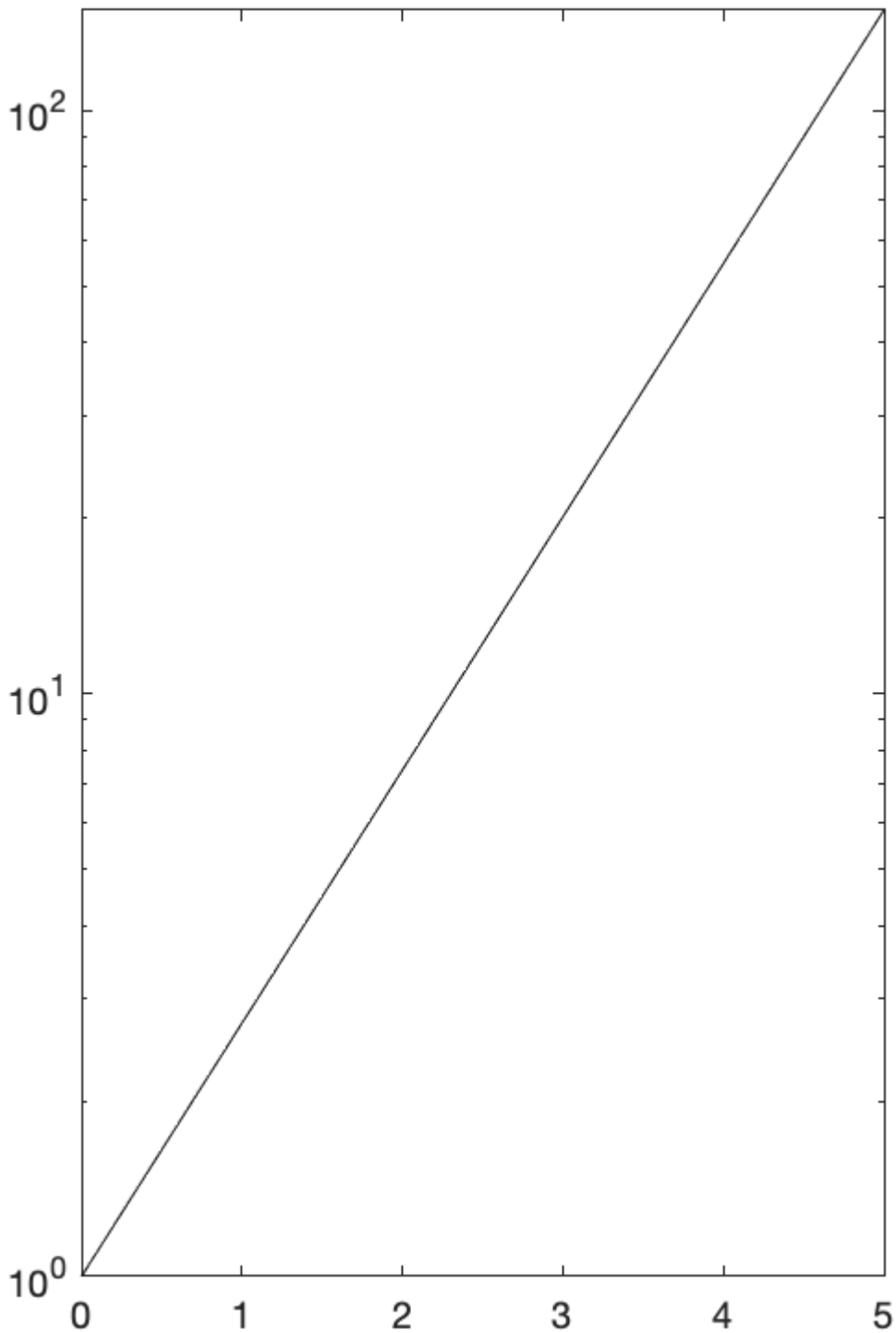
```
ans = 3
```

Specifiying a line specification or plotting option in the plot function is convenient for changing formatting perferences. It is equilavent to setting a field (parameter) for a graphics handle to a different value.

Linear vs log axes

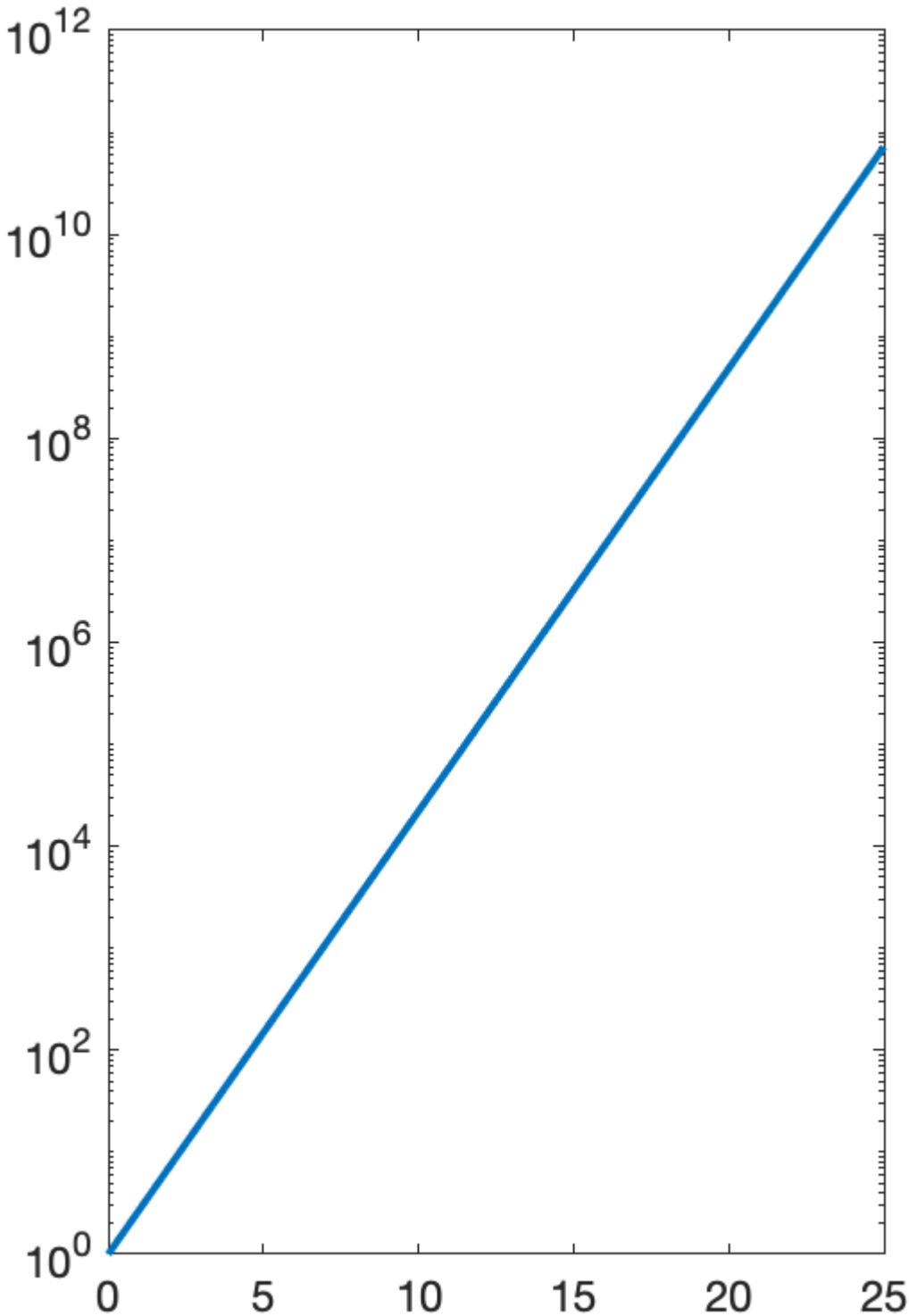
- Using `set` to **change axes properties**

```
% Create new plot of f(t) = exp(t)
t = 0:0.1:5;
f = exp(t);
plot(t,f,'k-')
% Set y scale to logarithmic
set(gca,'YScale','log')
```


10^2
10^1
10^0
0
1
2
3
4
5

- **Logarithmic y axis**: `semilogy`

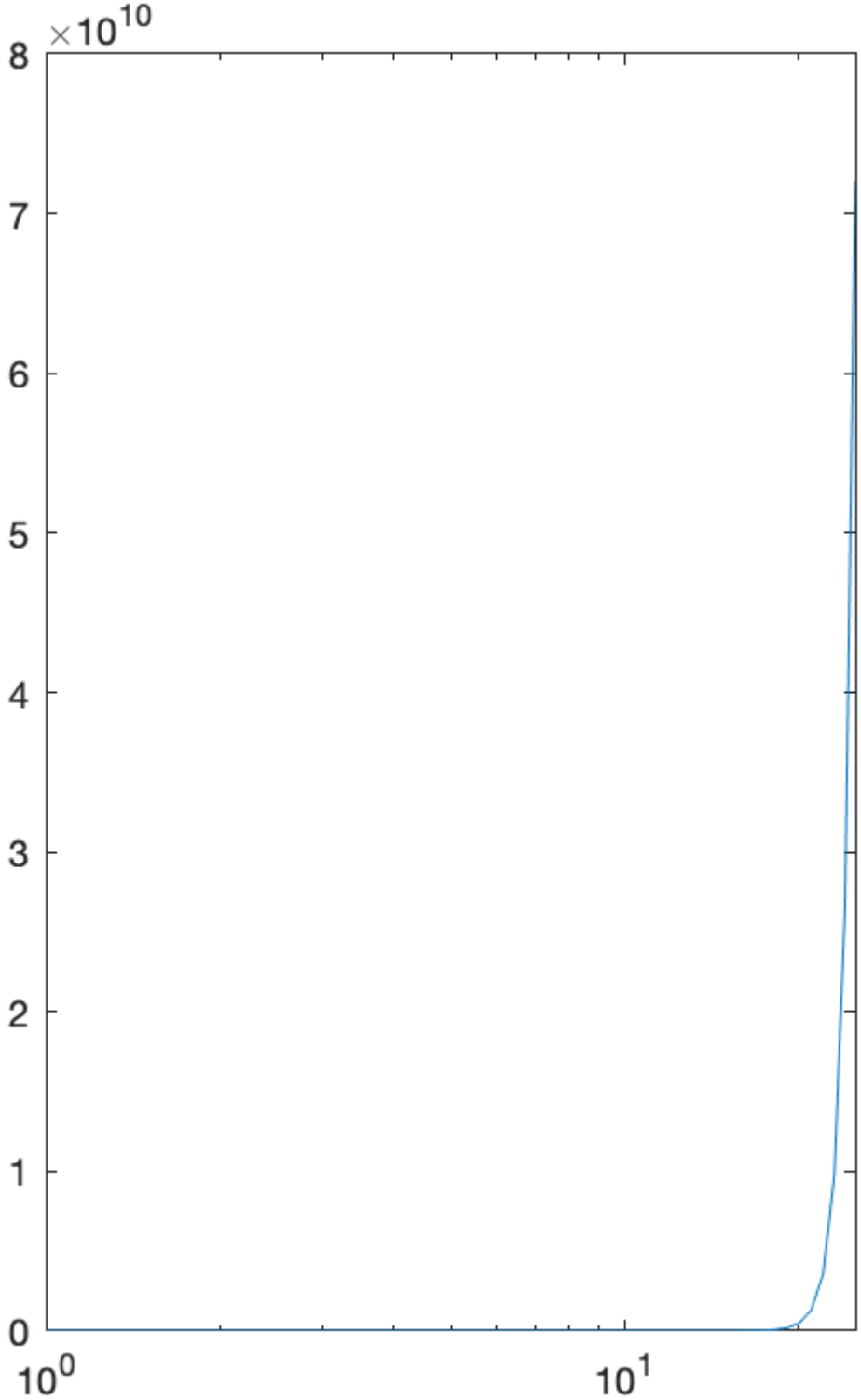
```
% x and y values
x = 0:1:25;
y = exp(x);

% Plot: y axis will have logarithmic scale
p = semilogy(x,y);
% Change the line width
set(p,'LineWidth',2)
set(p,'LineWidth',2)
set(gca,'FontSize',12)
```


10^{12}
10^{10}
10^{8}
10^{6}
10^{4}
10^{2}
10^{0}
0
5
10
15
20
25

<u>Note</u>: line parameters can be changed using set, as above, **OR** by specifying line specification into plotting function.

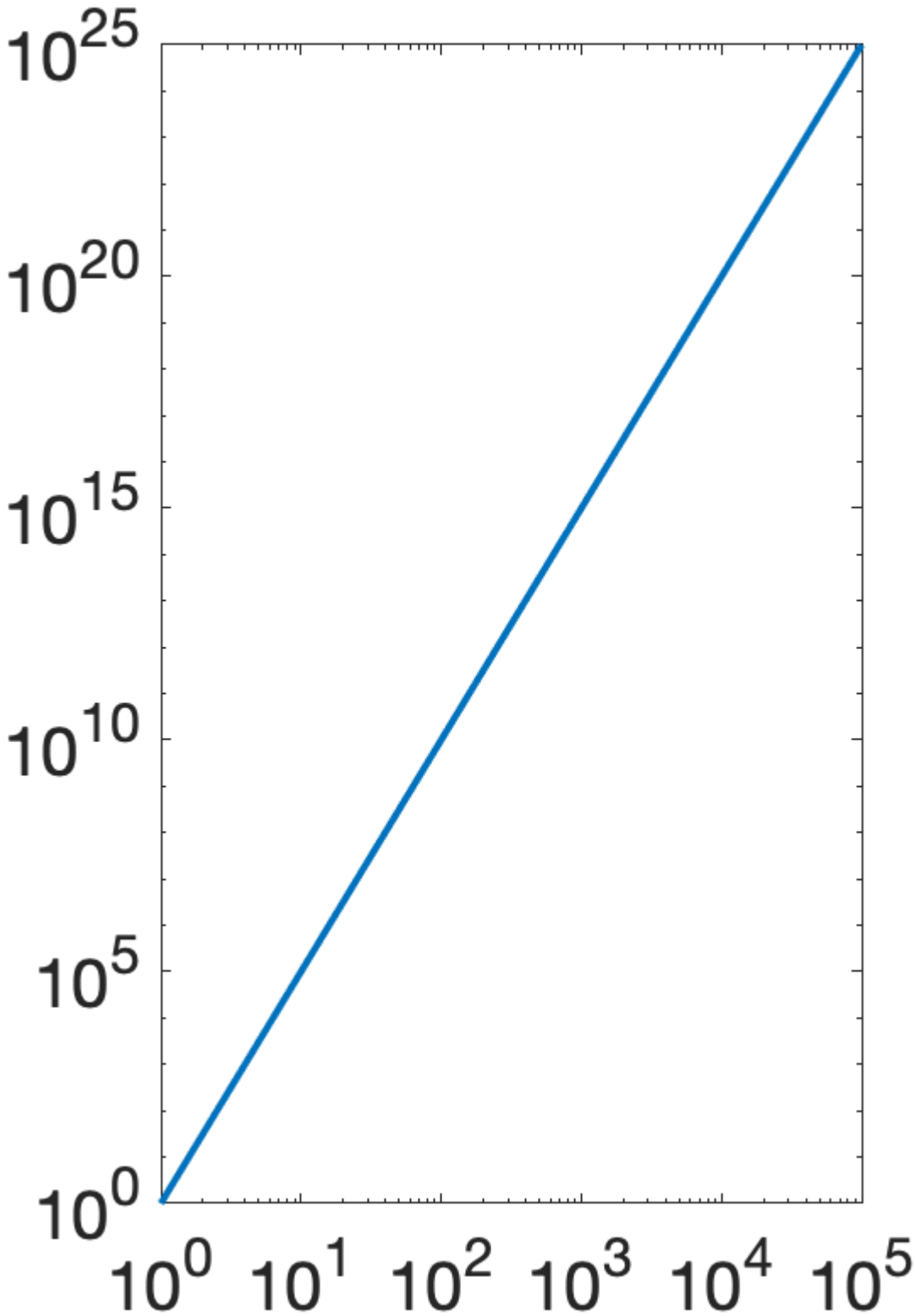
- **Logarithmic x axis**: `semilogx`

```
% Plot: y axis will have logarithmic scale
p = semilogx(x,y);
```


×10^10
8
7
6
5
4
3
2
1
0
10^0
10^1

- **Log-log plot**: `loglog`

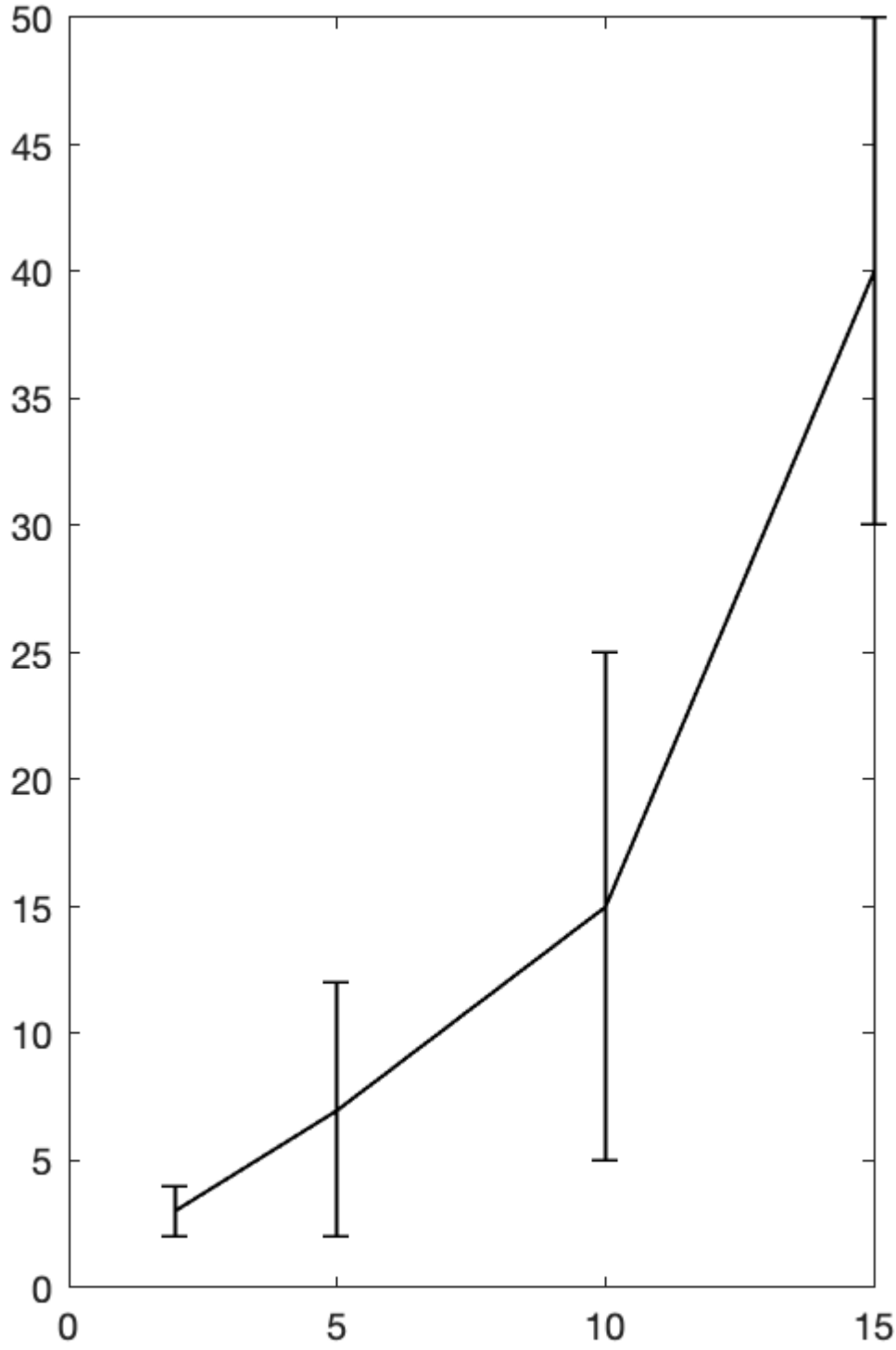
```
x = logspace(0,5,25);
y = x.^5;
p = loglog(x,y);

set(p,'LineWidth',2)
set(gca,'FontSize',20)
set(gca,'XTick',logspace(0,5,6))
```


10^{25}
10^{20}
10^{15}
10^{10}
10^{5}
10^{0}
10^{0} 10^{1} 10^{2} 10^{3} 10^{4} 10^{5}

Error bars

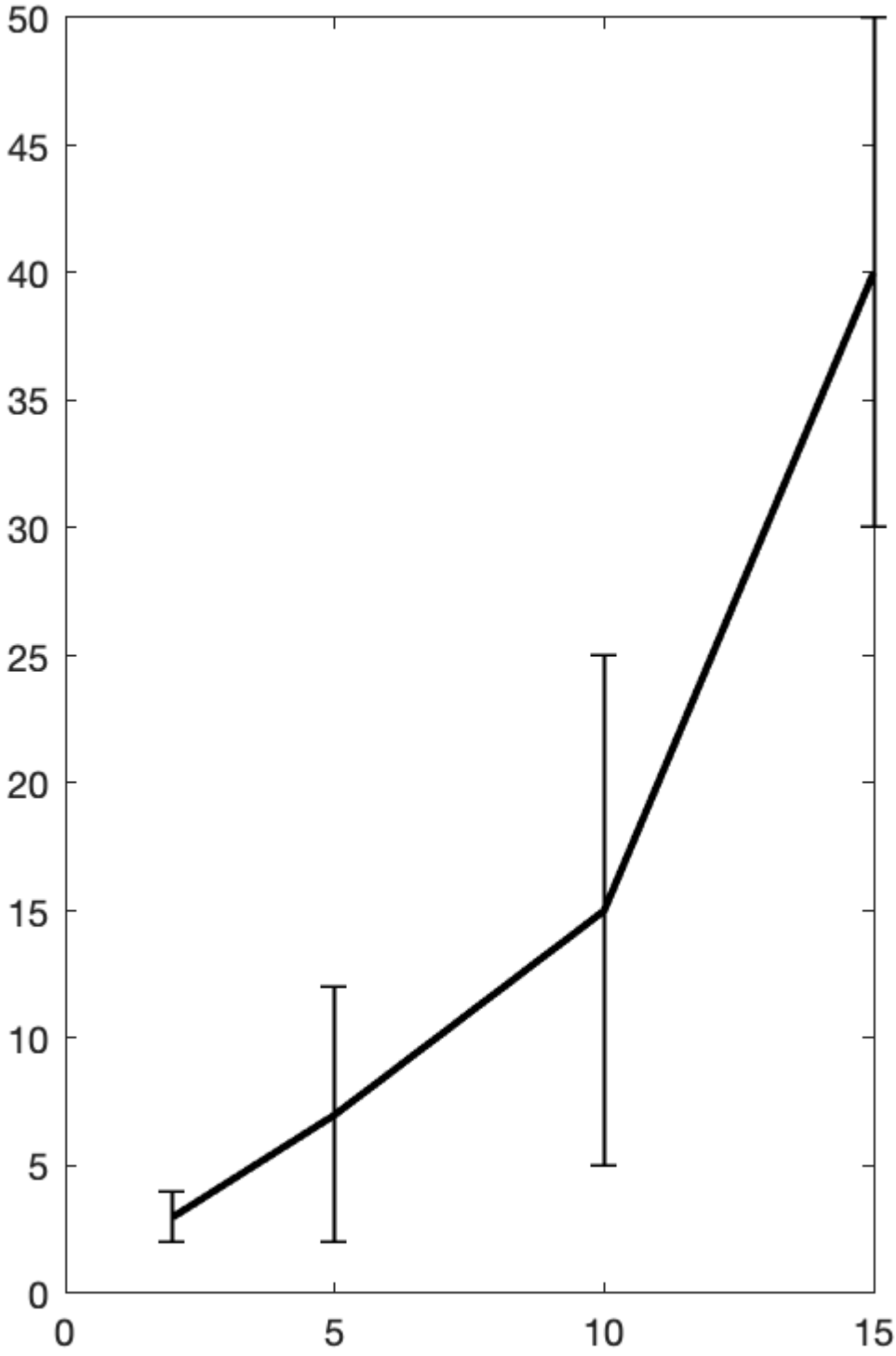
- **Y error bars**: use `errorbar` plotting function

```
% Data values
x = [2,5,10,15];
y = [3,7,15,40];

% Y error bars: positive and negative are same
ye = [1,5,10,10];

% Graphics handle representing error bar plot function
h = errorbar(x,y,ye,'k-');
set(h,'LineWidth',1)
```


50
45
40
35
30
25
20
15
10
5
0
0
5
10
15

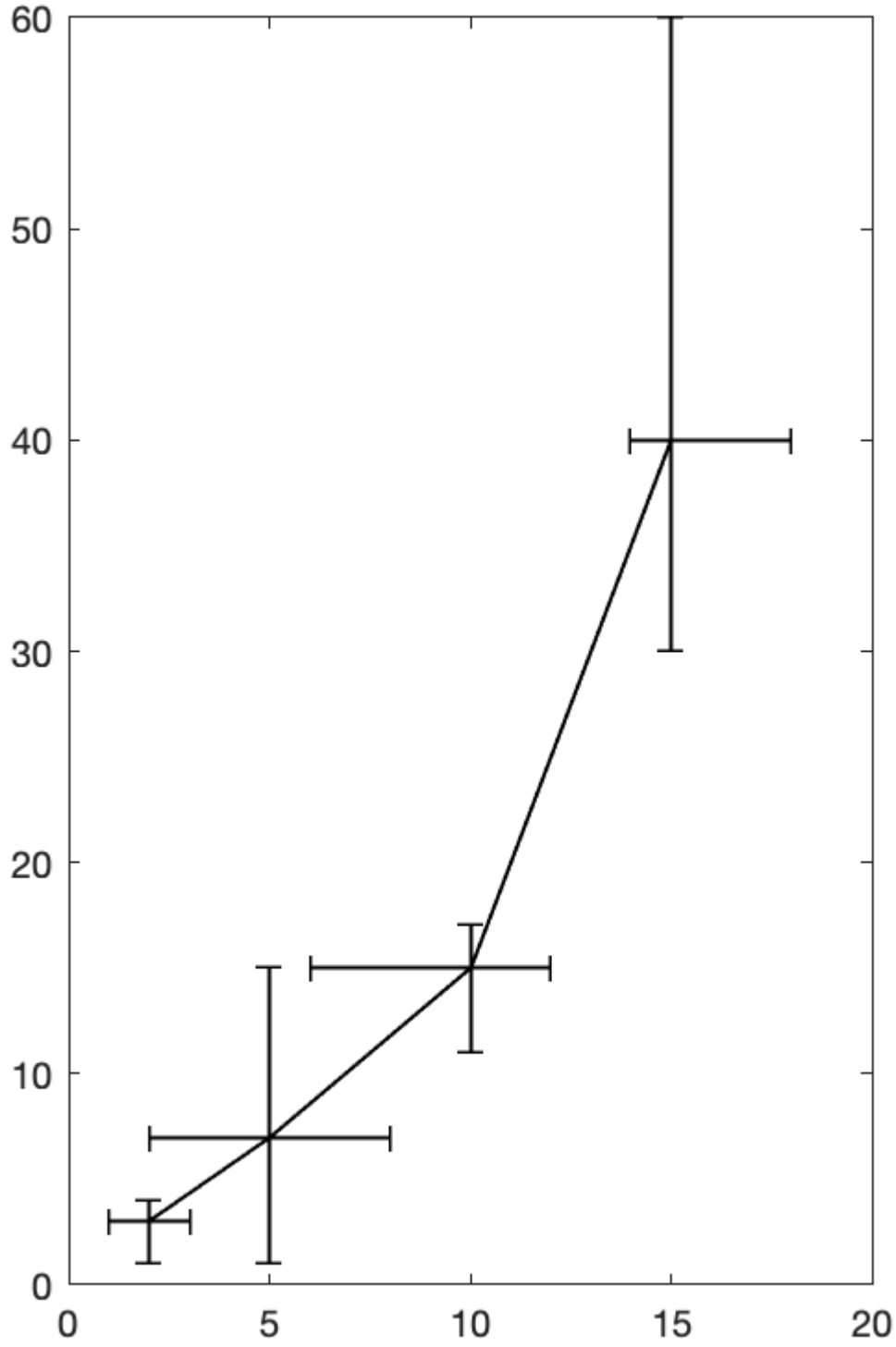
Example. How can we change error bar line properties and not the "data" line?

```
% Graphics handle representing error bar plot function
h = errorbar(x,y,ye,'k.','Marker','none');
hold on
plot(x,y,'k-','LineWidth',2)
hold off
% Change the parameters *ONLY* of the error bars
set(h,'LineWidth',1)
```


50
45
40
35
30
25
20
15
10
5
0
0
5
10
15

- **Y and X error bars**

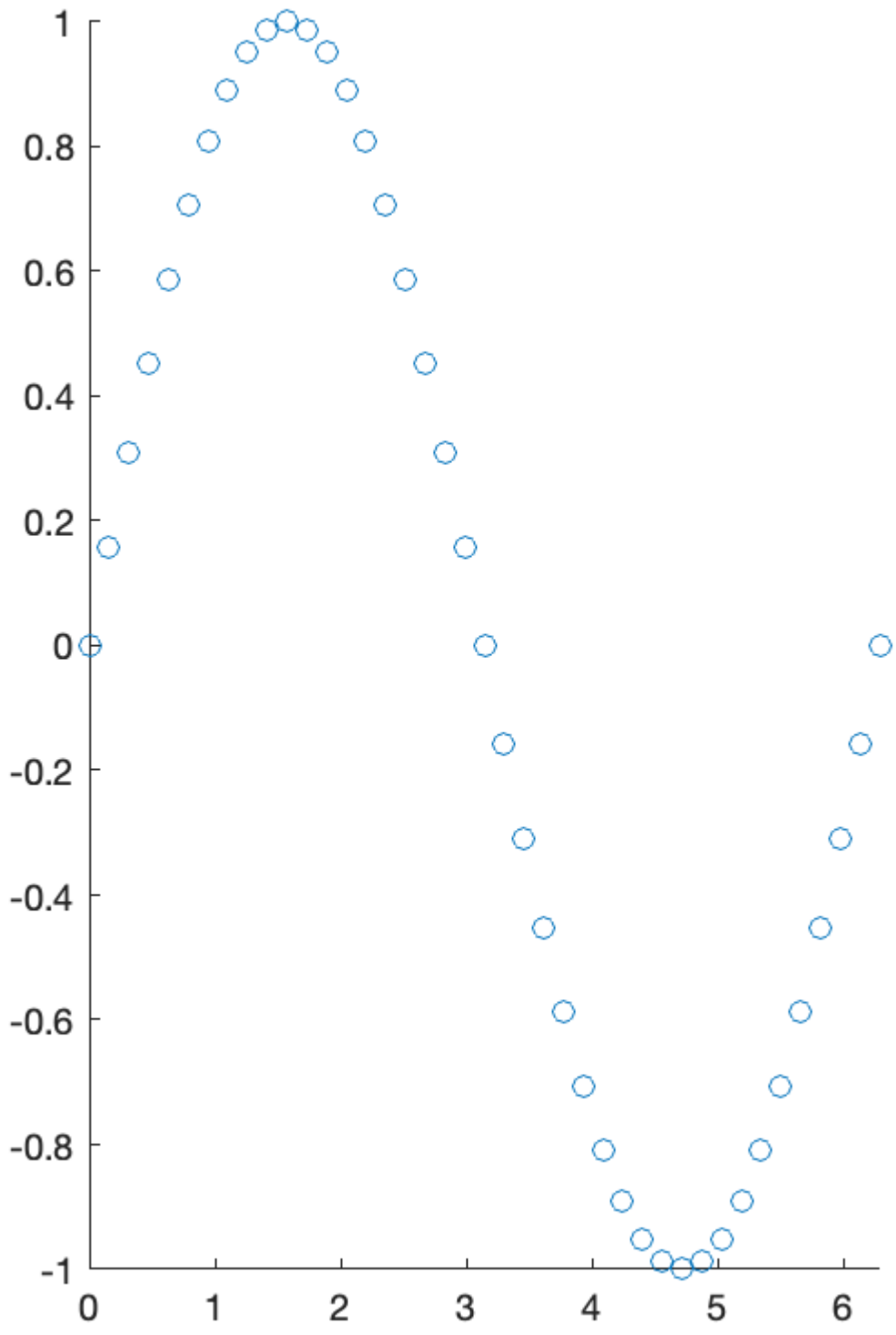
```
% Data values
x = [2,5,10,15];
y = [3,7,15,40];

% Positive and negative x error bars
xePos = [1,3,2,3];
xeNeg = [1,3,4,1];

% Positive and negative y error bars
yePos = [1,8,2,20];
yeNeg = [2,6,4,10];

% Create x and y error bar plot
h = errorbar(x,y,yeNeg,yePos,xeNeg,xePos,'k-');
set(h,'LineWidth',1)
```


60
50
40
30
20
10
0
0
5
10
15
20

Additional 2D plotting functions

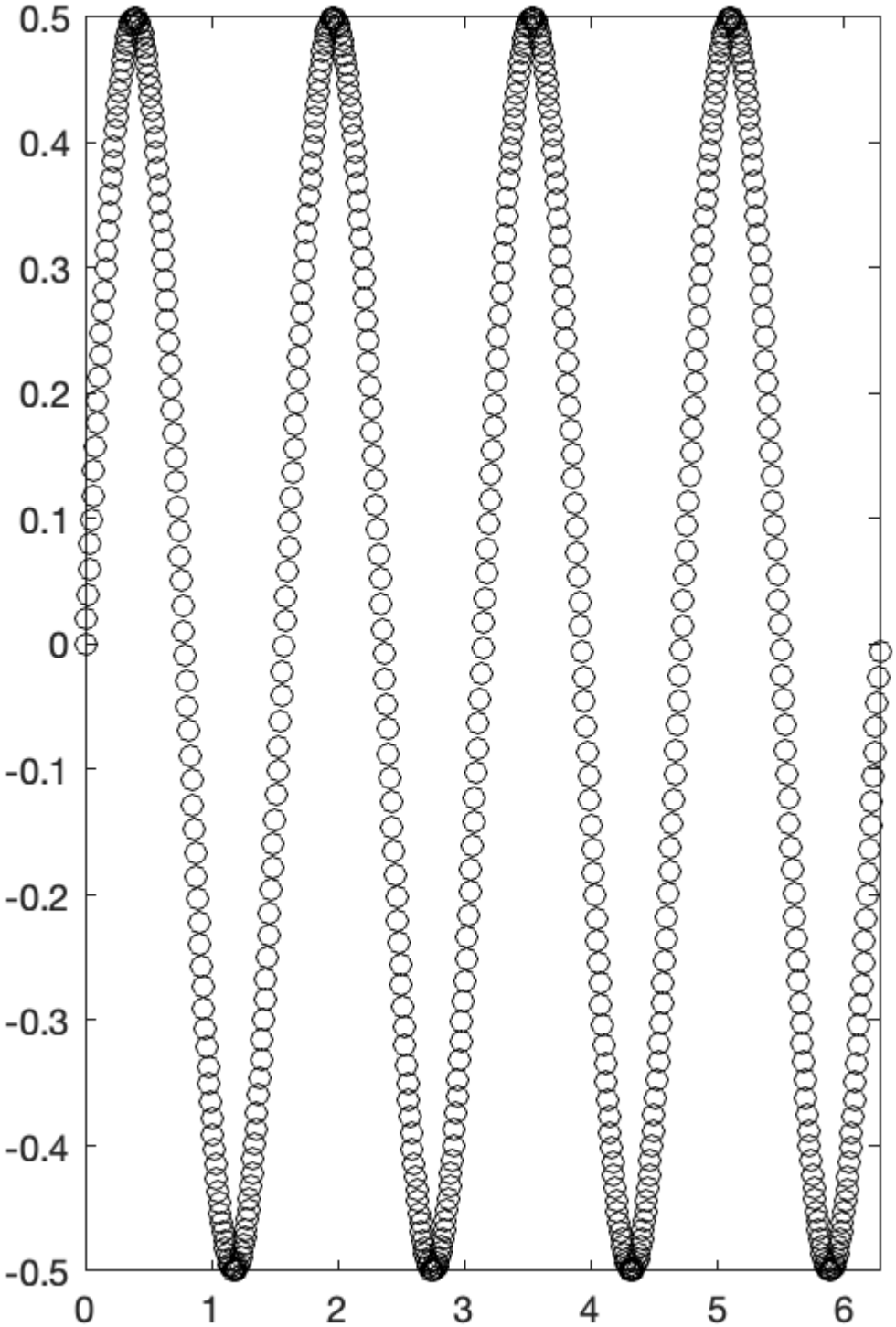
- **Scatter plots.** Like plot, `scatter()`require vectors of equal length. the default plot attributes are circular blue data markers.

```
x = 0:pi/20:2*pi;
y = sin(x);
scatter(x,y)
```

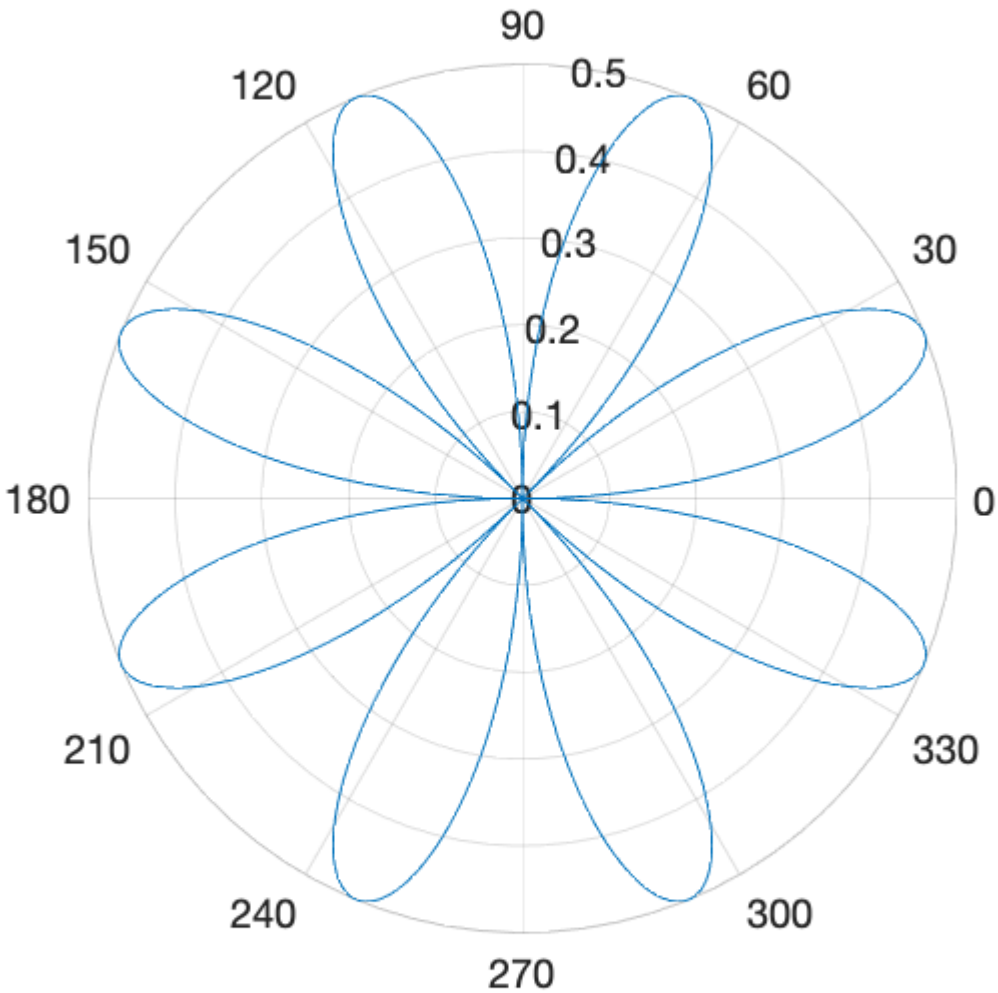

1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
0
1
2
3
4
5
6

- **Polarplot**

```
theta = 0:0.01:2*pi;
rho = sin(2*theta).*cos(2*theta);
% Plot theta vs. rho
plot(theta,rho,'ko')
```


0.5
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
-0.5
0
1
2
3
4
5
6

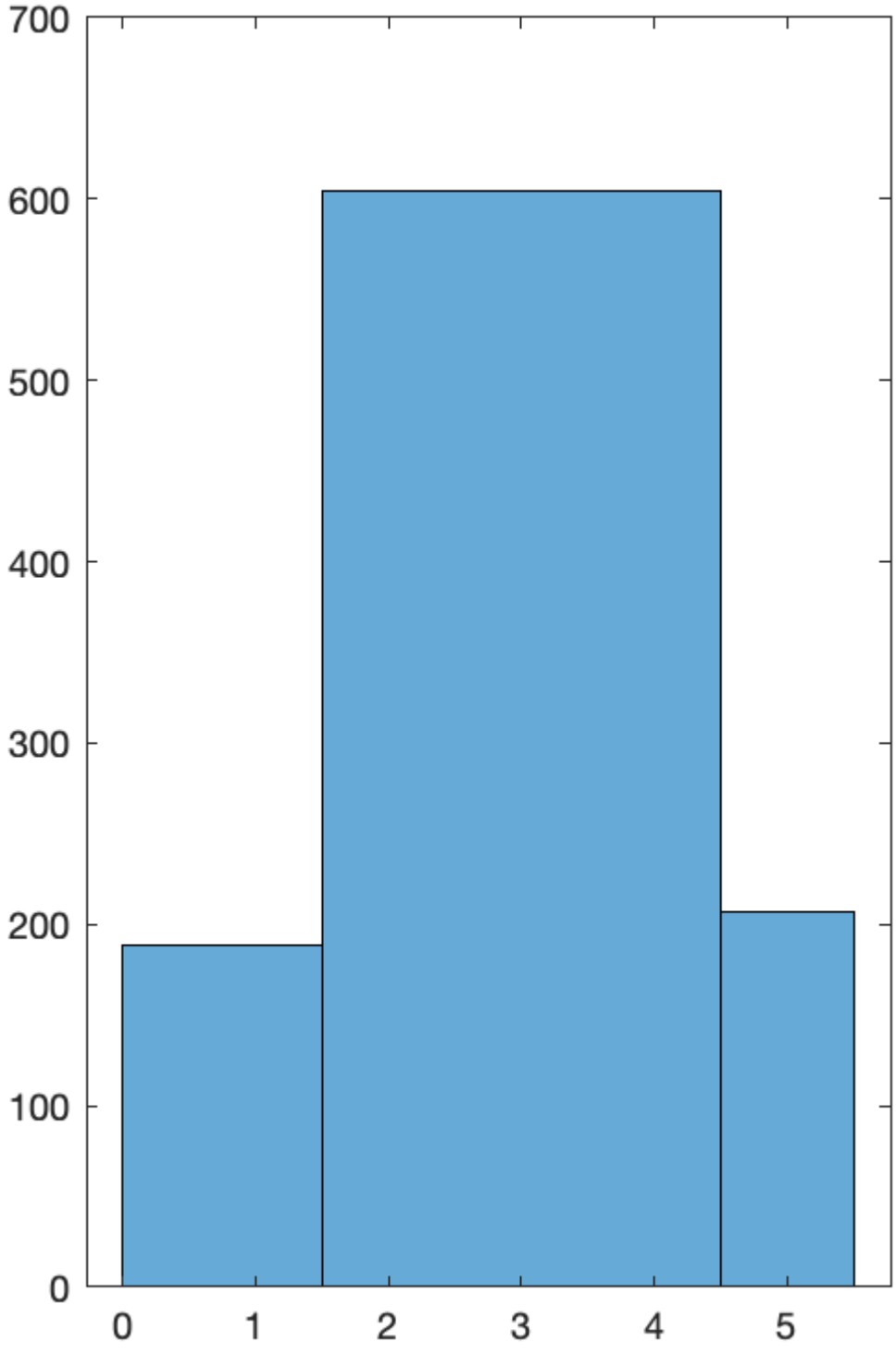
```
% Plot in polar space
polarplot(theta,rho)
```


90
120
60
150
30
180
0
210
330
240
300
270
0.5
0.4
0.3
0.2
0.1
0

Note: input is in units of radians.

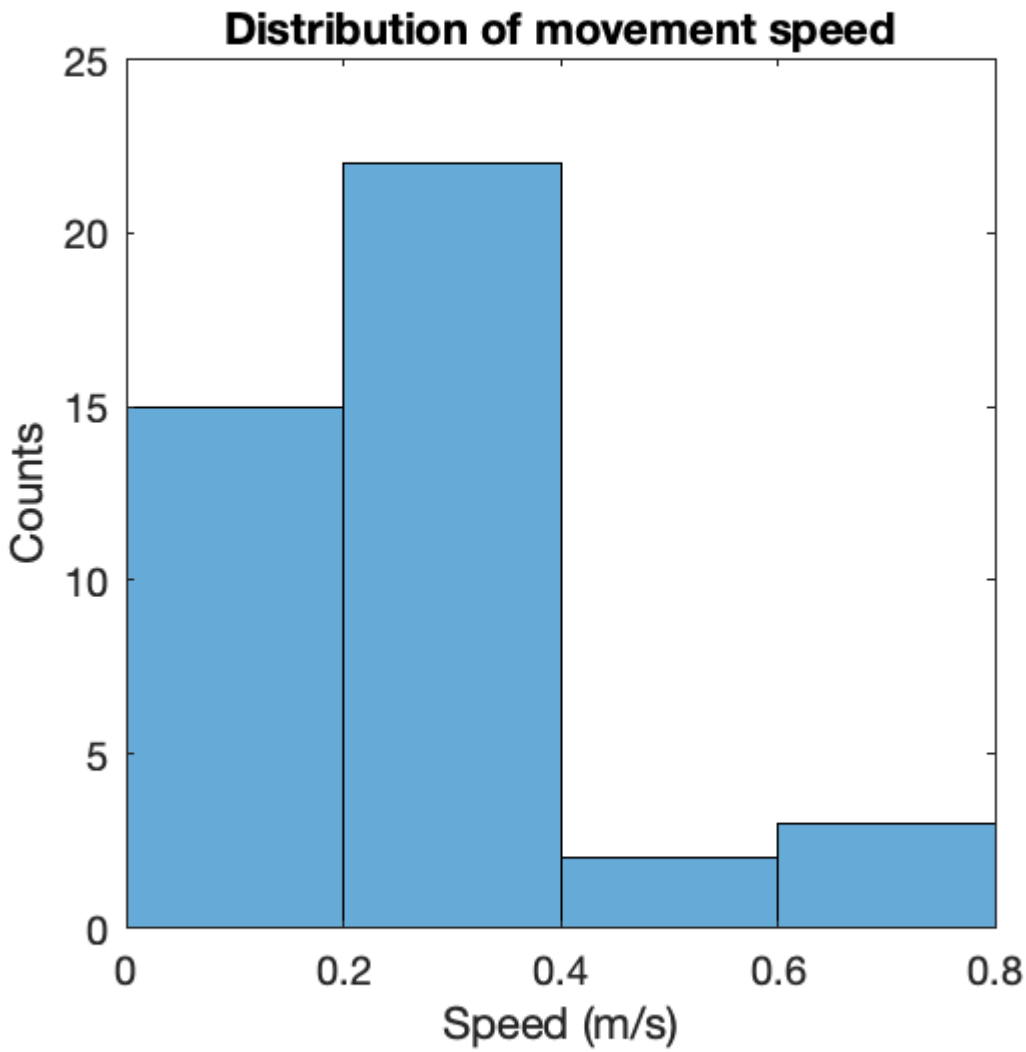
- **histogram.** Display counts in different bins

```
data = randi(5,[1,1000]);
h = histogram(data);
% Change bin sizes
set(h,'BinEdges',[0,1.5,4.5,5.5])
```


700
600
500
400
300
200
100
0
0
1
2
3
4
5

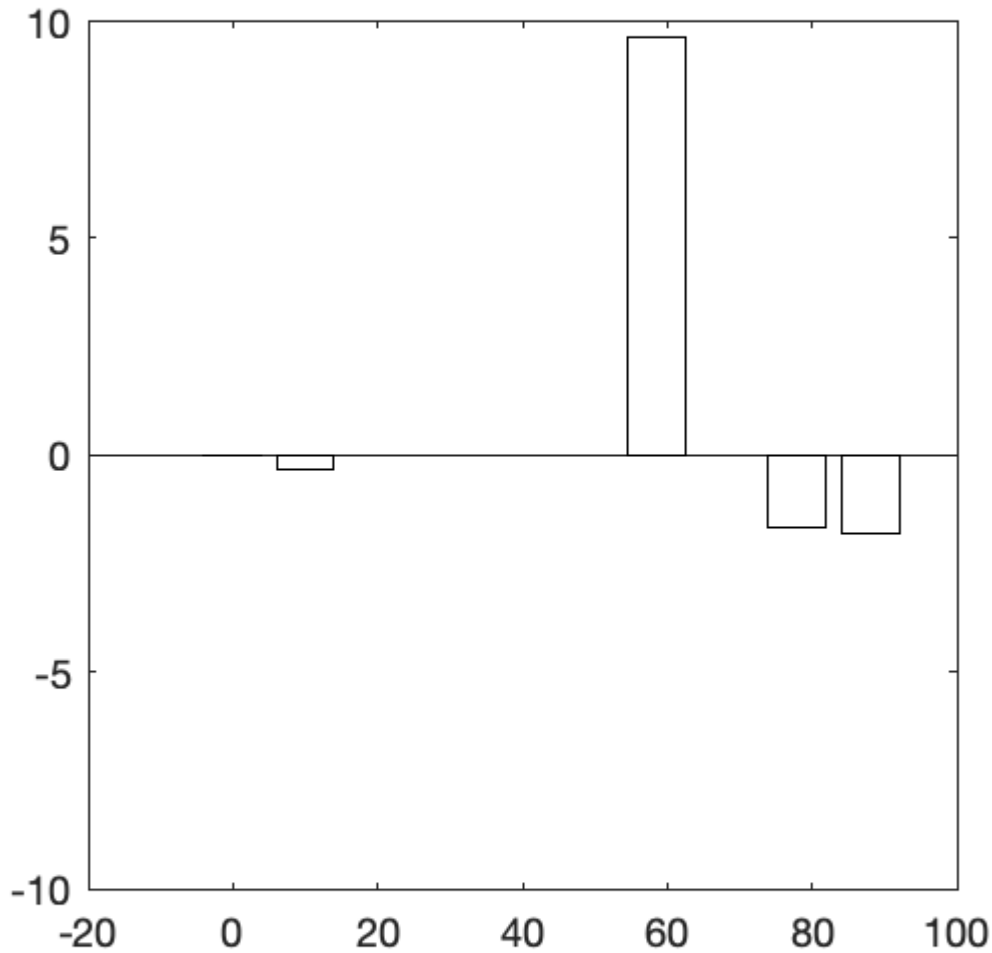
Back to our data. Create a histogram of speeds I traveled

```
histogram(myData(:,5));
axis square
xlabel('Speed (m/s)')
ylabel('Counts')
title('Distribution of movement speed')
```


Distribution of movement speed
Counts
Speed (m/s)
0
5
10
15
20
25
0
0.2
0.4
0.6
0.8

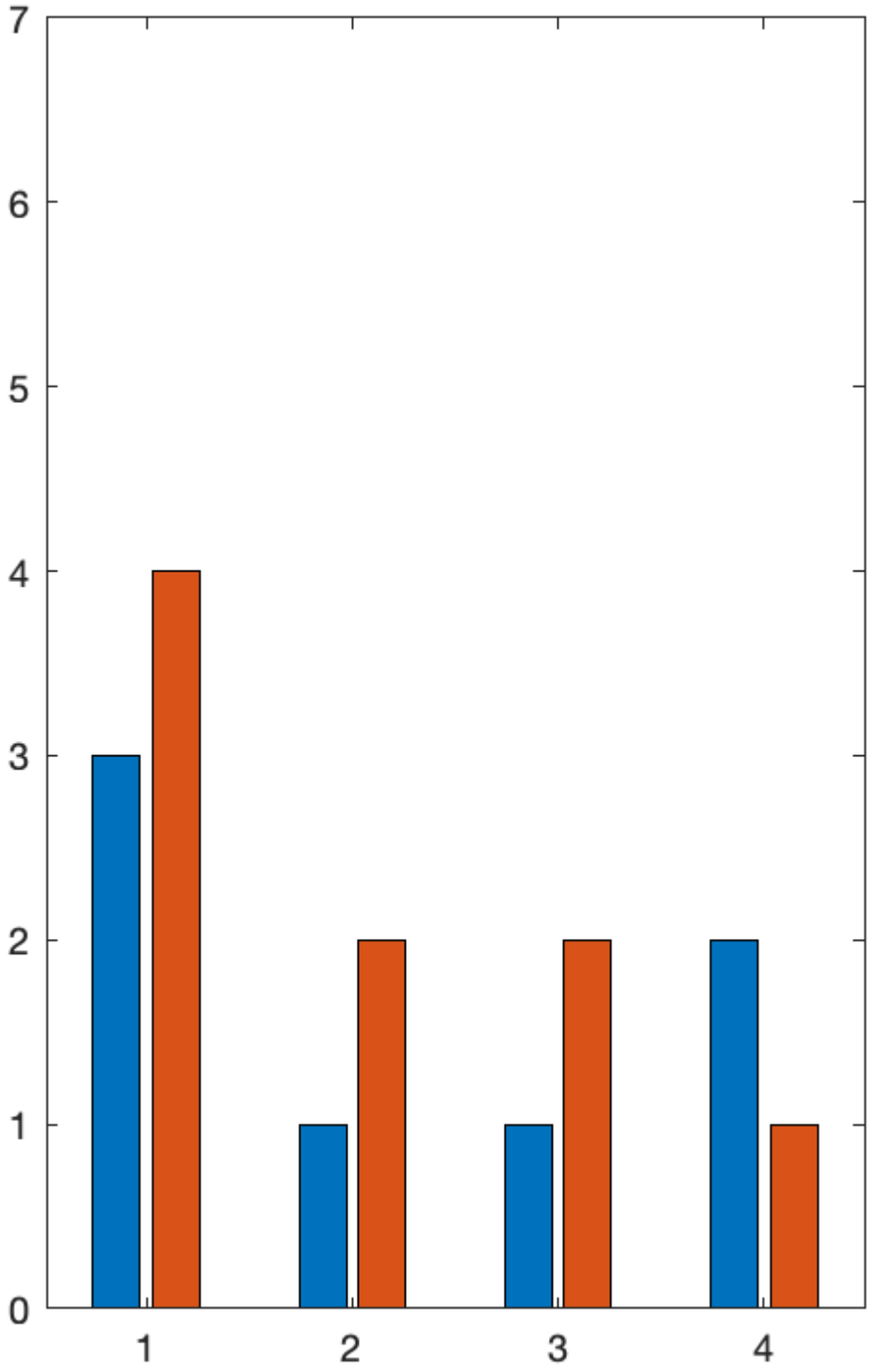
- **bar** plot

```
t = myData(1:10:end,1);
z = myData(1:10:end,4);

h = bar(t,z);
ylim([-10,10])
set(h,'FaceColor','none')
axis square
```


10
5
0
-5
-10
-20
0
20
40
60
80
100

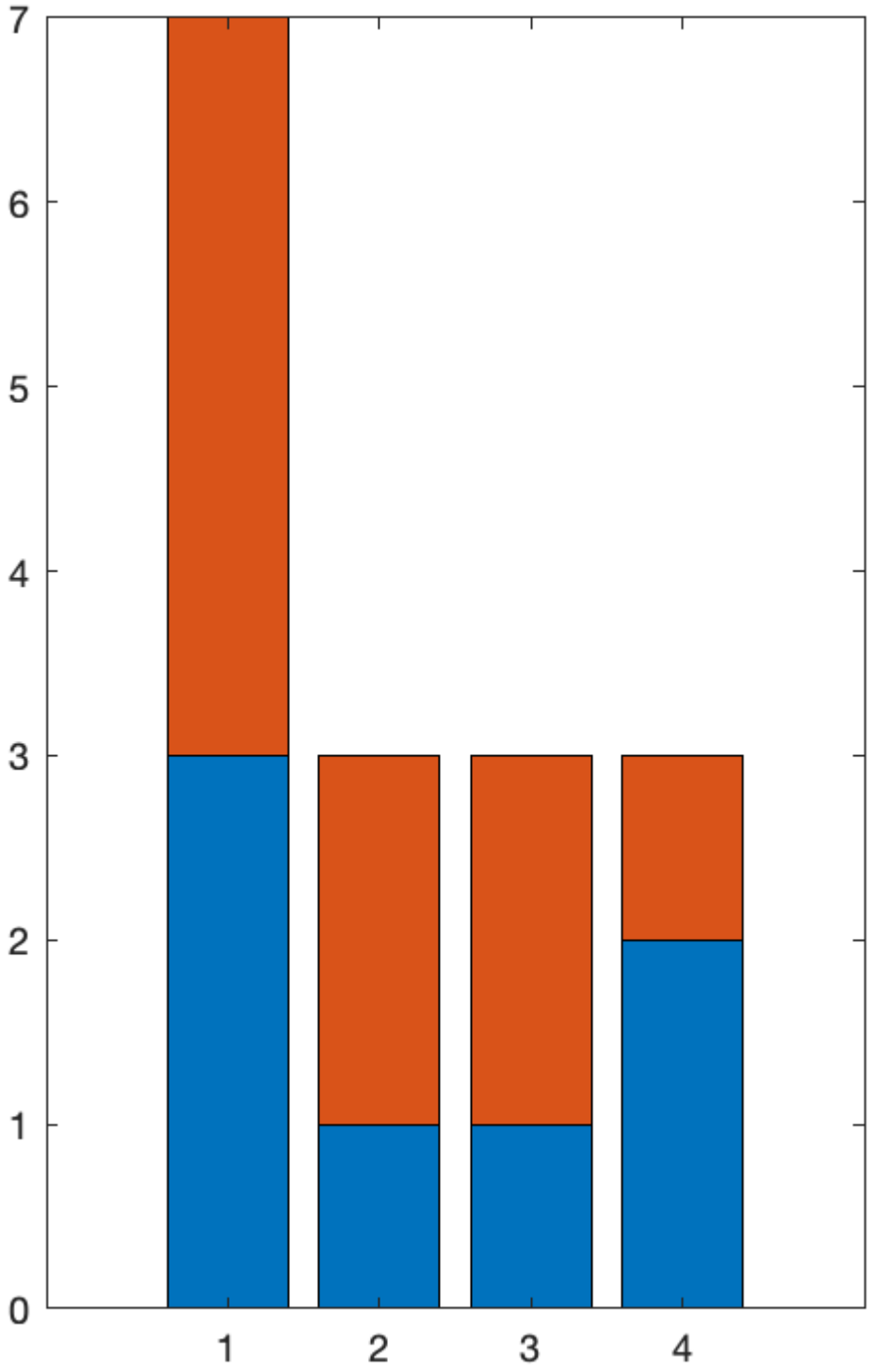
Grouped bar plots

```
x = 1:4;
y1 = randi(3,[1,4]);
y2 = randi(5,[1,4]);
% Note: dimensions of "Y" input
h = bar(x,[y1;y2]');
ylim([0,7])
```


7
6
5
4
3
2
1
0
1
2
3
4

Stacked bar plots

```
h = bar(x,[y1;y2]','stacked');
ylim([0,7])
```


7
6
5
4
3
2
1
0
1
2
3
4

Closing graphics objects -- SURPRISINGLY USEFUL

In an automated script, you may need to close figure windows. Figures may be closed using the `close` command

```
close: closes the last figure
close(n): closes figure number n
close all: closes all figures
```

Sometimes when my figures are not responding correctly, I can fix the issue by closing all figure instances. The issue may be related to an open figure in the background. Figures, axes, and plots may also be assigned to handles and the handle specified to **close** the object.

```
f = figure;
plot(rand(1,20))
close(f)
```

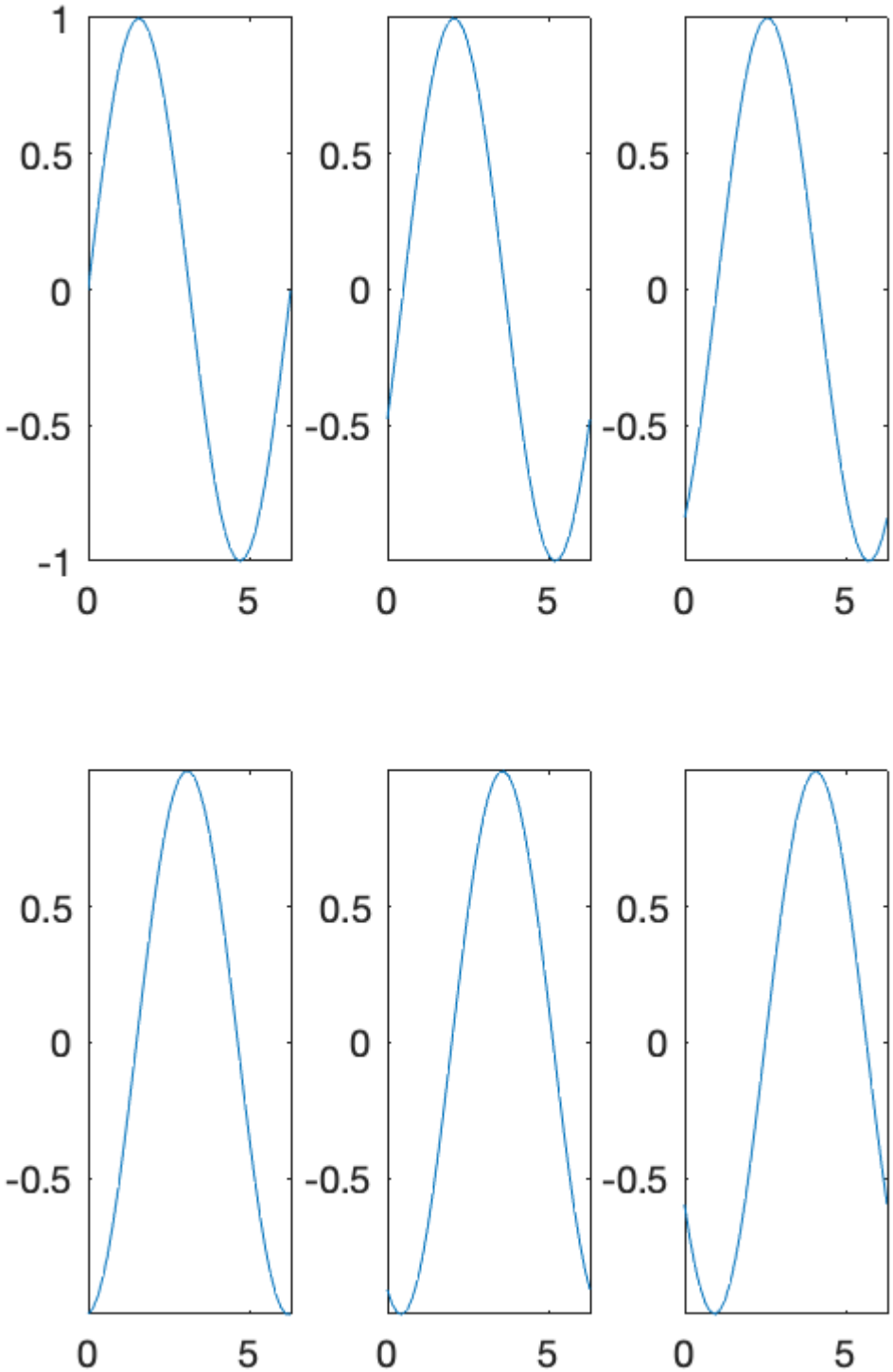
Multiple axes in a single figure window

Multiple axes may be assigned to a single figure using the `subplot` command. This command has the following syntax:

```
subplot(<number of rows>,<number of columns>,<current figure index>)
```

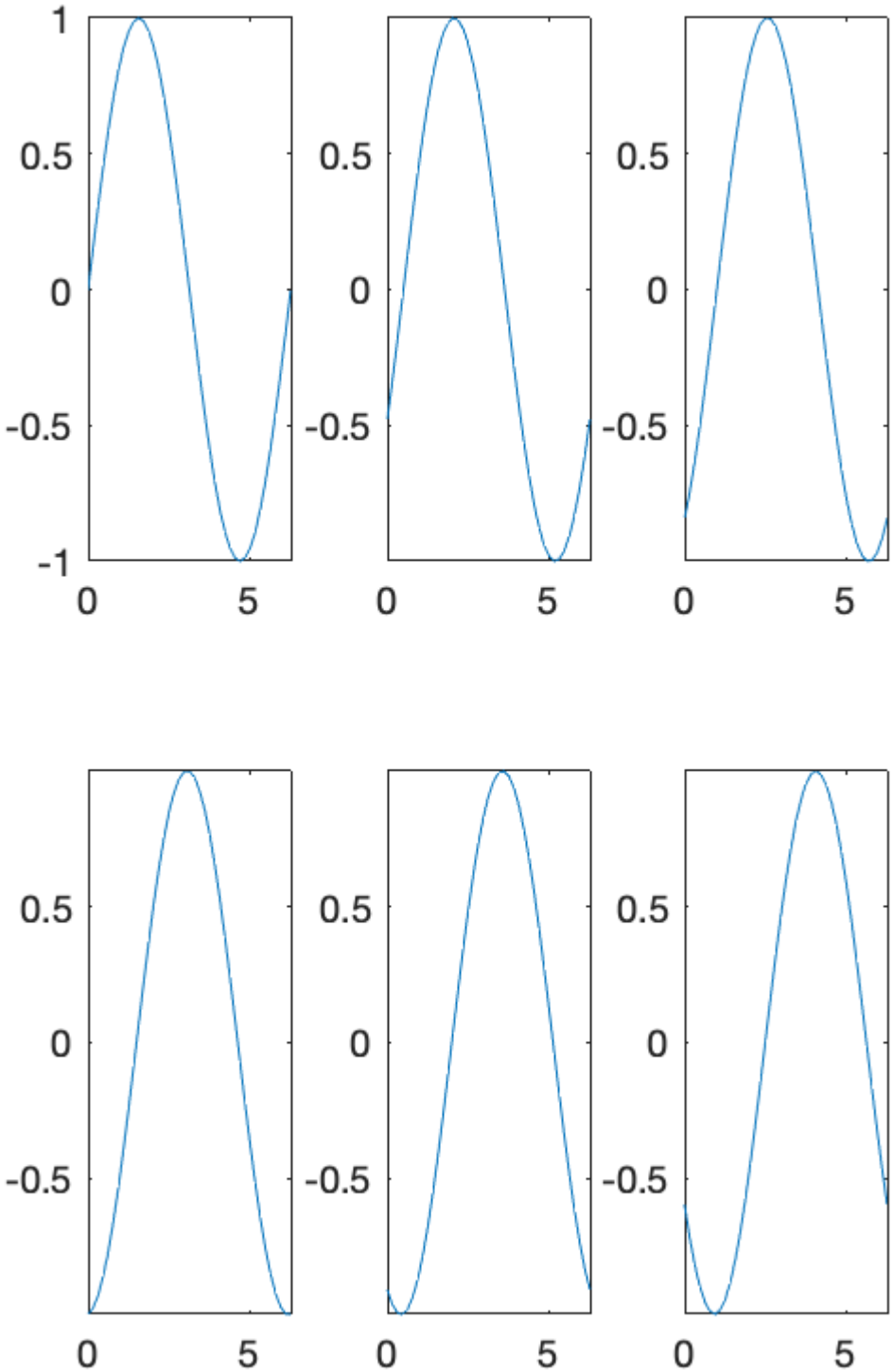
```
% X values
x = 0:pi/20:2*pi;

% Separate Y functions
y1 = sin(x-0);
y2 = sin(x-.5);
y3 = sin(x-1);
y4 = sin(x-1.5);
y5 = sin(x-2);
y6 = sin(x-2.5);

subplot(2,3,1)
plot(x,y1)
axis tight

subplot(2,3,2)
plot(x,y2)
axis tight

subplot(2,3,3)
plot(x,y3)
axis tight

subplot(2,3,4)
plot(x,y4)
```

```
axis tight

subplot(2,3,5)
plot(x,y5)
axis tight

subplot(2,3,6)
plot(x,y6)
axis tight
```


1
0.5
0
-0.5
-1
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5

Question. How can this code be made **clearer**? Use a `for` loop -- we know how many iterations to perform

```
% X values
x = 0:pi/20:2*pi;

% For each loop iteration, fill in one of the subplots
for k = 0:5
    y = sin(x-0.5*k);

    subplot(2,3,k+1)
    plot(x,y)
    axis tight
end
```


1
0.5
0
-0.5
-1
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5

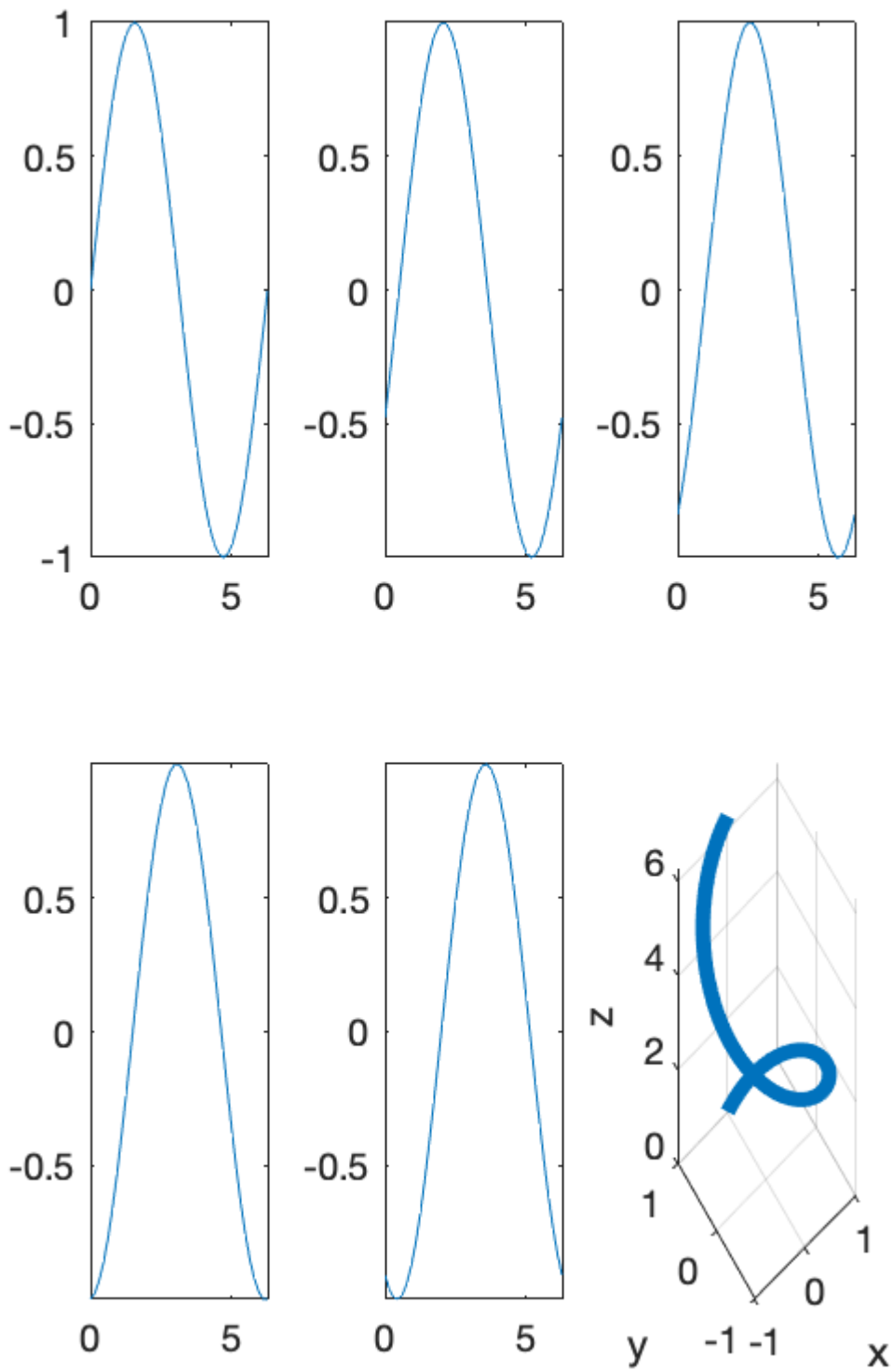
Plotting in 3D

All of the same concepts apply to producing 3D plots. In this case, the workhorse plot function is `plot3`.

`plot3`

plot3 takes in 3 inputs corresponding to x, y, and z positions

```
t = 0:pi/100:2*pi;
x = sin(t);
y = cos(t);
z = t;
plot3(x,y,z,'LineWidth',4)
xlabel('x')
ylabel('y')
zlabel('z')
grid on
```

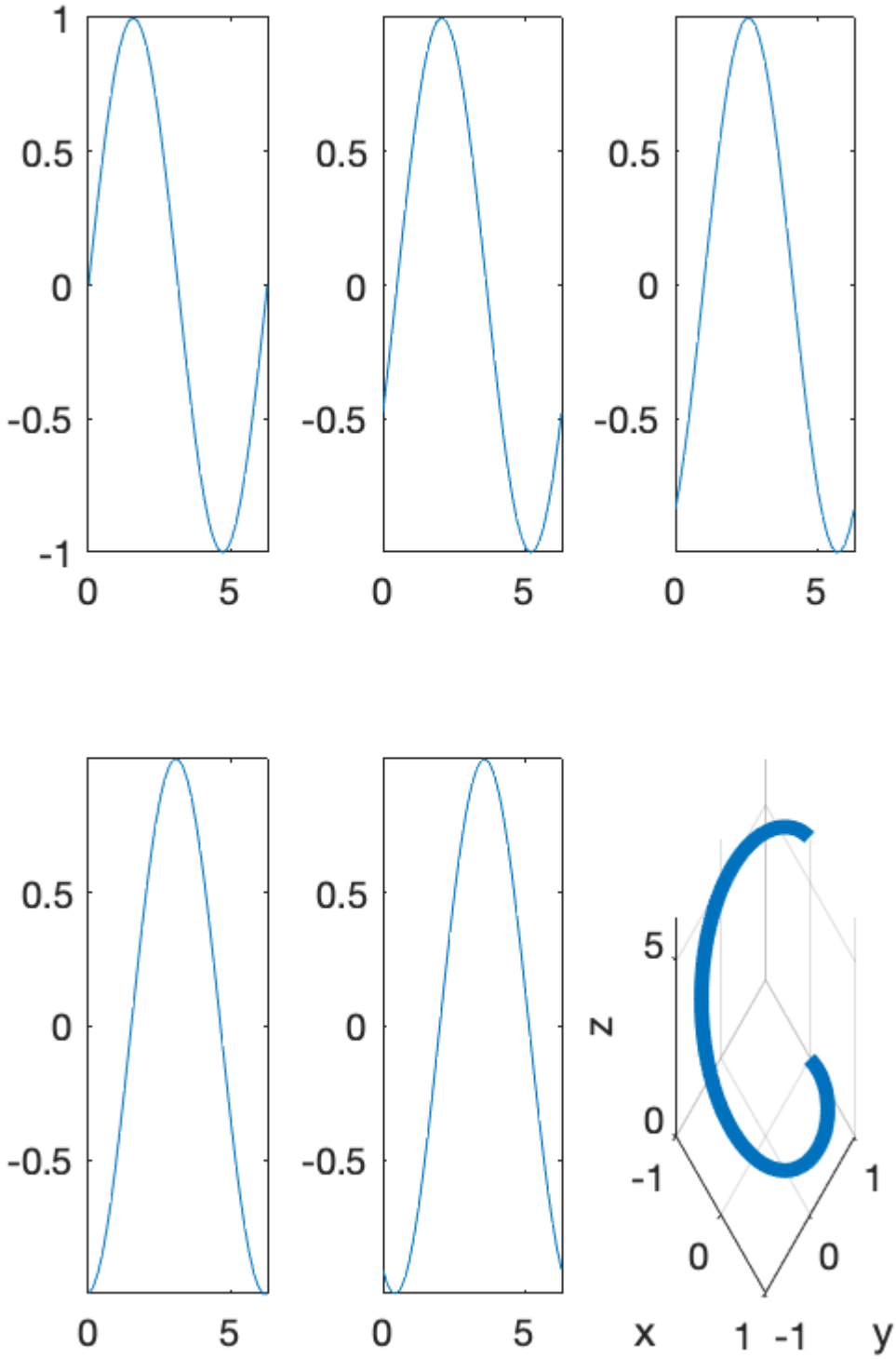

1
0.5
0
-0.5
-1
0
5
z
6
4
2
y
x
-1

Note: you can manually change the view point by clicking and dragging on the plot.

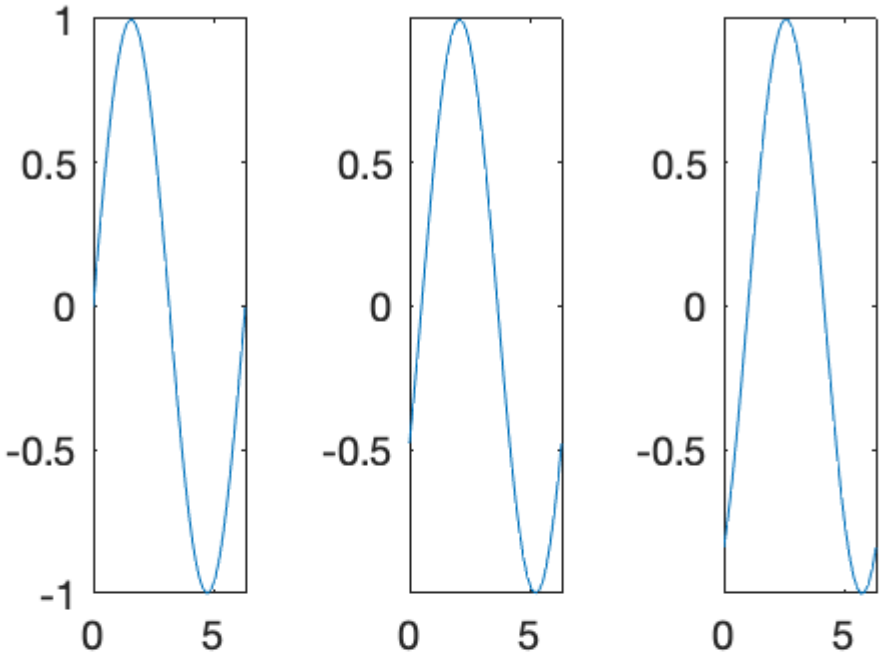
Important note: it is strongly recommended to specify a view point using `view`. This will help you precisely pick an angle to view.

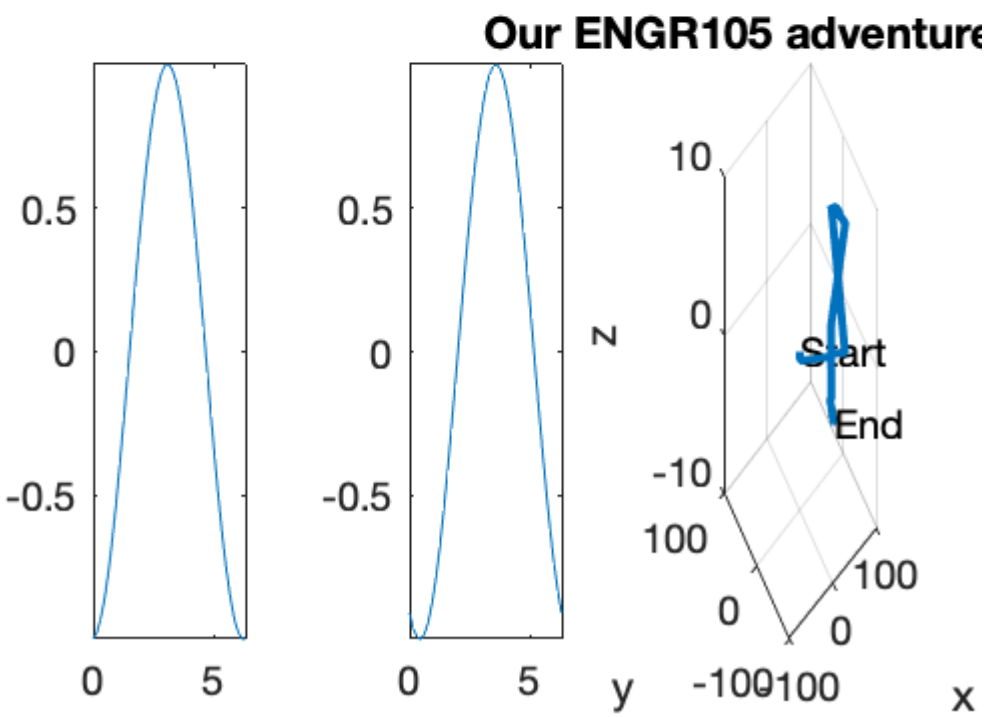
```
view(<azimuth angle>,<elevation angle>)
```

```
% Change the viewpoint
view(45,45)
```


1
0.5
0
-0.5
-1
0
5
z
x
y

```
plot3(myData(:,2),myData(:,3),myData(:,4),'LineWidth',2)
xlabel('x')
ylabel('y')
zlabel('z')
xlim([-100,100])
ylim([-100,100])
zlim([-10,10])
grid on
title('Our ENGR105 adventure in 3D')
text(myData(1,2),myData(1,3),myData(1,4),'Start')
text(myData(end,2),myData(end,3),myData(end,4),'End')
```


1
0.5
0
-0.5
-1
0
5

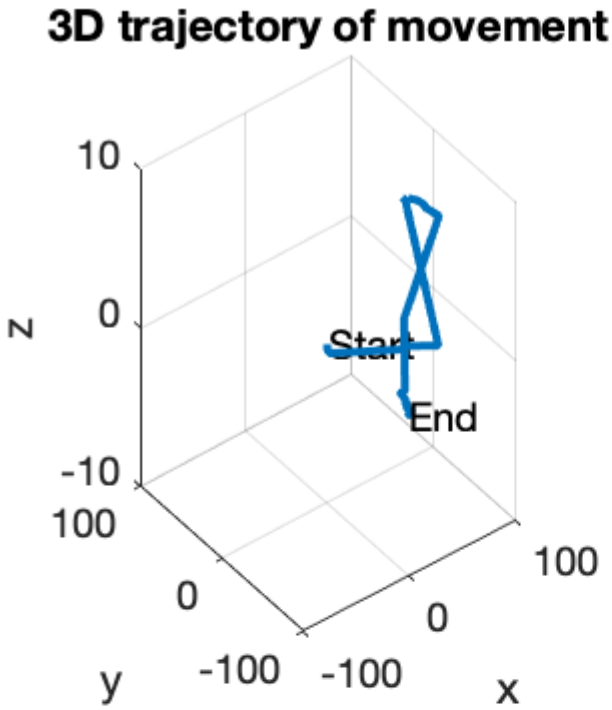

Our ENGR105 adventure
0.5
0
-0.5
0
5
10
0
-10
z
Start
End
100
0
-100
y
100
0
-100
x

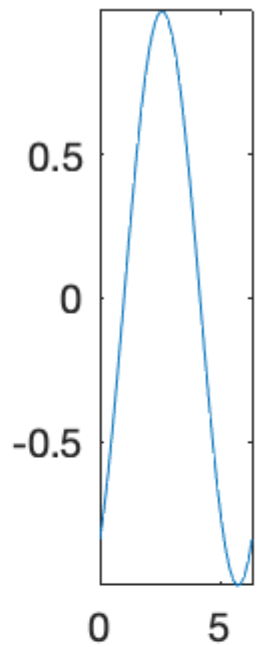
Practice Problem: adventure summary

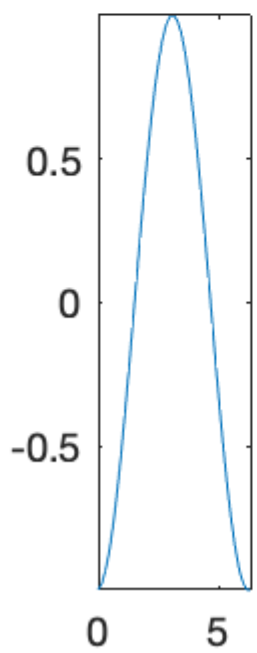
Produce one figure that summarizes the walk. That is, it should contain:

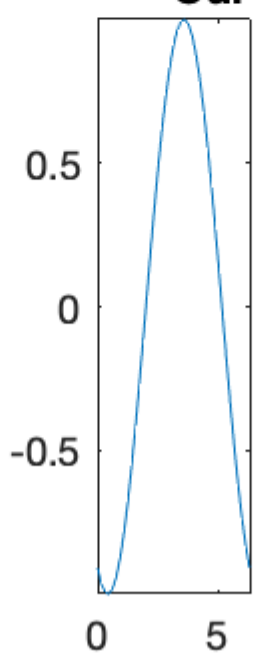
1. Travel in 3D space
2. x vs. y travel
3. altitude as a function of time
4. distribution of walking speeds

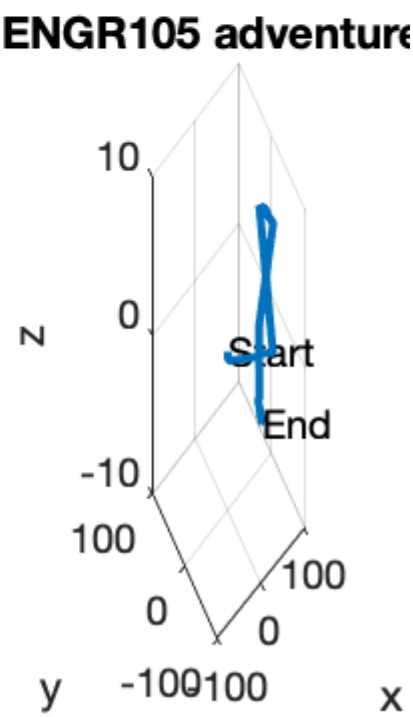
```
% Subplot 1: Travel in 3D
subplot(2,2,1)
plot3(myData(:,2), myData(:,3), myData(:,4), 'LineWidth', 2)
xlabel('x')
ylabel('y')
zlabel('z')
xlim([-100,100])
ylim([-100,100])
zlim([-10,10])
grid on
title('3D trajectory of movement')
text(myData(1,2), myData(1,3),myData(1,4),'Start')
text(myData(end,2), myData(end,3),myData(end,4),'End')
```


3D trajectory of movement
10
0
-10
z
Start
End
100
0
-100
y
100
0
-100
x

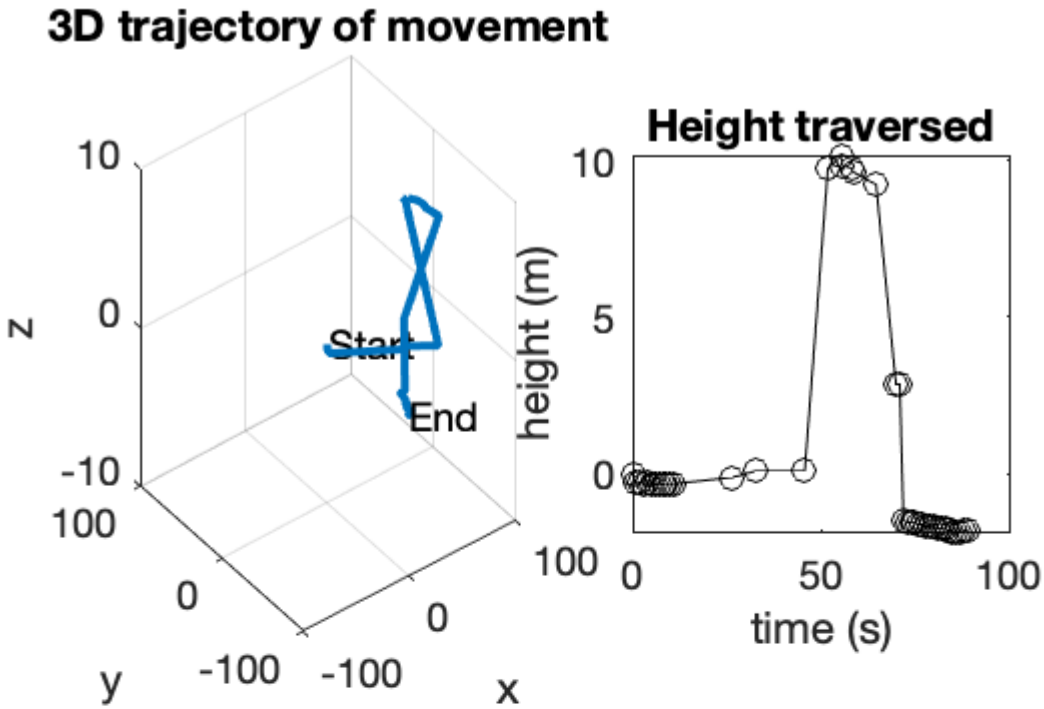

0.5
0
-0.5
0
5

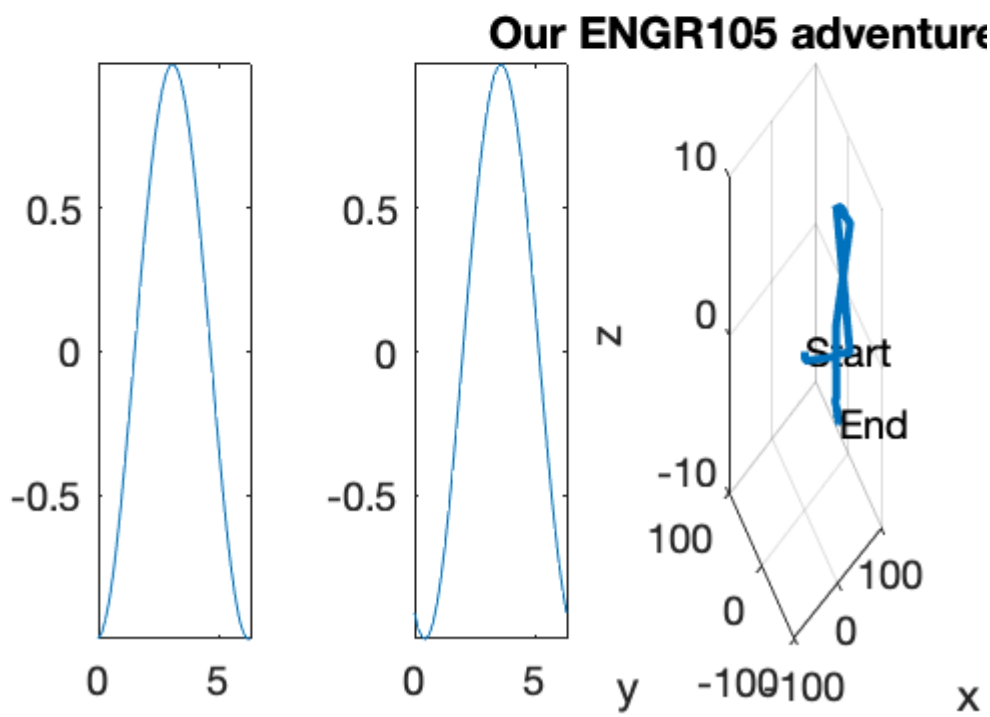

0.5
0
-0.5
0
5


0.5
0
-0.5
0
5

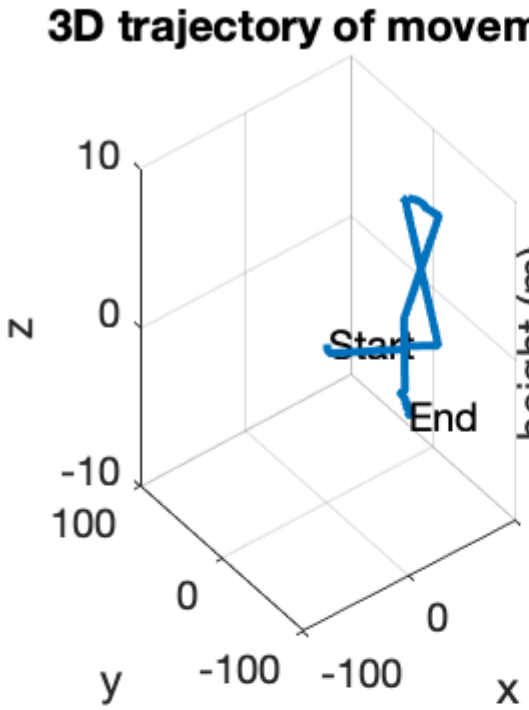

Our ENGR105 adventure
10
0
-10
z
Start
End
100
0
-100
y
100
0
-100
x

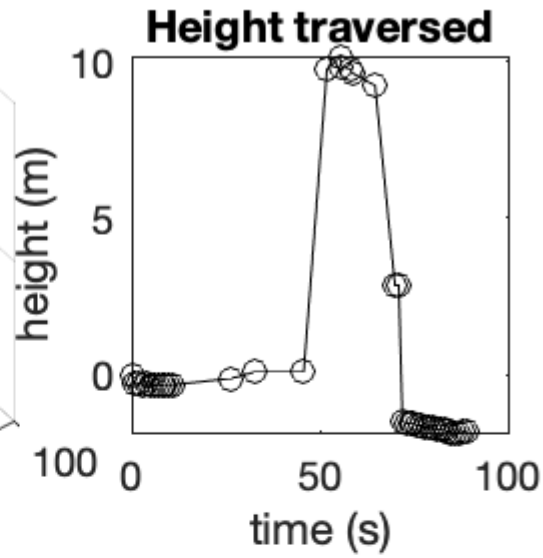
```
% Subplot 2: altitude as a function of time
subplot(2,2,2)
plot(myData(:,1), myData(:,4), 'ko-')
zlim([-10,10])
title('Height traversed')
xlabel('time (s)')
ylabel('height (m)')
axis square
```

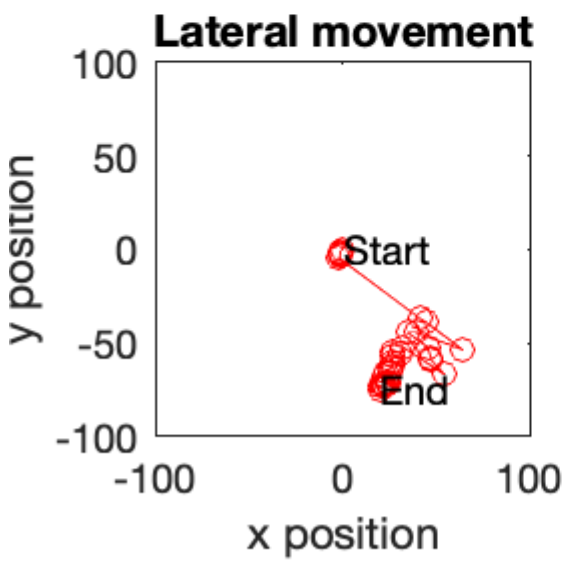

3D trajectory of movement
Start
End
z
y
x
10
0
-10
100
0
-100
-100
0
Height traversed
height (m)
time (s)
10
5
0
0
50
100

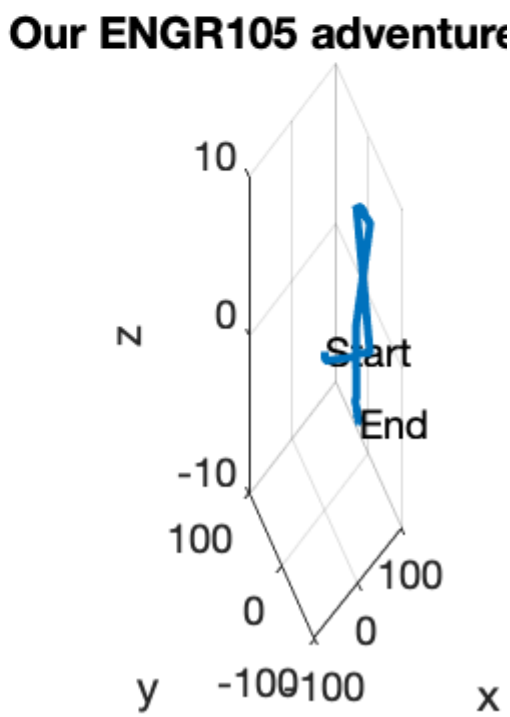

Our ENGR105 adventure
0.5
0
-0.5
0
5
0.5
0
-0.5
0
5
Start
End
z
y
x
10
0
-10
100
0
-100
100
0
-100

```
% Subplot 3: x vs y position
subplot(2,2,3)
plot(myData(:,2), myData(:,3), 'ro-')
xlabel('x position')
ylabel('y position')
xlim([-100,100])
ylim([-100,100])
axis square
title('Lateral movement')
text(myData(1,2), myData(1,3), 'Start')
text(myData(end,2), myData(end,3), 'End')
```


3D trajectory of movement
10
0
-10
z
Start
End
100
0
-100
y
-100
0
100
x

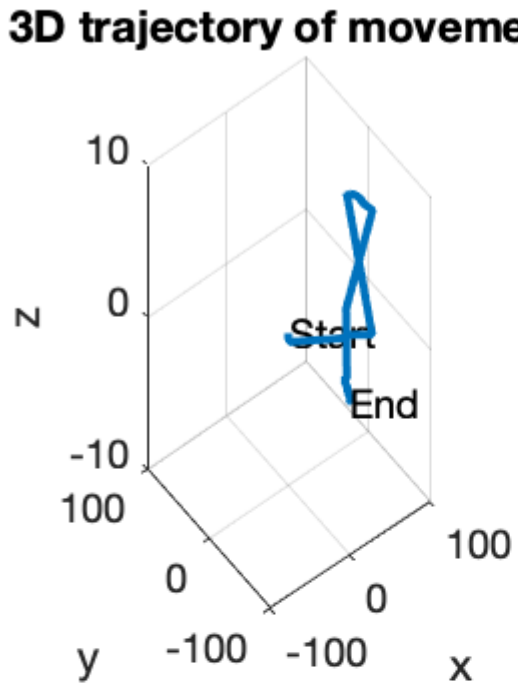

Height traversed
10
5
0
height (m)
0
50
100
time (s)

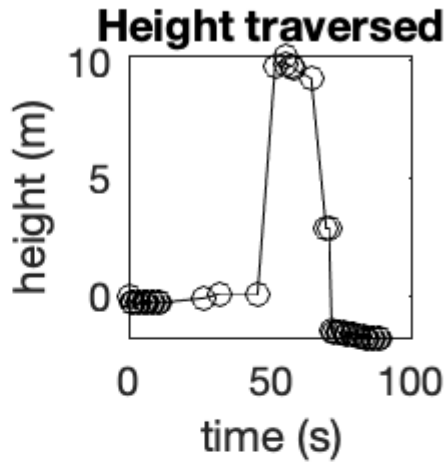

Lateral movement
100
50
0
-50
-100
y position
Start
End
-100
0
100
x position

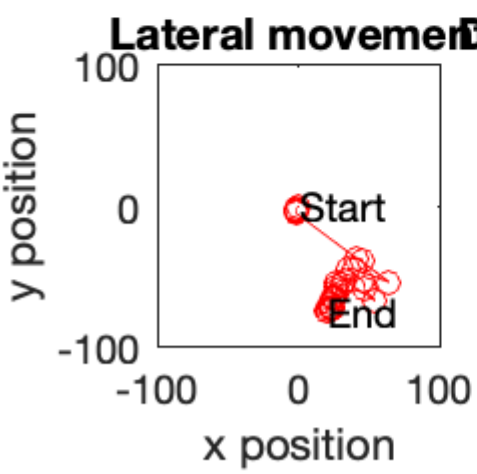

Our ENGR105 adventure
10
0
-10
z
Start
End
100
0
-100
y
-100
0
100
x

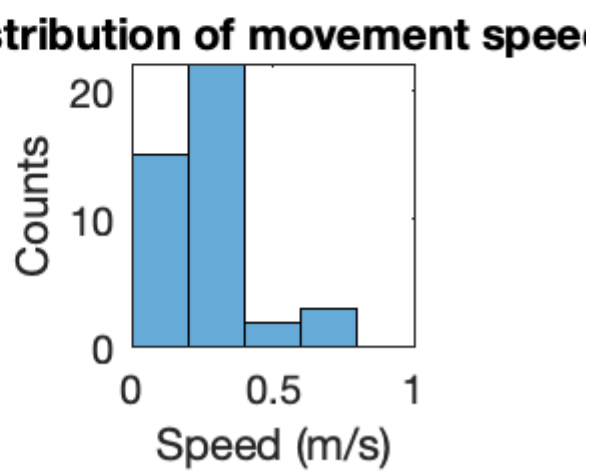
```
% Subplot 4: histogram of speeds
subplot(2,2,4)
histogram(myData(:,5));
axis square
xlabel('Speed (m/s)')
ylabel('Counts')
title('Distribution of movement speed')

% Add a title to the entire figure
sgtitle('ENGR105 adventure summary')
```

ENGR105 adventure summary









TOPIC 14: Data types in MATLAB

ENGR105, 10/20/20

Practice problem: Legendre polynomials and recursion

Legendre's differential equation, depicted below, is seen in many engineering topics including mechanics, fluid dynamics, thermodynamics, and electromagnetism.

$$(1-x^2)\frac{d^2}{dx^2}y - 2x\frac{d}{dx}y + n(n+1)y = 0$$

A differential equation of this form is convenient, because solutions can be given as **Legendre polynomials**. The general recurrence formula for the Legendre polynomial may be mathematically represented as the following.

$$P_n(x) = \frac{1}{n}\left[(2n-1)x\,P_{n-1}(x) - (n-1)\,P_{n-2}(x)\right]$$

The first two Legendre polynomials are defined as $P_0(x) = 1$ and $P_1(x) = x$.

a) Create a recursive function named `legendreP` that takes **two inputs**: 1) a vector containing positions at which to evaluate the legendre polynomial, and 2) `n`, a scalar (positive) integer specifying the degree of the Legendre polynomial, as defined in the above equation, to be determined. Your function should have **one output**: `pn`, a vector containing the specified Legendre polynomial. This function should be clear and stored in the space provided at the end of this LiveScript document.

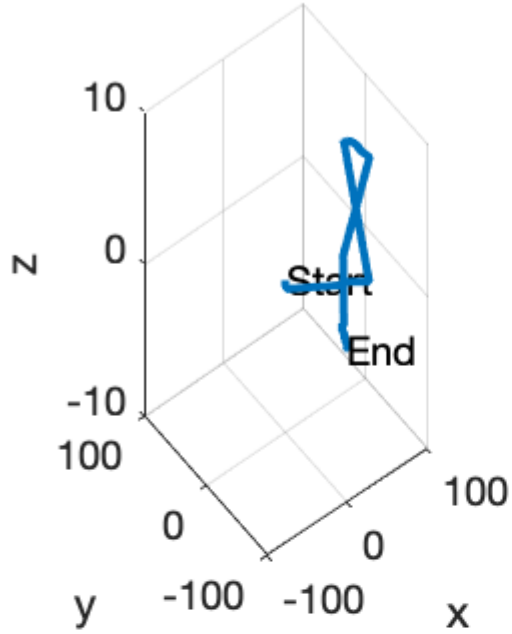
b) *Using your function*, produce one plot that contains $P_0(x)$, $P_1(x)$, $P_2(x)$, $P_3(x)$, $P_4(x)$ plotted over the domain $x \in [0, 1]$. As always, ensure that your plot is professional and clear.

```
% specify a range of x
x = 0:0.01:1;

%plot the lengendre polynomial
plot(x,legendreP(x,0),'k-', 'LineWidth', 2)
hold on
plot(x,legendreP(x,1),'r-', 'LineWidth', 1)
hold on
plot(x,legendreP(x,2),'k--', 'LineWidth', 2)
hold on
plot(x,legendreP(x,3),'r--', 'LineWidth', 1)
hold on
plot(x,legendreP(x,4),'b:', 'LineWidth', 2)
hold off
```

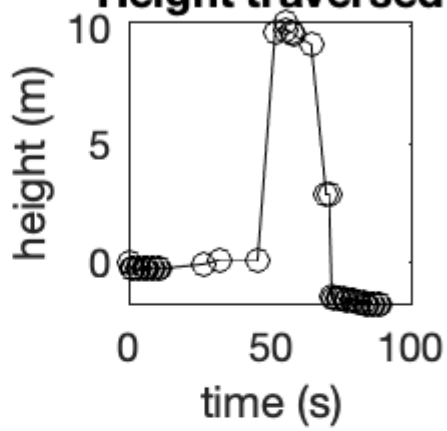
```
axis square
ylim([-1,1.5])
set(gca,'Color', [0.95, 0.95, 0.95])
xlim([0,1.3])
legend('P_0','P_1','P_2','P_3','P_4')
```

ENGR105 adventure summary

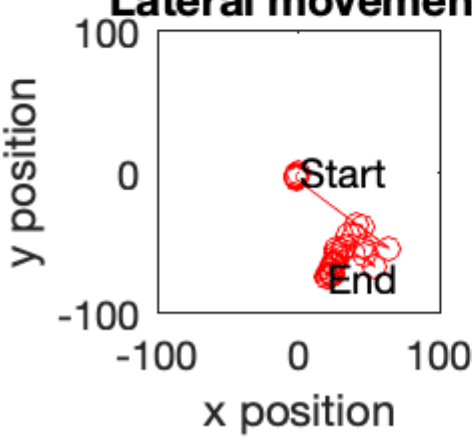
3D trajectory of movement

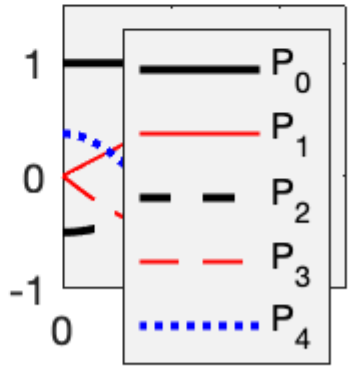


Height traversed



Lateral movement





Recall: different data types

In producing the above plot (and throughout this class), we have used many different *types* of variables. Last week, we saw that we could use `class` to give information about what **kind of information** is stored in a given variable.

```
class(x)
```

```
ans = 'double'
```

```
class('LineWidth')
```

```
ans = 'char'
```

```
class(legendreP(x,2))
```

```
ans = 'double'
```

Last week:

- `myData` is `timetable` class
- `myData.Timestamp` is `datetime` class
- `myData.speed` is `double` class
- `'hello world'` is `char` class

We can also use `whos` to give information, including the data type (or class) of a given variable.

```
whos x
```

```
  Name      Size            Bytes  Class     Attributes

  x         1x101             808  double
```

```
a = 2;
whos a
```

```
  Name      Size           Bytes  Class     Attributes

  a         1x1                8  double
```

In the above, variable a is class *double* requiring a memory of 8 *bytes*.

Data types (classes) in MATLAB

Useful MATLAB documentation:

- https://www.mathworks.com/help/matlab/matlab_prog/fundamental-matlab-classes.html

Bits, bytes, and memory

Remember that computation occurs at the **transistor scale**. Each of the memory locations in a computer is assoicated with multiple transistors and stores a finite sequence of **bi**nary dig**its** (**bits**).

- Typical storage sizes are **8**, **16**, **32** or **64 bits**
- A group of **8 bits** is referred to as a **byte**

Note: in the above example, a is associated with (8x8) **64 bits**.

Binary vs. base 10

Each bit may be `off` (**0**) or `on` (**1**). A eight-component line of binary storage can represent a number in base 10.

Storage element state	0	1	0	1	1	1	0	0
Base 2 to base 10 multiplier	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Base 10 representation	$0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0$							

Sums to 92

In the line of binary storage, the **position matters** just as in a base 10 number (e.g 9332). In the case of base 2 (binary), each individual digit is either 1 or 0, instead of 0-9 in base 10.

Note: the base 10 number is the sum of the binary base (2) raised to the corresponding 0 or 1.

Storage element state	0	0	0	0	0	0	0	0	All off Sum of 0
Base 10 numbers to be summed	$0*2^7$	$0*2^6$	$0*2^5$	$0*2^4$	$0*2^3$	$0*2^2$	$0*2^1$	$0*2^0$	

Storage element state	0	0	0	0	1	1	0	0	Some on Sum of 12
Base 10 numbers to be summed	$0*2^7$	$0*2^6$	$0*2^5$	$0*2^4$	$1*2^3$	$1*2^2$	$0*2^1$	$0*2^0$	

Storage element state	1	1	1	1	1	1	1	1	All on Sum of 255
Base 10 numbers to be summed	$1*2^7$	$1*2^6$	$1*2^5$	$1*2^4$	$1*2^3$	$1*2^2$	$1*2^1$	$1*2^0$	

That is, only `on` (1) bits are multiplied by the base 2 multiplier. The sum of the on (1) bits in base 10 is how our programs interact with memory.

Bit size and numbers

Bits enable a computer to represent a "number" using a **digital** infrastructure. **In the section just above, an 8-bit line of storage can be used to store an 8-bit integer.**

- That is, the integer can take any value between 0 and 2^8 - 1 (255).

In a computer, the bit system limits the **range of values** that can be stored and distinguished. Modern computers operate with **64 bits** while some older architectures and microcontrollers utilize 32 bits.

<u>Takeaway</u>: a **larger bit size** yields a larger range of base 10 numbers:

- 8 bit unsigned integers: 0 - 255
- 16 bit unsigned integers: 0 – 65535
- 32 bit unsigned integers: 0 – approx. 4 billion (or $2^{32} - 1$)
- 64 bit unsigned integers: 0 – approx. 20 quintillion (or $2^{64} - 1$)

In MATLAB, integers may be **signed** or **unsigned**

- **<u>unsigned</u>: <u>only</u> positive**
- **<u>signed</u>: positive <u>or</u> negative, though the range of potential options is reduced by two.**

```
% unsigned integers
uint8()
uint16()
uint32()
uint64()

% signed integers
int8()
int16()
int32()
int64()
```

Motivation: this will be very important when we talk about audio and image processing.

```
% Convert "numbers" to an integer
a = uint8(4)
```

```
a = uint8
    4
```

```
b = uint16(20)
```

```
b = uint16
    20
```

```
c = int32(200)
```

```
c = int32
    200
```

```
whos a b c
```

```
  Name      Size            Bytes  Class     Attributes
```

```
  a         1x1                 1  uint8
  b         1x1                 2  uint16
  c         1x1                 4  int32
```

Note: a stored number is clipped (overflow) if numbers larger or smaller than the bit size are stored

```
d = uint8(4000)
```

```
d = uint8
    255
```

```
% Note the range for "signed" integers
f = int8(200)
```

```
f = int8
    127
```

Thus: smaller bit size corresponds to a smaller range of potential integer values

Unrelated side question: where have we seen integers to be important? Indexing!

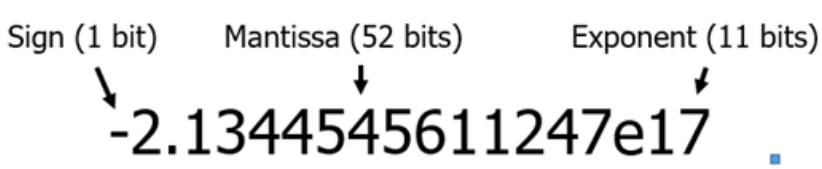
```
data = [1.5,3,4,2.2];
output = zeros(1,length(data));

% Goal: element-by-element, we want to square each element of data
for k = 1:length(data)
    output(k) = data(k)^2;
    disp(output)
end
```

```
    2.2500         0         0         0

    2.2500    9.0000         0         0

    2.2500    9.0000   16.0000         0

    2.2500    9.0000   16.0000    4.8400
```

MATLAB defaults to `double` precision

We often are not interested only in integers but in decimal values. To represent these, MATLAB defaults to the most accurate storage bit length available in the program – **64 bits (aka "`double` precision")**. This accuracy **costs storage space** – you can only store so many numbers given your RAM.

- Assignment of bits to representation of a number in 64 bits in accordance with standard IEEE 754



You can query the largest and smallest double precision numbers in MATLAB by invoking `realmax` and `realmin`.

```
realmax
```

```
ans = 1.7977e+308
```

```
realmin
```

```
ans = 2.2251e-308
```

Note: in the above, "e17" is equivalent to $\times 10^{17}$

Note 2: double precision can be overridden to economize memory use and (sometimes) computation time

Characters arrays and strings

Character arrays. We have worked a lot with **character arrays**, which can be read as text/symbols

```
plot(x,legendreP(x,0),'k--','LineWidth',2)
```

Just like with number, function handles, and graphic handles, character arrays can be assigned to variables.

```
a = 'k--';
b = 'LineWidth';
c = 'This is a character array';
disp(c)
```

```
This is a character array
```

Character arrays are distinguished by single quotes (`'`).

Strings. String arrays are created using double quotes (").

```
s1 = "This is a string";
disp(s1)
```

```
This is a string
```

Similarity: both can be used to store text information

Difference:

- Each character (including spaces and punctuation) in a character arrays corresponds to an index position in a "vector" or array.
- An entire string corresponds to one index position

```
% Index the above character array
c(1)
```

```
ans = 'T'
```

```
c(5:10)
```

```
ans = ' is a '
```

```
c(1:2:end)
```

```
ans = 'Ti sacaatrary'
```

```
% Use vector operations to determine the number of characters (including
% spaces) in a piece of text.
length(c)
```

```
ans = 25
```

```
% Use indexing on string. NOTE: only ONE element of s1
s1(1)
```

```
ans = "This is a string"
```

Thus: character arrays enable you to retrieve and manipulate each character in a piece of text, while strings treat the entire "block" of text as one unit.

Cell arrays

Cell arrays are arrays with indices that can hold a variety of data types or sizes - **all in the same variable**.

Example:

```
C = {'it is Tuesday', [1 5 9], [2 3 1; 5 9 7], ...
    uint8([1 4]), [.5; .1; .9], {'hello',[1 5]}}
```

```
C = 1×6 cell array
    {'it is Tuesday'}    {1×3 double}    {2×3 double}    {1×2 uint8}    {3×1 double}
{1×2 cell}
```

In this case, we even place a `cell` in a `cell`.

Indexing cell arrays. Cells are indexed much like arrays, except "`()`" **references a cell container** and "`{}`" references the **contents of the cells**

```
C = {'it is Tuesday', [1 5 9], [2 3 1; 5 9 7], ...
```

```
    uint8([1 4]), [.5; .1; .9], {'hello',[1 5]}}
```

```
C = 1×6 cell array
    {'it is Tuesday'}    {1×3 double}    {2×3 double}    {1×2 uint8}    {3×1 double}
{1×2 cell}
```

```
% Reference the cell container
C(1)
```

```
ans = 1×1 cell array
    {'it is Tuesday'}
```

```
class(C(1))
```

```
ans = 'cell'
```

```
% Reference the contents of a particular cell
C{1}
```

```
ans = 'it is Tuesday'
```

```
class(C{1})
```

```
ans = 'char'
```

That is:

- Individual cells can be extracted as full cell units using `()`
- Cell contents can be extracted using `{}`

<u>Important</u>: to write to a cell, you must write to the "contents" of a particular cell

Creating cell arrays. Cell arrays in productive code are often built within loops and are useful for storing numeric/character arrays of mixed sizes and/or mixed data types

```
for jj = 1:3
    myCell{jj} = rand(randi(5),randi(5));
end
```

Note: vector and array operations work on cells too:

```
length(myCell)
```

```
ans = 3
```

```
size(myCell)
```

```
ans = 1×2
     1     3
```

```
myCell{1}
```

```
ans = 4×2
    0.8072    0.4328
    0.4914    0.2072
    0.8769    0.3132
    0.5737    0.7182
```

Growing cell arrays. `end + 1` indexing can also be used to grow a cell arrays

```
tic % initialize timer

myCell = cell(0); % create empty cell

% Keep filling cell array with random matrices
% until .25 secs have passed
while toc < .01
    myCell{end+1} = rand(randi(5),randi(5));
end
size(myCell)
```

```
ans = 1×2
           1        1894
```

<u>Note</u>: use `cell(0)` to declare an empty cell array before the loop.

Shrinking cell arrays. Cells can be deleted by setting a particular element to `[]`.

```
for jj = 1:100
    myCell(jj) = [];
end
size(myCell)
```

```
ans = 1×2
           1        1794
```

```
myCell{1}
```

```
ans = 5×4
    0.4347    0.3987    0.8685    0.4439
    0.0850    0.2961    0.6257    0.0953
    0.6032    0.2887    0.5583    0.3038
    0.7470    0.8574    0.2847    0.3740
    0.3474    0.0337    0.6995    0.8692
```

Note: this is different from deleting the contents of individual cells.

```
for jj = 1:100
    myCell{jj} = [];
end
size(myCell)
```

```
ans = 1×2
            1         1794
```

```
myCell{1}
```

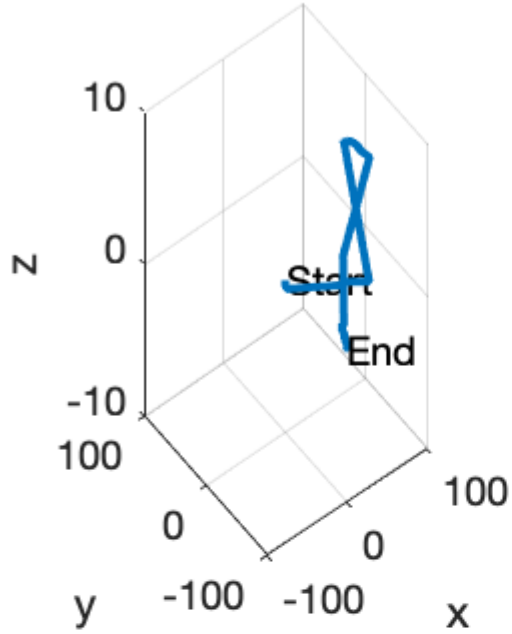
```
ans =

     []
```

Back to the practice problem: making more efficient

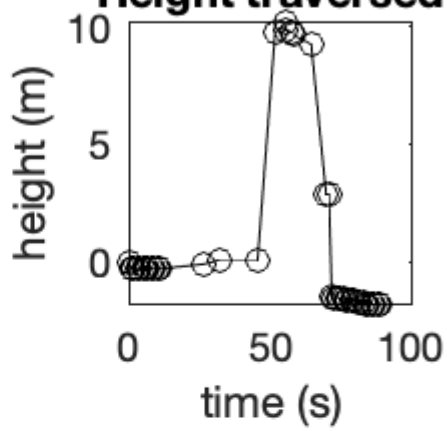
```
%Specify a range of x
x = 0:0.01:1;

%Cell array to store line specifications
Str = {'k-', 'r-', 'k--', 'r--', 'b:'};

for k = 0:4
    %Plot legrendre polynomial
    plot(x,legendreP(x,k),Str{k+1},'LineWidth',mod(k+1,2)+1)
    hold on
end
hold off


% Graphical parameters
axis square
ylim([-1,1.5])
set(gca,'Color', [0.95, 0.95, 0.95])
xlim([0,1])
legend('P_0','P_1','P_2','P_3','P_4','Location','eastoutside')
```

ENGR105 adventure summary

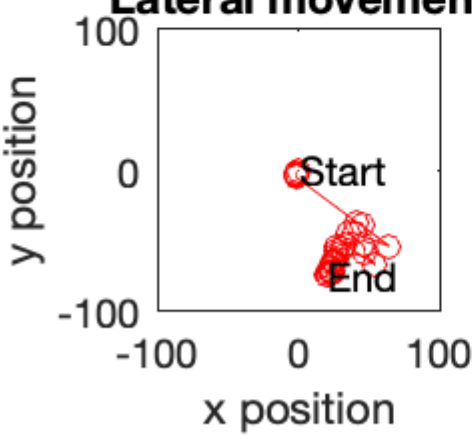
3D trajectory of movement

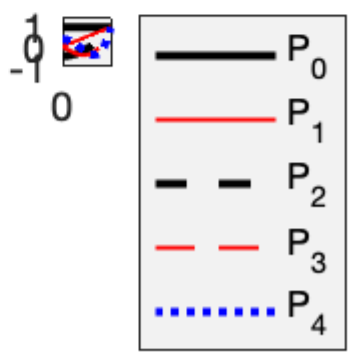


Height traversed



Lateral movement





Functions stored at the end of the document.

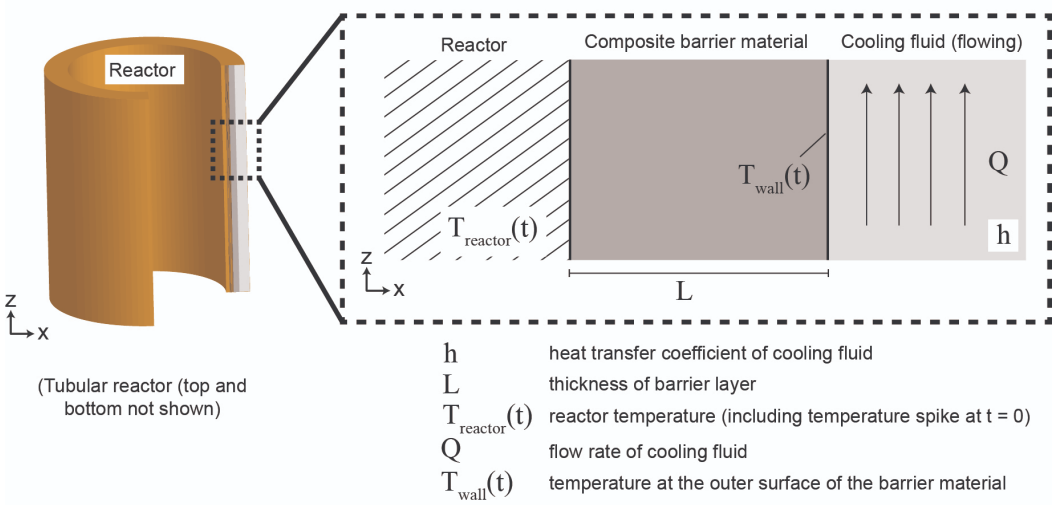
Practice problem: Legendre polynomial

TOPIC 15: Working with text

ENGR105, 10/22/20

Practice problem: reactor cooling

Consider a reactor (e.g. a bioreactor, chemical reactor, nuclear reactor, etc.) that, due to some internal perturbation**, experiences a temporary increase in temperature right before `t = 0`**. The reactor itself is designed such that it is encased by a "barrier material" that serves to protect the wall of the reactor from a potentially caustic cooling fluid. This constantly flowing low temperature cooling fluid serves as a heat sink that should (hopefully!) keep the reactor at a sufficiently low temperature for the reaction to proceed safely and efficiently



Assume you have already applied your knowledge of heat transport to quantitatively describe this system. **You have produced the following equation** that describes `T_wall`, the temperature of the outer surface of the barrier layer as a function of time, where the **time-dependency** (*i.e.* the shape of the curve) **is influenced by the heat transfer coefficient of the cooling fluid,** as well as many other system parameters.

```
% Equation describing the wall temperature as a function of t and h
T_wall = 50*(1./(t+1) + 5*(t+1).^(-4*h)) + 300;
```

Issue: `T_wall` (defined above) increases in temperature to above <u>400 degrees</u> following the perturbation in the reactor. If `T_wall` remains greater than `400C` for more than <u>some important length of time</u> (*e.g.* 30 minutes), there is a significant risk of barrier layer degradation or some other adverse outcome.

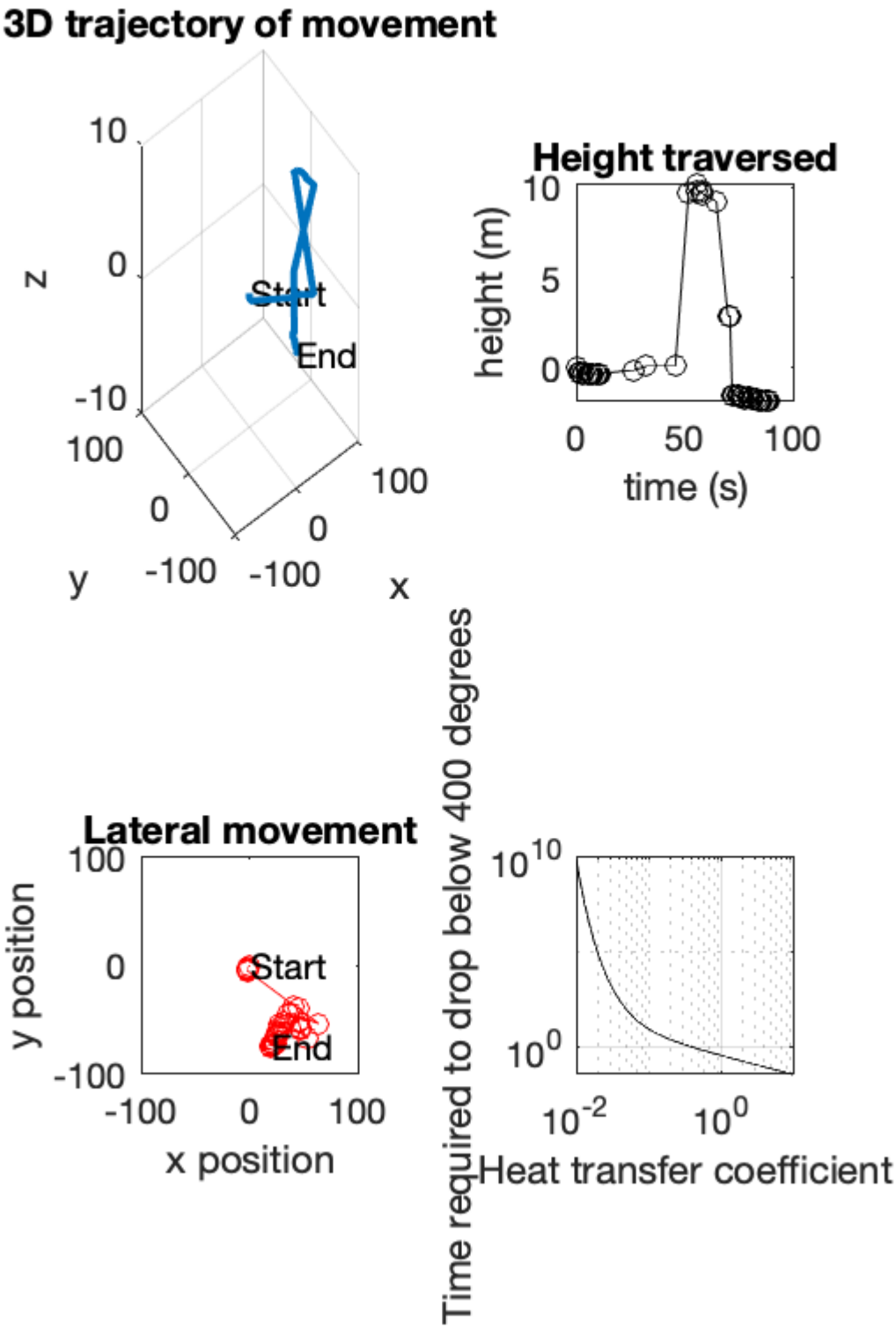
Your task as an engineer: produce a MATLAB script that enables you to visualize how **the time required for `T_wall` to drop below 400C** (in the system described above) **depends on the cooling fluid heat transfer coefficient, `h`.**

```
% Step 1: Plot the raw data -> can do so by plotting T_wall vs time
%create a vector for time
t = 0:0.01:50;

Twall = @(h) 50*(1./(t+1) + 5*(t+1).^(-4*h)) + 300;

plot(t,Twall(1), 'k-')
axis square
xlabel('time (minutes)')
ylabel('Temperature (degrees C)')
ylim([350,600])
%Step 2: specify ranges of h
%"relevant" -> determined from our system
h = logspace(-2, 1, 200);

%preallocate the vector
t400 = zeros(1,length(h))
```

```
t400 = 1×200
     0     0     0     0     0     0     0     0     0     0     0     0
 0     0     0     0     0     0     0     0     0     0     0     0
 0     0     0     0     0     0
```

```
%Step 3: For each h value, solve for the cooling time
for k = 1:length(h)
   %Step 4: Store cooling time in a vector so that we can produce a plot
   t400(k) = newtonSolve(@(x) quiz3(x,h(k)),0,10^-4);
end

% Step 5: Produce a plot to visualize how t400 depends on the heat transfer
% coefficient
plot(h,t400,'k-')
%set graphical parameters
xlabel('Heat transfer coefficient')
ylabel('Time required to drop below 400 degrees')
axis square
set(gca, 'YScale', 'log')
set(gca, 'XScale', 'log')
set(gca, 'XMinorTick', 'on')
set(gca, 'TickLength', [0.02, 0.05])
grid on
```

ENGR105 adventure summary



<u>Question</u>: define `t400` as the time required for `T_wall` to drop to `400C` for a given value of `h`. **<u>How can we solve for `t400`?</u>**

There are (at least) two different potential philosophical approaches to solve this problem:

1) "Brute force" arrival at the solution

- Calculate a large array of time points, determine the temperature at these points, then determine the first temperature that is less than 400C.
- Or, similarly, calculate the temperature at time zero after every small increment until the temperature drops below 400C.

<u>Example:</u>

```
% Time vector
t1 = 0:0.001:50000;
% Temperature vector
T = @(h) 50*(1./(t1+1) + 5*(t1+1).^(-4*h)) + 300;

% Preallocate vector to store cooling time
t_400v2 = zeros(size(h));

% Loop for each value value of h
for k = 1:length(h)
    % Calculate temperature profile
    T_h = T(h(k));

    % Use logical indexing to determine all time points corresponding to
    % T_h being less than 400
    t2 = t1(T_h < 400);

    % Store the first such time point
    t_400v2(k) = t2(1);
end
```

2) Use a numerical solver

- Newton-Raphson: root-finding algorithm. Use iterative calculations until we converge at the root of an equation.

<u>Example:</u> We have used this method a lot so far, and have already created a function to carry out this approach.

```
function xn = newtonSolve(F,xn,tol)
    % The purpose of this function is to apply the Newton-Raphson method
to
    % solve any arbitrary function, specified in an input F.
    % Inputs:
    % F: function and derivative
    % x: initial guess
    % tol: specified tolerance

    % Step 1: specify initial guess
```

```
    % Input xn -> initial guess
    xprev = inf; % Variable to store previous xn value -> ensure it is
                 % not xn to enter while loop

    % Step 2: iterate until xn -> xn+1
    while abs(xn - xprev) > tol
        % Apply input function to algorithm. NOTE: first output is
function
        % evaluated at xn, the second output is the derivative of the
        % function evaluated at xn
        f = F(xn);

        % Instead of calculating the "next" x value, store the previous
xn
        % value
        xprev = xn;

        % Step 3b: update xn to next value for the iteration
        xn = xn - f(1)/f(2);
    end
end
```

<u>For applications like this (well-behaved, arbitrary function): Newton-Raphson will generally work better than a brute force approach</u>

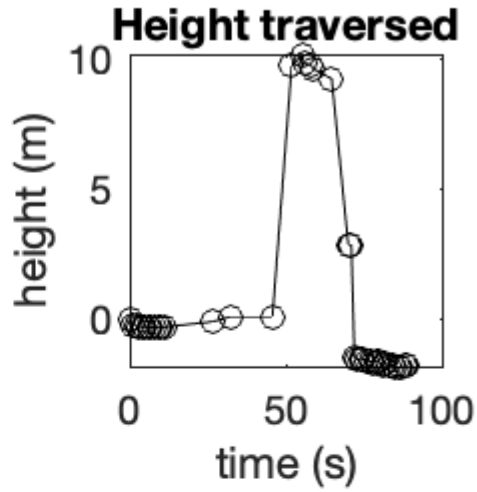
This can be seen in two ways:

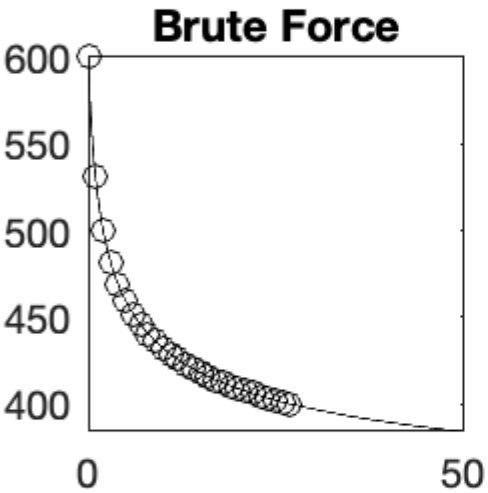
A) <u>Computational considerations</u> -- how many "calculations" need to be performed for each approach. We can depict this visually by plotting the data point following each iteration

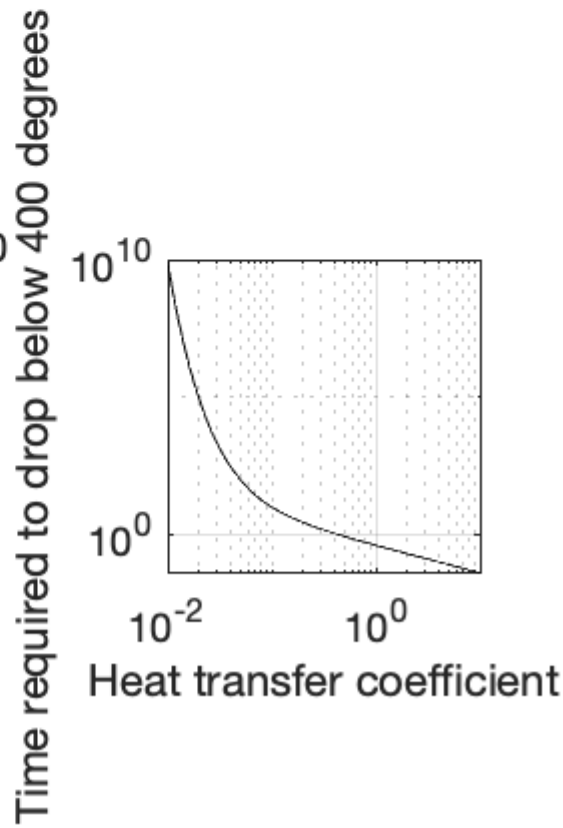
```
% Heat transfer coefficient to consider
h1 = 0.07;

% Temperature
T_wall = @(h) 50*(1./(t+1) + 5*(t+1).^(-4*h)) + 300;

% FIRST Subplot: Brute Force approach
subplot(1,2,1)

% Temperature profile corresponding to tested heat transfer coefficient
TempH1 = T_wall(h1);

% Simulate "brute force" approach
t_BF = t(TempH1 > 400);
TempH2 = TempH1(TempH1 > 400);

% Plot temperature profile
plot(t,TempH1,'k-')
hold on
% Plot brute force calculations
```

```
plot(t_BF(1:100:end),TempH2(1:100:end),'ko')
hold off
axis square
title('Brute Force')
```

ENGR105 adventure summary

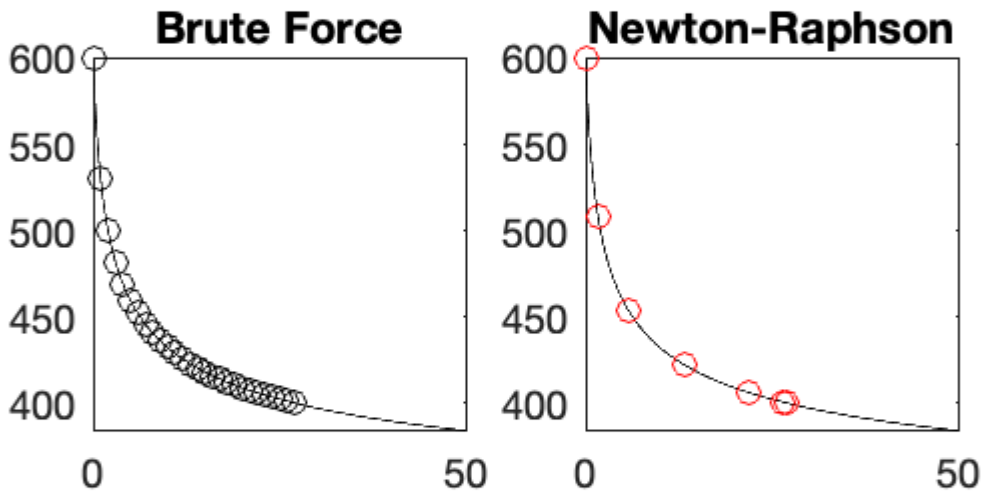






```
% SECOND Subplot: Newton-Raphson
subplot(1,2,2)

% Simulate Newton-Raphson:
% Determine t values for each iteration
t_NRiterations = newtonSolveReturnXvalues(@(x) quiz3(x,h1),0,10^-4);
% Determine corresponding temperature value
T_NR = 50*(1./(t_NRiterations+1) + 5*(t_NRiterations+1).^(-4*h1)) + 300;

% Plot temperature profile
plot(t,TempH1,'k-')
hold on
% Plot each NR iteration
plot(t_NRiterations,T_NR,'ro')
hold off
axis square
title('Newton-Raphson')
```

ENGR105 adventure summary



Takeaway: Newton-Raphson "finds" the solution in **fewer iterations** at a **higher precision** (stricter tolerance).

B) Memory considerations -- Newton-Raphson requires substantially fewer resources. In the above example, note that vectors (`t1` and `T`) each require >400MB of memory!

Other common themes and issues in Quiz 3

1) Uncertainty about `for` loop notation and indexing / not preallocating

```
close all % Close above plots
% Goal: perform some operation on each entry of the vector h.
% Declare vector of heat transfer coefficients
h = 1:0.1:10;
% Use a FOR loop, do some operation on each element of h and store the
% output in the corrsponding entry in some other vector
t_out = [];
for k = h
    t_out(end+1) = k^2;
end
```

This could be rewritten using **array preallocation**:

```
% Use a FOR loop, do some operation on each element of h and store the
% output in the corrsponding entry in some other vector
t_out = zeros(1,length(h));
for k = 1:length(h)
    t_out(k) = h(k)^2;
end
```

2) Newton Raphson / root-finding

Recall: N-R is a root-finding algorithm. As such, it can only be used to solve for when some function evaluates to zero. This is not a problem though, as f(t) = a can be rewritten as g(t) = f(t) - a, which can be used with the Newton-Raphson method.

Reminder: characters and strings

MATLAB can process text as character arrays and strings.

Character arrays are distinguished by single quotes (`'`).

```
a = 'k--';
b = 'LineWidth';
c = 'This is a character array';
disp(c)
```

```
This is a character array
```

Strings are created using double quotes (").

```
s1 = "This is a string";
disp(s1)
```

```
This is a string
```

Similarity: both can be used to store text information

Difference:

- Each character (including spaces and punctuation) in a character arrays corresponds to an index position in a "vector" or array.
- An entire string corresponds to one index position

```
% Index the above character array
c(1)
```

```
ans = 'T'
```

```
c(5:10)
```

```
ans = ' is a '
```

```
c(1:2:end)
```

```
ans = 'Ti sacaatrary'
```

```
% Use vector operations to determine the number of characters (including
% spaces) in a piece of text.
length(c)
```

```
ans = 25
```

```
% Use indexing on string. NOTE: only ONE element of s1
s1(1)
```

```
ans = "This is a string"
```

<u>Thus</u>: character arrays enable you to retrieve and manipulate each character in a piece of text, while strings treat the entire "block" of text as one unit.

Reminder: cell arrays

Cell arrays are arrays with indices that can hold a variety of data types or sizes - **all in the same variable**. Use `{}` to refer to cell contents and `()` to refer to the container.

```
% Create cell array
C = {'it is Thursday', [1 5 9], [2 3 1; 5 9 7], ...
    uint8([1 4]), [.5; .1; .9], {'hello',[1 5]}}
```

```
C = 1×6 cell array
```

```
    {'it is Thursday'}    {1×3 double}    {2×3 double}    {1×2 uint8}    {3×1
double}    {1×2 cell}
```

```
% Reference the cell container
C(1)
```

```
ans = 1×1 cell array
    {'it is Thursday'}
```

```
% Reference the contents of a particular cell
Str = C{1}
```

```
Str = 'it is Thursday'
```

```
% The contents could then be processed
Str(1:9 )
```

```
ans = 'it is Thu'
```

Working with text

The ability to work with and manipulate text is important in many different applications: both those that *explicitly* deal with text data (e.g. studying genomic data) and those that involve numerical data but have some *text identifier* (e.g. batch processing of data files, images, etc.).

Loading text

- `importdata()` reads in each line of text as a cell array index.

```
txt1 = importdata("TextData.txt");
disp(txt1)
```

```
    {'Hey Macklemore? can we go thrift shopping?'                                  }
    {'What, what, what, what'                                                      }
    {'What, what, what, what'                                                      }
    {'What, what, what, what'                                                      }
    {'What, what, what, what'                                                      }
    {'What, what, what, what'                                                      }
    {'What, what, what, what'                                                      }
    {'What, what, what, what'                                                      }
    {'What, what, what, what'                                                      }
    {'I'm gonna pop some tags'                                                     }
    {'Only got twenty dollars in my pocket'                                        }
    {'I, I, I'm hunting, looking for a come-up'                                    }
    {'This is frickin awesome'                                                     }
    {'Nah walk up to the club like, what up, I got a big shirt!'                   }
    {'I'm just pumped, just bought some stuff from the thrift shop'                }
    {'Ice on the fringe, it's so damn frosty'                                      }
    {'The people like, Damn! That's a cold ass honkey.'                            }
    {'Rollin' in, hella deep, headin' to the mezzanine'                            }
    {'Dressed in all pink, 'cept my gator shoes, those are green'                  }
    {'Draped in a leopard mink, girls standin' next to me'                         }
    {'Probably shoulda washed this, smells like R. Kelly's sheets'                 }
```

```
{'(Piss)'                                                                  }
{'But shoot, it was ninety-nine cents! (Bag it) Coppin' it, washin' it'    }
{''Bout to go and get some compliments'                                    }
{'Passin' up on those moccasins someone else's been walkin' in them'       }
{'Bummy and grungy, f it man, I am stuntin' and flossin' and'              }
{'And savin' my money and I'm hella happy that's a bargain, b'             }
{'I'ma take your grandpa's style, I'ma take your grandpa's style'          }
{'No for real ask your grandpa can I have his hand-me-downs?'              }
{'(Thank you) Velour jumpsuit and some house slippers'                     }
{'Dookie brown leather jacket that I found diggin''                        }
{'They had a broken keyboard, I bought a broken keyboard'                  }
{'I bought a skeet blanket, and then I bought a kneeboard'                 }
{'Hello, hello, my ace man, my Miller'                                     }
{'John Wayne ain't got nothing on my fringe game, hell no'                 }
{'I could take some Pro Wings, make them cool, sell those'                 }
{'The sneaker heads would be like Aw, he got the Velcros'                  }
{'I'm gonna pop some tags'                                                 }
{'Only got twenty dollars in my pocket'                                    }
{'I, I, I'm hunting, looking for a come-up'                                }
{'This is frickin awesome'                                                 }
{'I'm gonna pop some tags'                                                 }
{'Only got twenty dollars in my pocket'                                    }
{'I, I, I'm hunting, looking for a come-up'                                }
{'This is frickin awesome'                                                 }
{'What you know about rockin' a wolf on your noggin?'                      }
{'What you knowin' about wearin' a fur fox skin?'                          }
{'I'm digging, I'm digging, I'm searching right through that luggage'     }
{'One man's trash, that's another man's come up'                           }
{'Thank your granddad for donating that plaid button-up shirt'             }
{''Cause right now I'm up in her skirt'                                    }
{'I'm at the Goodwill, you can find me in the (Uptons)'                    }
{'I'm that, I'm that sucker searchin' in that section (Uptons)'            }
{'Your grammy, your aunty, your momma, your mammy'                         }
{'I'll take those flannel zebra jammies, second-hand, I rock that muhmuhmuhmuh'}
{'The built-in onesie with the socks on that muhmuhmuhmuh'                 }
{'I hit the party and they stop in that muhmuhmuhmuh'                      }
{'They be like, Oh, that Gucci. That's hella tight'                        }
{'I'm like, Yo that's fifty dollars for a T-shirt'                         }
{'Limited edition, let's do some simple addition'                          }
{'Fifty dollars for a T-shirt, that's just some ignorant fool (Shoot)'     }
{'I call that getting swindled and pimped (Shoot)'                         }
{'I call that getting tricked by a business'                               }
{'That shirt's hella dope'                                                 }
{'And having the same one as six other people in this club is a hella don't'   }
{'Peep game, come take a look through my telescope'                        }
{'Trying to get girls from a brand? Then you hella won't'                  }
{'Then you hella won't'                                                    }
{'I'm gonna pop some tags'                                                 }
{'Only got twenty dollars in my pocket'                                    }
{'I, I, I'm hunting, looking for a come-up'                                }
{'This is frickin awesome'                                                 }
{'I wear your granddad's clothes'                                          }
{'I look incredible'                                                       }
{'I'm in this big ass coat'                                                }
{'From that thrift shop down the road'                                     }
{'I wear your granddad's clothes'                                          }
{'I look incredible'                                                       }
{'I'm in this big ass coat'                                                }
{'From that thrift shop down the road'                                     }
{'I'm gonna pop some tags'                                                 }
{'Only got twenty dollars in my pocket'                                    }
```

```
    {'I, I, I'm hunting, looking for a come-up'                                    }
    {'This is frickin awesome'                                                     }
    {'Is that your grandma's coat?'                                                }
```

- Text files may be read into a workspace using `fileread()`, which brings in the text as one long character string.

```
txt2 = fileread("TextData.txt")
```

```
txt2 =
    'Hey Macklemore? can we go thrift shopping?
     What, what, what, what
     What, what, what, what
     What, what, what, what
     What, what, what, what
     What, what, what, what
     What, what, what, what
     What, what, what, what
     What, what, what, what
     I'm gonna pop some tags
     Only got twenty dollars in my pocket
     I, I, I'm hunting, looking for a come-up
     This is frickin awesome
     Nah walk up to the club like, what up, I got a big shirt!
     I'm just pumped, just bought some stuff from the thrift shop
     Ice on the fringe, it's so damn frosty
     The people like, Damn! That's a cold ass honkey.
     Rollin' in, hella deep, headin' to the mezzanine
     Dressed in all pink, 'cept my gator shoes, those are green
     Draped in a leopard mink, girls standin' next to me
     Probably shoulda washed this, smells like R. Kelly's sheets
     (Piss)
     But shoot, it was ninety-nine cents! (Bag it) Coppin' it, washin' it
     'Bout to go and get some compliments
     Passin' up on those moccasins someone else's been walkin' in them
     Bummy and grungy, f it man, I am stuntin' and flossin' and
     And savin' my money and I'm hella happy that's a bargain, b
     I'ma take your grandpa's style, I'ma take your grandpa's style
     No for real ask your grandpa can I have his hand-me-downs?
     (Thank you) Velour jumpsuit and some house slippers
     Dookie brown leather jacket that I found diggin'
     They had a broken keyboard, I bought a broken keyboard
     I bought a skeet blanket, and then I bought a kneeboard
     Hello, hello, my ace man, my Miller
     John Wayne ain't got nothing on my fringe game, hell no
     I could take some Pro Wings, make them cool, sell those
     The sneaker heads would be like Aw, he got the Velcros
     I'm gonna pop some tags
     Only got twenty dollars in my pocket
     I, I, I'm hunting, looking for a come-up
     This is frickin awesome
     I'm gonna pop some tags
     Only got twenty dollars in my pocket
     I, I, I'm hunting, looking for a come-up
     This is frickin awesome
     What you know about rockin' a wolf on your noggin?
     What you knowin' about wearin' a fur fox skin?
     I'm digging, I'm digging, I'm searching right through that luggage
     One man's trash, that's another man's come up
```

```
Thank your granddad for donating that plaid button-up shirt
'Cause right now I'm up in her skirt
I'm at the Goodwill, you can find me in the (Uptons)
I'm that, I'm that sucker searchin' in that section (Uptons)
Your grammy, your aunty, your momma, your mammy
I'll take those flannel zebra jammies, second-hand, I rock that muhmuhmuhmuh
The built-in onesie with the socks on that muhmuhmuhmuh
I hit the party and they stop in that muhmuhmuhmuh
They be like, Oh, that Gucci. That's hella tight
I'm like, Yo that's fifty dollars for a T-shirt
Limited edition, let's do some simple addition
Fifty dollars for a T-shirt, that's just some ignorant fool (Shoot)
I call that getting swindled and pimped (Shoot)
I call that getting tricked by a business
That shirt's hella dope
And having the same one as six other people in this club is a hella don't
Peep game, come take a look through my telescope
Trying to get girls from a brand? Then you hella won't
Then you hella won't
I'm gonna pop some tags
Only got twenty dollars in my pocket
I, I, I'm hunting, looking for a come-up
This is frickin awesome
I wear your granddad's clothes
I look incredible
I'm in this big ass coat
From that thrift shop down the road
I wear your granddad's clothes
I look incredible
I'm in this big ass coat
From that thrift shop down the road
I'm gonna pop some tags
Only got twenty dollars in my pocket
I, I, I'm hunting, looking for a come-up
This is frickin awesome
Is that your grandma's coat?'
```

```
disp(txt2)
```

```
Hey Macklemore? can we go thrift shopping?
What, what, what, what
What, what, what, what
What, what, what, what
What, what, what, what
What, what, what, what
What, what, what, what
What, what, what, what
What, what, what, what
I'm gonna pop some tags
Only got twenty dollars in my pocket
I, I, I'm hunting, looking for a come-up
This is frickin awesome
Nah walk up to the club like, what up, I got a big shirt!
I'm just pumped, just bought some stuff from the thrift shop
Ice on the fringe, it's so damn frosty
The people like, Damn! That's a cold ass honkey.
Rollin' in, hella deep, headin' to the mezzanine
Dressed in all pink, 'cept my gator shoes, those are green
Draped in a leopard mink, girls standin' next to me
Probably shoulda washed this, smells like R. Kelly's sheets
```

(Piss)
But shoot, it was ninety-nine cents! (Bag it) Coppin' it, washin' it
'Bout to go and get some compliments
Passin' up on those moccasins someone else's been walkin' in them
Bummy and grungy, f it man, I am stuntin' and flossin' and
And savin' my money and I'm hella happy that's a bargain, b
I'ma take your grandpa's style, I'ma take your grandpa's style
No for real ask your grandpa can I have his hand-me-downs?
(Thank you) Velour jumpsuit and some house slippers
Dookie brown leather jacket that I found diggin'
They had a broken keyboard, I bought a broken keyboard
I bought a skeet blanket, and then I bought a kneeboard
Hello, hello, my ace man, my Miller
John Wayne ain't got nothing on my fringe game, hell no
I could take some Pro Wings, make them cool, sell those
The sneaker heads would be like Aw, he got the Velcros
I'm gonna pop some tags
Only got twenty dollars in my pocket
I, I, I'm hunting, looking for a come-up
This is frickin awesome
I'm gonna pop some tags
Only got twenty dollars in my pocket
I, I, I'm hunting, looking for a come-up
This is frickin awesome
What you know about rockin' a wolf on your noggin?
What you knowin' about wearin' a fur fox skin?
I'm digging, I'm digging, I'm searching right through that luggage
One man's trash, that's another man's come up
Thank your granddad for donating that plaid button-up shirt
'Cause right now I'm up in her skirt
I'm at the Goodwill, you can find me in the (Uptons)
I'm that, I'm that sucker searchin' in that section (Uptons)
Your grammy, your aunty, your momma, your mammy
I'll take those flannel zebra jammies, second-hand, I rock that muhmuhmuhmuh
The built-in onesie with the socks on that muhmuhmuhmuh
I hit the party and they stop in that muhmuhmuhmuh
They be like, Oh, that Gucci. That's hella tight
I'm like, Yo that's fifty dollars for a T-shirt
Limited edition, let's do some simple addition
Fifty dollars for a T-shirt, that's just some ignorant fool (Shoot)
I call that getting swindled and pimped (Shoot)
I call that getting tricked by a business
That shirt's hella dope
And having the same one as six other people in this club is a hella don't
Peep game, come take a look through my telescope
Trying to get girls from a brand? Then you hella won't
Then you hella won't
I'm gonna pop some tags
Only got twenty dollars in my pocket
I, I, I'm hunting, looking for a come-up
This is frickin awesome
I wear your granddad's clothes
I look incredible
I'm in this big ass coat
From that thrift shop down the road
I wear your granddad's clothes
I look incredible
I'm in this big ass coat
From that thrift shop down the road
I'm gonna pop some tags
Only got twenty dollars in my pocket

```
I, I, I'm hunting, looking for a come-up
This is frickin awesome
Is that your grandma's coat?
```

Note the differences: `importdata` creates a **cell** array, `fileread` creates a **character** array.

```
size(txt1)
```

```
ans = 1×2
    85     1
```

```
size(txt2)
```

```
ans = 1×2
           1        3630
```

Character arrays

- **Storing consistent width character arrays**. Character arrays can be stored in a array-like format provided each **row is of the same length**.

```
M = ['hello' 'hi' ;'phone' 'it';'funny' 'oh']
```

```
M = 3×7 char array
    'hellohi'
    'phoneit'
    'funnyoh'
```

Concatenation errors will occur if all columns are not of equal length.

```
% F = ['mat' 'labs' ;'fun' 'times';'a' 'b']
```

- **Character arrays of variable width can be stored in cells.**

```
V = {'mat', 'labs' ;'fun' , 'times';'a', 'b'}
```

```
V = 3×2 cell array
    {'mat'}    {'labs' }
    {'fun'}    {'times'}
    {'a'  }    {'b'    }
```

Individual lists of characters within the cell can then be accessed.

```
% Reference the text in the above cell array
c = V{2,1}
```

```
c = 'fun'
```

```
% Note the indexing is the same as before. Above, we have a cell array in
% the format of a vector.
d = txt1{10}
```

```
d = 'I'm gonna pop some tags'
```

<u>Note</u>: In practice, **character arrays are typically utilized to store vectors of characters** and these vectors are stored in cells

- Character vectors and arrays can be **indexed similarly as numeric arrays** and operated upon by many common array functions

```
% Using vector indexing to retrieve sub arrays of characters from the "d"
t1 = d(1)
```

```
t1 = 'I'
```

```
t2 = d(7)
```

```
t2 = 'n'
```

```
t3 = d(1:2:end-2)
```

```
t3 = 'Imgnappsm a'
```

```
t4 = fliplr(d)
```

```
t4 = 'sgat emos pop annog m'I'
```

- Like numeric arrays, character array indices may be concatenated, replaced, and expanded

```
% Concatenate character arrays
d = [d,', yo']
```

```
d = 'I'm gonna pop some tags, yo'
```

```
% Expand array
d(end+1:end+5) = '. Woo'
```

```
d = 'I'm gonna pop some tags, yo. Woo'
```

```
% Delete entries
d([11:16,18:end-2]) = []
```

```
d = 'I'm gonna moo'
```

- Character arrays can use logical indexing

```
% Remove all spaces
d(d ~= ' ')
```

```
ans = 'I'mgonnamoo'
```

- Character arrays stored in cells may be indexed by **reference to cell** and **then array indices.**

```
% Create new character array
t1 = txt1{29}
```

```
t1 = 'No for real ask your grandpa can I have his hand-me-downs?'
```

```
% Create subarray
t2  = txt1{29}(1:11)
```

```
t2 = 'No for real'
```

```
% Return character array containing all "r"s
t3 = txt1{29}(txt1{29} == 'r')
```

```
t3 = 'rrrr'
```

String arrays

String arrays can be formulated just like character arrays, except columns need not be of the same length - **each string** is viewed as a single index of data.

```
S = ["position", "velocity";...
    "acceleration","g-force"]
```

```
S = 2×2 string array
    "position"        "velocity"
    "acceleration"    "g-force"
```

```
% Return the string in index position 1 (single index notation)
S(1)
```

```
ans = "position"
```

<u>Note</u>: Because each string in a string array is viewed as a single index, any index can be replaced by a string without explicit reference to the indices of the characters

```
S(1,2) = "velocity in x"
```

```
S = 2×2 string array
    "position"        "velocity in x"
    "acceleration"    "g-force"
```

```
S
```

```
S = 2×2 string array
    "position"        "velocity in x"
    "acceleration"    "g-force"
```

<u>Note</u>: Because each string is viewed as a **single index**, a string array index **must be converted to a character array** in order to access specific characters.This can be done with the `char` function.

```
% Convert one entry into a character sub array
c = char(S(2,1))
```

```
c = 'acceleration'
```

```
c = c(1:3)
```

```
c = 'acc'
```

Character and string functions

There are many high level functions to operate on character arrays and strings

- `isletter()`: returns logical array indicating whether a particular character is a **letter** or not (**true**: that character is a letter).

```
c1 = txt1{39}
```

```
c1 = 'Only got twenty dollars in my pocket'
```

```
% Returns logical array indicating whether a particular character is a
% letter or not (true: that character is a letter)
v = isletter(c1)
```

```
v = 1×36 logical array
   1   1   1   1   0   1   1   1   0   1   1   1   1   1   1   0   1   1   1   1   1
1   1   0   1   1   0   1   1   0   1   1   1   1   1   1
```

```
% Convert all to upper case
sU = upper(c1)
```

```
sU = 'ONLY GOT TWENTY DOLLARS IN MY POCKET'
```

```
% Convert all to lower case
sL = lower(c1)
```

```
sL = 'only got twenty dollars in my pocket'
```

```
% Create a character array of spaces
sB = blanks(7)
```

```
sB = '       '
```

- **Finding** a piece of text inside a character array

```
c1
```

```
c1 = 'Only got twenty dollars in my pocket'
```

```
% Create index at which word "dollars" appears
ind = strfind(c1, 'dollars')
```

```
ind = 17
```

Note: `strfind` is case-sensitive

```
% Create index at which word "Dollars" appears
ind = strfind(c1,'Dollars')
```

```
ind =

     []
```

- **Comparing** text

Character arrays. `strcmp()`can be used to successfully evaluate if one character array is the same as another

```
if ~strcmp('twenty',c1)
    disp('yes')
end
```

```
yes
```

Strings. Because strings (denoted by double quotation marks) are viewed as singular entities, they can be compared via decision even if they have differing lengths

```
if "owl" ~= "owls"
    disp('yes')
end
```

```
yes
```

Converting numbers to characters

`num2str()` can be used to convert numbers to characters for concatenation with other characters.

- One situation where this can be useful: converting a loop index or counter into text.

```
% Assume: 10 image files are present in a folder and I want to easily
% open all of them. First: declare base file name
basefile = 'imageFile';
% Second: use a for loop to create text (that could be used to open each)
for k = 1:10
    disp([basefile,num2str(k),'.tif'])
```

```
end
```

```
imageFile1.tif
imageFile2.tif
imageFile3.tif
imageFile4.tif
imageFile5.tif
imageFile6.tif
imageFile7.tif
imageFile8.tif
imageFile9.tif
imageFile10.tif
```

Practice Problem: analyzing song lyrics

Consider the data in `txt1`, imported above. How many words are in the 46-th line? What is the average word length?

```
%Step 1: Create variable with 46th of code from txt1

%Step 2: Can use spaces to determine where the words are -> have an
%isletter function and an isspace function
%This would determine where spaces are

%Step 3: for space, assign "previous" letters to a cell storing words

%add space to front and end of character array
```

Q1: how many words are there?

Q2: What is the average word length?

TOPIC 16: Working with text, pt 2

ENGR105, 10/27/20

Reminder: working with text

The ability to work with and manipulate text is important in many different applications: both those that *explicitly* deal with text data (e.g. studying genomic data) and those that involve numerical data but have some *text identifier* (e.g. batch processing of data files, images, etc.).

Loading text.. `importdata()` reads in each line of text as a cell array index.

```
txt1 = importdata('TextData.txt');
size(txt1)
```

```
ans = 1×2
```

```
    85     1
```

`fileread()` reads a text file as one long character array.

```
%read as one long character array
txt2 = fileread('TextData.txt');
size(txt2)
```

```
ans = 1×2
           1        3630
```

Note: unless explicitly specified (HW 8), you should use which ever works better for a given situation.

Character arrays can be read from cells in a cell array.

```
% Read the 39th line as a character array
c = txt1{39}
```

```
c = 'Only got twenty dollars in my pocket'
```

Character vectors and arrays can be **indexed similarly as numeric arrays** and operated upon by many common array functions

```
% Using vector indexing to retrieve sub arrays of characters from the "c"
t1 = c(1:5)
```

```
t1 = 'Only '
```

```
% Concatenate array
t2 = [t1,' , ',t1]
```

```
t2 = 'Only  , Only '
```

Character arrays can use **logical indexing**.

```
% Create a text array without spaces
t2 = t2(t2 ~= ' ')
```

```
t2 = 'Only,Only'
```

Character and string functions. There are many high level functions to operate on character arrays and strings

- `isletter()`: returns logical array indicating whether a particular character is a **letter** or not (**true**: that character is a letter).

```
c = txt1(39)
```

```
c = 1×1 cell array
    {'Only got twenty dollars in my pocket'}
```

```
% Logical array indicating positions that are letters
v1 = isletter(c)
```

```
v1 = logical
   0
```

- `isspace()`: returns logical array indicating whether a particular character is a **space** or not (**true**: that character is a space).

```
% Logical array indicating positions that are spaces
v2 = isspace(c)
```

```
v2 = logical
   0
```

- Functions to change the **case** of a character sting

```
% Convert all to upper case
sU = upper(c)
```

```
sU = 1×1 cell array
    {'ONLY GOT TWENTY DOLLARS IN MY POCKET'}
```

```
% Convert all to lower case
sL = lower(c)
```

```
sL = 1×1 cell array
    {'only got twenty dollars in my pocket'}
```

- **Finding** a piece of text inside a character array

```
% Create index at which word "dollars" appears
ind = strfind(c,'twenty')
```

```
ind = 1×1 cell array
    {[10]}
```

```
ind = strfind(c,'t')
```

```
ind = 1×1 cell array
    {1×4 double}
```

Many more functions may be found here:

https://www.mathworks.com/help/matlab/characters-and-strings.html

Practice Problem: analyzing song lyrics

Consider the data in `txt1`, imported above. How many words are in the 46-th line? What is the average word length?

```
% Step 1: Read the 46th line
c1 = txt1{46}
```

```
c1 = 'What you know about rockin' a wolf on your noggin?'
```

```
%Step 2: Remove punctiation
c1 = c1(isletter(c1)|isspace(c1))
```

```
c1 = 'What you know about rockin a wolf on your noggin'
```

```
%Step 3: Isolate each word
% Step 3a: add a space to front and back of text
c1 = [' ', c1, ' ']
```

```
c1 = ' What you know about rockin a wolf on your noggin '
```

```
% Step 3b: Determine where spaces are
spaces = strfind(c1, ' ')
```

```
spaces = 1×11
     1     6    10    15    21    28    30    35    38    43    50
```

```
% Step 3c: Store each word in cell array. note: if we have n spaces, then
% we will have n - 1 words.
myCell = cell(1,length(spaces)-1); % Preallocate cell array

% loop through all spaces to store words
for k = 1:length(myCell)
    myCell{k} = c1(spaces(k)+1:spaces(k+1)-1);
end
disp(myCell)
```

```
    {'What'}    {'you'}    {'know'}    {'about'}    {'rockin'}    {'a'}    {'wolf'}
{'on'}    {'your'}    {'noggin'}
```

Q1: how many words are there?

```
numWords = numel(myCell)
```

```
numWords = 10
```

Q2: What is the average word length?

```
numLetters = 0;
%Loop through all words, add number of letters
for k = 1:numWords
    numLetters = numLetters + length(myCell{k})
end
```

```
numLetters = 4
numLetters = 7
numLetters = 11
numLetters = 16
numLetters = 22
numLetters = 23
numLetters = 27
numLetters = 29
numLetters = 33
numLetters = 39
```

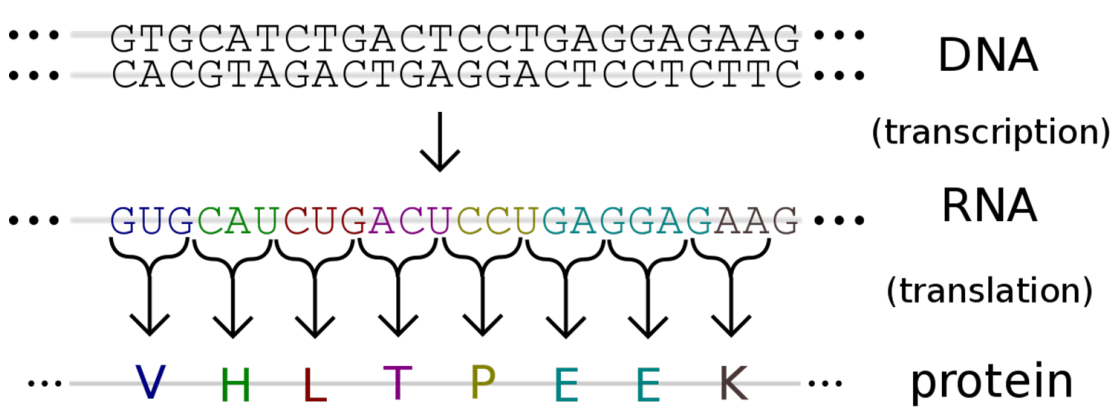
```
avgLength = numLetters/numWords
```

```
avgLength = 3.9000
```

Q3: in the above case, we analyzed one *line* of text. How could we modify this to analyze *all* of the lines of text in the file?

Practice Problem: the central dogma of molecular biology

The **central dogma of molecular biology** is an explanation of the genetic flow of information in a biological system, often stated as "DNA makes RNA, and RNA makes protein." *Depicted visually (from* wikipedia*):*



<u>Key terms for this problem:</u>

- **DNA**: the "genetic blueprint" for life. Strands of DNA are biopolymers consisting of a sequence of monomers, known as nucleotides. Each nucleotide consists of one base (cytosine [**C**], guanine [**G**], adenine [**A**] or thymine [**T**]), a sugar (deoxyribose), and a phosphate group. **Information is stored in the sequence of bases**, and DNA sequences are represented as a sequence of A, T, C, and G. Example: `TCTTGTTTTATTG`
- **mRNA**: the "instructions copied from DNA" dictating how to construct a protein. Like DNA, strands of RNA are biopolymers consisting of a sequence of nucleotide monomers. In this case, each nucleotide consists of one base (cytosine [**C**], guanine [**G**], adenine [**A**] or uracil [**U**]), a sugar (ribose), and a phosphate group. An mRNA strand is **produced from its DNA template**, with all bases matching except with uracils replacing thymines. Information is stored in the sequence of bases, and mRNA sequences are represented as a sequence of A, U, C, and G. Example corresponding to the above: `UCUUGUUUUAUUG`
- **Protein**: the "functional units" produced from an mRNA template. Encoded from mRNA, proteins are biopolymers that consist of a sequence of amino acid monomers. There are 20 naturally occuring amino acids.
- **Codon**: sequence of three bases that specify which amino acid will be added next to *a protein that is being constructed*.

```
% List of codons. Note the use of importdata()
codon_list = importdata('codonList.txt');
disp(codon_list)
```

```
    {'TTT,Phenylalanine,F'}
    {'TTC,Phenylalanine,F'}
    {'TTA,Leucine,L'      }
    {'TTG,Leucine,L'      }
    {'TCT,Serine,S'       }
    {'TCC,Serine,S'       }
    {'TCA,Serine,S'       }
    {'TCG,Serine,S'       }
    {'TAT,Tyrosine,Y'     }
    {'TAC,Tyrosine,Y'     }
    {'TAA,Stop,X'         }
    {'TAG,Stop,X'         }
    {'TGT,Cysteine,C'     }
    {'TGC,Cysteine,C'     }
    {'TGA,Stop,X'         }
    {'TGG,Tryptophan,W'   }
    {'CTT,Leucine,L'      }
    {'CTC,Leucine,L'      }
    {'CTA,Leucine,L'      }
    {'CTG,Leucine,L'      }
    {'CCT,Proline,P'      }
    {'CCC,Proline,P'      }
    {'CCA,Proline,P'      }
    {'CCG,Proline,P'      }
    {'CAT,Histidine,H'    }
    {'CAC,Histidine,H'    }
    {'CAA,Glutamine,Q'    }
    {'CAG,Glutamine,Q'    }
    {'CGT,Arginine,R'     }
    {'CGC,Arginine,R'     }
    {'CGA,Arginine,R'     }
    {'CGG,Arginine,R'     }
    {'ATT,Isoleucine,I'   }
    {'ATC,Isoleucine,I'   }
```

```
{'ATA,Isoleucine,I'   }
{'ATG,Methionine,M'   }
{'ACT,Threonine,T'    }
{'ACC,Threonine,T'    }
{'ACA,Threonine,T'    }
{'ACG,Threonine,T'    }
{'AAT,Asparagine,N'   }
{'AAC,Asparagine,N'   }
{'AAA,Lysine,K'       }
{'AAG,Lysine,K'       }
{'AGT,Serine,S'       }
{'AGC,Serine,S'       }
{'AGA,Arginine,R'     }
{'AGG,Arginine,R'     }
{'GTT,Valine,V'       }
{'GTC,Valine,V'       }
{'GTA,Valine,V'       }
{'GTG,Valine,V'       }
{'GCT,Alanine,A'      }
{'GCC,Alanine,A'      }
{'GCA,Alanine,A'      }
{'GCG,Alanine,A'      }
{'GAT,Aspartate,D'    }
{'GAC,Aspartate,D'    }
{'GAA,Glutamate,E'    }
{'GAG,Glutamate,E'    }
{'GGT,Glycine,G'      }
{'GGC,Glycine,G'      }
{'GGA,Glycine,G'      }
{'GGG,Glycine,G'      }
```

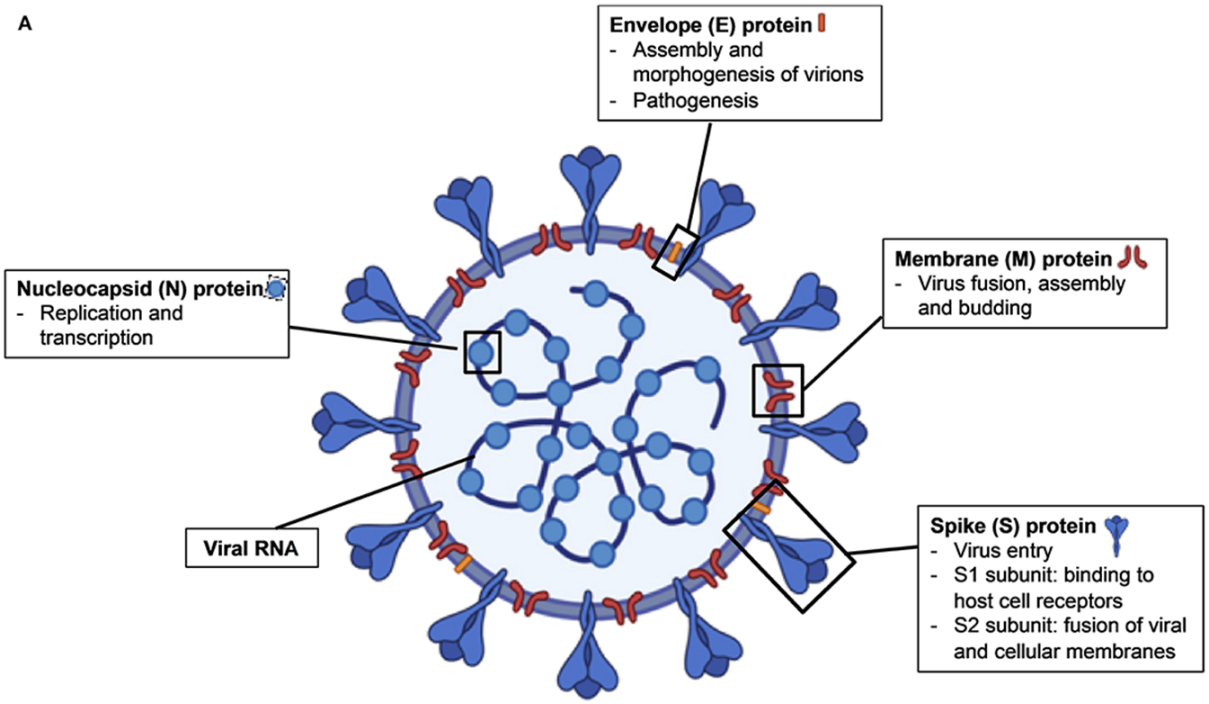
Important notes:

- Genes occur in sets of three bases. One codon (three bases) specifies an amino acid.
- Methionine (DNA base sequence `ATG`) is nearly always the "Start" codon, indicating the start of a sequence for a gene.
- There are three common "Stop" codons that indicate where a gene sequence stops: DNA base sequences `TAA`, `TAG`, `TGA`.
- We are representing these biomolecules as text. Remember that these correspond to *different chemical structures*.

Short summary:

- DNA specifies mRNA, which encodes the protein to be made.
- DNA/mRNA can be represented as text - either ATCG or AUCG
- Protein can be represented as text - a string of one of twenty letters.

In this problem, we will analyze this flow of biological information describing the **"spike"** protein on the membrane of **severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2)**.SARS-CoV-2 is an enveloped, *positive-sense*, single-stranded RNA virus that causes coronavirus disease 2019 (COVID-19). Virus particles include the RNA genetic material and structural proteins needed for invasion of host cells. Once inside the cell the infecting RNA is used to encode structural proteins that make up virus particles, nonstructural proteins that direct virus assembly, transcription, replication and host control and accessory proteins whose function has not been determined.



Source DOI: 10.3389/fimmu.2020.00879

The structural proteins of SARS-CoV-2 include the envelope protein, **spike or surface glycoprotein**, membrane protein and the nucleocapsid protein. **The spike glycoprotein is found on the outside of the virus particle and gives coronavirus viruses their crown-like appearance. This glycoprotein mediates attachment of the virus particle and entry into the host cell.** This protein is an *important target* for vaccine development, antibody therapies and diagnostic antigen-based tests.

Key resources and references:

- Review of SARS-CoV-2 and COVID-19: https://www.nature.com/articles/s41579-020-00459-7
- Genomic information (Wuhan-Hu-1): https://www.ncbi.nlm.nih.gov/nuccore/NC_045512

Problem statement

Given a DNA primer (a short sequence of DNA representing the first few bases of a given gene) for the spike glycoprotein and the genome of SARS-CoV-2, answer the following questions:

Question 1: how many bases are in the SARS-CoV-2 genome?

Question 2: how many of *each* base appear in the genome?

Question 3: given a primer, what is the DNA sequence for the spike glycoprotein? How many bases long is it?

Question 4: what is the mRNA transcript of the spike protein?

Question 5: What is the amino acid sequence of the spike protein?

Question 6: How many of each amino acid are in the spike protein?

First thing to do: organize information that is provided

SARS-CoV-2 genome:

```
% Import genomic information as one long character array.
% NOTE: difference from importdata()
sarsCov2_genome = fileread('sarsCov2_genome.txt');  % Read text file
sarsCov2_genome = sarsCov2_genome(isletter(sarsCov2_genome));   % remove spaces
and commas
sarsCov2_genome = upper(sarsCov2_genome);   % Convert to upper case
disp(sarsCov2_genome)
```

```
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAACGAACTTTAAAATC
TGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAACTAATTACTGTCGTTGACAGGACACGAGT
AACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTGTTGCAGCCGATCATCAGCACATCTAGGTTTCGTCCGGGTGTG
ACCGAAAGGTAAGATGGAGAGCCTTGTCCCTGGTTTCAACGAGAAAACACACGTCCAACTCAGTTTGCCTGTTTTACAGGTTCG
CGACGTGCTCGTACGTGGCTTTGGAGACTCCGTGGAGGAGGTCTTATCAGAGGCACGTCAACATCTTAAAGATGGCACTTGTGG
CTTAGTAGAAGTTGAAAAAGGCGTTTTGCCTCAACTTGAACAGCCCTATGTGTTCATCAAACGTTCGGATGCTCGAACTGCACC
TCATGGTCATGTTATGGTTGAGCTGGTAGCAGAACTCGAAGGCATTCAGTACGGTCGTAGTGGTGAGACACTTGGTGTCCTTGT
CCCTCATGTGGGCGAAATACCAGTGGCTTACCGCAAGGTTCTTCTTCGTAAGAACGGTAATAAAGGAGCTGGTGGCCATAGTTA
CGGCGCCGATCTAAAGTCATTTGACTTAGGCGACGAGCTTGGCACTGATCCTTATGAAGATTTTCAAGAAAACTGGAACACTAA
ACATAGCAGTGGTGTTACCCGTGAACTCATGCGTGAGCTTAACGGAGGGGCATACACTCGCTATGTCGATAACAACTTCTGTGG
CCCTGATGGCTACCCTCTTGAGTGCATTAAAGACCTTCTAGCACGTGCTGGTAAAGCTTCATGCACTTTGTCCGAACAACTGGA
CTTTATTGACACTAAGAGGGGTGTATACTGCTGCCGTGAACATGAGCATGAAATTGCTTGGTACACGGAACGTTCTGAAAAGAG
CTATGAATTGCAGACACCTTTTGAAATTAAATTGGCAAAGAAATTTGACACCTTCAATGGGGAATGTCCAAATTTTGTATTTCC
CTTAAATTCCATAATCAAGACTATTCAACCAAGGGTTGAAAAGAAAAAGCTTGATGGCTTTATGGGTAGAATTCGATCTGTCTA
TCCAGTTGCGTCACCAAATGAATGCAACCAAATGTGCCTTTCAACTCTCATGAAGTGTGATCATTGTGGTGAAACTTCATGGCA
GACGGGCGATTTTGTTAAAGCCACTTGCGAATTTTGTGGCACTGAGAATTTGACTAAAGAAGGTGCCACTACTTGTGGTTACTT
ACCCCAAAATGCTGTTGTTAAAATTTATTGTCCAGCATGTCACAATTCAGAAGTAGGACCTGAGCATAGTCTTGCCGAATACCA
TAATGAATCTGGCTTGAAAACCATTCTTCGTAAGGGTGGTCGCACTATTGCCTTTGGAGGCTGTGTGTTCTCTTATGTTGGTTG
CCATAACAAGTGTGCCTATTGGGTTCCACGTGCTAGCGCTAACATAGGTTGTAACCATACAGGTGTTGTTGGAGAAGGTTCCGA
AGGTCTTAATGACAACCTTCTTGAAATACTCCAAAAAGAGAAAGTCAACATCAATATTGTTGGTGACTTTAAACTTAATGAAGA
GATCGCCATTATTTTGGCATCTTTTTCTGCTTCCACAAGTGCTTTTGTGGAAACTGTGAAAGGTTTGGATTATAAAGCATTCAA
ACAAATTGTTGAATCCTGTGGTAATTTTAAAGTTACAAAAGGAAAAGCTAAAAAAGGTGCCTGGAATATTGGTGAACAGAAATC
AATACTGAGTCCTCTTTATGCATTTGCATCAGAGGCTGCTCGTGTTGTACGATCAATTTTCTCCCGCACTCTTGAAACTGCTCA
AAATTCTGTGCGTGTTTTACAGAAGGCCGCTATAACAATACTAGATGGAATTTCACAGTATTCACTGAGACTCATTGATGCTAT
GATGTTCACATCTGATTTGGCTACTAACAATCTAGTTGTAATGGCCTACATTACAGGTGGTGTTGTTCAGTTGACTTCGCAGTG
GCTAACTAACATCTTTGGCACTGTTTATGAAAAACTCAAACCCGTCCTTGATTGGCTTGAAGAGAAGTTTAAGGAAGGTGTAGA
GTTTCTTAGAGACGGTTGGGAAATTGTTAAATTTATCTCAACCTGTGCTTGTGAAATTGTCGGTGGACAAATTGTCACCTGTGC
AAAGGAAATTAAGGAGAGTGTTCAGACATTCTTTAAGCTTGTAAATAAATTTTTGGCTTTGTGTGCTGACTCTATCATTATTGG
TGGAGCTAAACTTAAAGCCTTGAATTTAGGTGAAACATTTGTCACGCACTCAAAGGGATTGTACAGAAAGTGTGTTAAATCCAG
AGAAGAAACTGGCCTACTCATGCCTCTAAAAGCCCCAAAAGAAATTATCTTCTTAGAGGGAGAAACACTTCCCACAGAAGTGTT
AACAGAGGAAGTTGTCTTGAAAACTGGTGATTTACAACCATTAGAACAACCTACTAGTGAAGCTGTTGAAGCTCCATTGGTTGG
TACACCAGTTTGTATTAACGGGCTTATGTTGCTCGAAATCAAAGACACAGAAAAGTACTGTGCCCTTGCACCTAATATGATGGT
AACAAACAATACCTTCACACTCAAAGGCGGTGCACCAACAAAGGTTACTTTTGGTGATGACACTGTGATAGAAGTGCAAGGTTA
CAAGAGTGTGAATATCACTTTTGAACTTGATGAAAGGATTGATAAAGTACTTAATGAGAAGTGCTCTGCCTATACAGTTGAACT
CGGTACAGAAGTAAATGAGTTCGCCTGTGTTGTGGCAGATGCTGTCATAAAAACTTTGCAACCAGTATCTGAATTACTTACACC
ACTGGGCATTGATTTAGATGAGTGGAGTATGGCTACATACTACTTATTTGATGAGTCTGGTGAGTTTAAATTGGCTTCACATAT
GTATTGTTCTTTCTACCCTCCAGATGAGGATGAAGAAGAAGGTGATTGTGAAGAAGAAGAGTTTGAGCCATCAACTCAATATGA
GTATGGTACTGAAGATGATTACCAAGGTAAACCTTTGGAATTTGGTGCCACTTCTGCTGCTCTTCAACCTGAAGAAGAGCAAGA
AGAAGATTGGTTAGATGATGATAGTCAACAAACTGTTGGTCAACAAGACGGCAGTGAGGACAATCAGACAACTACTATTCAAAC
```

AATTGTTGAGGTTCAACCTCAATTAGAGATGGAACTTACACCAGTTGTTCAGACTATTGAAGTGAATAGTTTTAGTGGTTATTT
AAAACTTACTGACAATGTATACATTAAAAATGCAGACATTGTGGAAGAAGCTAAAAAGGTAAAACCAACAGTGGTTGTTAATGC
AGCCAATGTTTACCTTAAACATGGAGGAGGTGTTGCAGGAGCCTTAAATAAGGCTACTAACAATGCCATGCAAGTTGAATCTGA
TGATTACATAGCTACTAATGGACCACTTAAAGTGGGTGGTAGTTGTGTTTTAAGCGGACACAATCTTGCTAAACACTGTCTTCA
TGTTGTCGGCCCAAATGTTAACAAAGGTGAAGACATTCAACTTCTTAAGAGTGCTTATGAAAATTTTAATCAGCACGAAGTTCT
ACTTGCACCATTATTATCAGCTGGTATTTTTGGTGCTGACCCTATACATTCTTTAAGAGTTTGTGTAGATACTGTTCGCACAAA
TGTCTACTTAGCTGTCTTTGATAAAAATCTCTATGACAAACTTGTTTCAAGCTTTTTGGAAATGAAGAGTGAAAAGCAAGTTGA
ACAAAAGATCGCTGAGATTCCTAAAGAGGAAGTTAAGCCATTTATAACTGAAAGTAAACCTTCAGTTGAACAGAGAAAACAAGA
TGATAAGAAAATCAAAGCTTGTGTTGAAGAAGTTACAACAACTCTGGAAGAAACTAAGTTCCTCACAGAAAACTTGTTACTTTA
TATTGACATTAATGGCAATCTTCATCCAGATTCTGCCACTCTTGTTAGTGACATTGACATCACTTTCTTAAAGAAAGATGCTCC
ATATATAGTGGGTGATGTTGTTCAAGAGGGTGTTTTAACTGCTGTGGTTATACCTACTAAAAAGGCTGGTGGCACTACTGAAAT
GCTAGCGAAAGCTTTGAGAAAAGTGCCAACAGACAATTATATAACCACTTACCCGGGTCAGGGTTTAAATGGTTACACTGTAGA
GGAGGCAAAGACAGTGCTTAAAAAGTGTAAAAGTGCCTTTTACATTCTACCATCTATTATCTCTAATGAGAAGCAAGAAATTCT
TGGAACTGTTTCTTGGAATTTGCGAGAAATGCTTGCACATGCAGAAGAAACACGCAAATTAATGCCTGTCTGTGTGGAAACTAA
AGCCATAGTTTCAACTATACAGCGTAAATATAAGGGTATTAAAATACAAGAGGGTGTGGTTGATTATGGTGCTAGATTTTACTT
TTACACCAGTAAAACAACTGTAGCGTCACTTATCAACACACTTAACGATCTAAATGAAACTCTTGTTACAATGCCACTTGGCTA
TGTAACACATGGCTTAAATTTGGAAGAAGCTGCTCGGTATATGAGATCTCTCAAAGTGCCAGCTACAGTTTCTGTTTCTTCACC
TGATGCTGTTACAGCGTATAATGGTTATCTTACTTCTTCTTCTAAAACACCTGAAGAACATTTTATTGAAACCATCTCACTTGC
TGGTTCCTATAAAGATTGGTCCTATTCTGGACAATCTACACAACTAGGTATAGAATTTCTTAAGAGAGGTGATAAAAGTGTATA
TTACACTAGTAATCCTACCACATTCCACCTAGATGGTGAAGTTATCACCTTTGACAATCTTAAGACACTTCTTTCTTTGAGAGA
AGTGAGGACTATTAAGGTGTTTACAACAGTAGACAACATTAACCTCCACACGCAAGTTGTGGACATGTCAATGACATATGGACA
ACAGTTTGGTCCAACTTATTTGGATGGAGCTGATGTTACTAAAATAAAACCTCATAATTCACATGAAGGTAAAACATTTTATGT
TTTACCTAATGATGACACTCTACGTGTTGAGGCTTTTGAGTACTACCACACAACTGATCCTAGTTTTCTGGGTAGGTACATGTC
AGCATTAAATCACACTAAAAAGTGGAAATACCCACAAGTTAATGGTTTAACTTCTATTAAATGGGCAGATAACAACTGTTATCT
TGCCACTGCATTGTTAACACTCCAACAAATAGAGTTGAAGTTTAATCCACCTGCTCTACAAGATGCTTATTACAGAGCAAGGGC
TGGTGAAGCTGCTAACTTTTGTGCACTTATCTTAGCCTACTGTAATAAGACAGTAGGTGAGTTAGGTGATGTTAGAGAAACAAT
GAGTTACTTGTTTCAACATGCCAATTTAGATTCTTGCAAAAGAGTCTTGAACGTGGTGTGTAAAACTTGTGGACAACAGCAGAC
AACCCTTAAGGGTGTAGAAGCTGTTATGTACATGGGCACACTTTCTTATGAACAATTTAAGAAAGGTGTTCAGATACCTTGTAC
GTGTGGTAAACAAGCTACAAAATATCTAGTACAACAGGAGTCACCTTTTGTTATGATGTCAGCACCACCTGCTCAGTATGAACT
TAAGCATGGTACATTTACTTGTGCTAGTGAGTACACTGGTAATTACCAGTGTGGTCACTATAAACATATAACTTCTAAAGAAAC
TTTGTATTGCATAGACGGTGCTTTACTTACAAAGTCCTCAGAATACAAAGGTCCTATTACGGATGTTTTCTACAAAGAAAACAG
TTACACAACAACCATAAAACCAGTTACTTATAAATTGGATGGTGTTGTTTGTACAGAAATTGACCCTAAGTTGGACAATTATTA
TAAGAAAGACAATTCTTATTTCACAGAGCAACCAATTGATCTTGTACCAAACCAACCATATCCAAACGCAAGCTTCGATAATTT
TAAGTTTGTATGTGATAATATCAAATTTGCTGATGATTTAAACCAGTTAACTGGTTATAAGAAACCTGCTTCAAGAGAGCTTAA
AGTTACATTTTTCCCTGACTTAAATGGTGATGTGGTGGCTATTGATTATAAACACTACACACCCTCTTTTAAGAAAGGAGCTAA
ATTGTTACATAAACCTATTGTTTGGCATGTTAACAATGCAACTAATAAAGCCACGTATAAACCAAATACCTGGTGTATACGTTG
TCTTTGGAGCACAAAACCAGTTGAAACATCAAATTCGTTTGATGTACTGAAGTCAGAGGACGCGCAGGGAATGGATAATCTTGC
CTGCGAAGATCTAAAACCAGTCTCTGAAGAAGTAGTGGAAAATCCTACCATACAGAAAGACGTTCTTGAGTGTAATGTGAAAAC
TACCGAAGTTGTAGGAGACATTATACTTAAACCAGCAAATAATAGTTTAAAAATTACAGAAGAGGTTGGCCACACAGATCTAAT
GGCTGCTTATGTAGACAATTCTAGTCTTACTATTAAGAAACCTAATGAATTATCTAGAGTATTAGGTTTGAAAACCCTTGCTAC
TCATGGTTTAGCTGCTGTTAATAGTGTCCCTTGGGATACTATAGCTAATTATGCTAAGCCTTTTCTTAACAAAGTTGTTAGTAC
AACTACTAACATAGTTACACGGTGTTTAAACCGTGTTTGTACTAATTATATGCCTTATTTCTTTACTTTATTGCTACAATTGTG
TACTTTTACTAGAAGTACAAATTCTAGAATTAAAGCATCTATGCCGACTACTATAGCAAAGAATACTGTTAAGAGTGTCGGTAA
ATTTTGTCTAGAGGCTTCATTTAATTATTTGAAGTCACCTAATTTTTCTAAACTGATAAATATTATAATTTGGTTTTTACTATT
AAGTGTTTGCCTAGGTTCTTTAATCTACTCAACCGCTGCTTTAGGTGTTTTAATGTCTAATTTAGGCATGCCTTCTTACTGTAC
TGGTTACAGAGAAGGCTATTTGAACTCTACTAATGTCACTATTGCAACCTACTGTACTGGTTCTATACCTTGTAGTGTTTGTCT
TAGTGGTTTAGATTCTTTAGACACCTATCCTTCTTTAGAAACTATACAAATTACCATTTCATCTTTTAAATGGGATTTAACTGC
TTTTGGCTTAGTTGCAGAGTGGTTTTTGGCATATATTCTTTTCACTAGGTTTTTCTATGTACTTGGATTGGCTGCAATCATGCA
ATTGTTTTTCAGCTATTTTGCAGTACATTTTATTAGTAATTCTTGGCTTATGTGGTTAATAATTAATCTTGTACAAATGGCCCC
GATTTCAGCTATGGTTAGAATGTACATCTTCTTTGCATCATTTTATTATGTATGGAAAAGTTATGTGCATGTTGTAGACGGTTG
TAATTCATCAACTTGTATGATGTGTTACAAACGTAATAGAGCAACAAGAGTCGAATGTACAACTATTGTTAATGGTGTTAGAAG
GTCCTTTTATGTCTATGCTAATGGAGGTAAAGGCTTTTGCAAACTACACAATTGGAATTGTGTTAATTGTGATACATTCTGTGC
TGGTAGTACATTTATTAGTGATGAAGTTGCGAGAGACTTGTCACTACAGTTTAAAAGACCAATAAATCCTACTGACCAGTCTTC
TTACATCGTTGATAGTGTTACAGTGAAGAATGGTTCCATCCATCTTTACTTTGATAAAGCTGGTCAAAAGACTTATGAAAGACA
TTCTCTCTCTCATTTTGTTAACTTAGACAACCTGAGAGCTAATAACACTAAAGGTTCATTGCCTATTAATGTTATAGTTTTTGA
TGGTAAATCAAAATGTGAAGAATCATCTGCAAAATCAGCGTCTGTTTACTACAGTCAGCTTATGTGTCAACCTATACTGTTACT
AGATCAGGCATTAGTGTCTGATGTTGGTGATAGTGCGGAAGTTGCAGTTAAAATGTTTGATGCTTACGTTAATACGTTTTCATC
AACTTTTAACGTACCAATGGAAAAACTCAAAACACTAGTTGCAACTGCAGAAGCTGAACTTGCAAAGAATGTGTCCTTAGACAA
TGTCTTATCTACTTTTATTTCAGCAGCTCGGCAAGGGTTTGTTGATTCAGATGTAGAAACTAAAGATGTTGTTGAATGTCTTAA
ATTGTCACATCAATCTGACATAGAAGTTACTGGCGATAGTTGTAATAACTATATGCTCACCTATAACAAAGTTGAAAACATGAC
ACCCCGTGACCTTGGTGCTTGTATTGACTGTAGTGCGCGTCATATTAATGCGCAGGTAGCAAAAAGTCACAACATTGCTTTGAT

ATGGAACGTTAAAGATTTCATGTCATTGTCTGAACAACTACGAAAACAAATACGTAGTGCTGCTAAAAAGAATAACTTACCTTT
TAAGTTGACATGTGCAACTACTAGACAAGTTGTTAATGTTGTAACAACAAAGATAGCACTTAAGGGTGGTAAAATTGTTAATAA
TTGGTTGAAGCAGTTAATTAAAGTTACACTTGTGTTCCTTTTTGTTGCTGCTATTTTCTATTTAATAACACCTGTTCATGTCAT
GTCTAAACATACTGACTTTTCAAGTGAAATCATAGGATACAAGGCTATTGATGGTGGTGTCACTCGTGACATAGCATCTACAGA
TACTTGTTTTGCTAACAAACATGCTGATTTTGACACATGGTTTAGCCAGCGTGGTGGTAGTTATACTAATGACAAAGCTTGCCC
ATTGATTGCTGCAGTCATAACAAGAGAAGTGGGTTTTGTCGTGCCTGGTTTGCCTGGCACGATATTACGCACAACTAATGGTGA
CTTTTTGCATTTCTTACCTAGAGTTTTTAGTGCAGTTGGTAACATCTGTTACACACCATCAAAACTTATAGAGTACACTGACTT
TGCAACATCAGCTTGTGTTTTGGCTGCTGAATGTACAATTTTTAAAGATGCTTCTGGTAAGCCAGTACCATATTGTTATGATAC
CAATGTACTAGAAGGTTCTGTTGCTTATGAAAGTTTACGCCCTGACACACGTTATGTGCTCATGGATGGCTCTATTATTCAATT
TCCTAACACCTACCTTGAAGGTTCTGTTAGAGTGGTAACAACTTTTGATTCTGAGTACTGTAGGCACGGCACTTGTGAAAGATC
AGAAGCTGGTGTTTGTGTATCTACTAGTGGTAGATGGGTACTTAACAATGATTATTACAGATCTTTACCAGGAGTTTTCTGTGG
TGTAGATGCTGTAAATTTACTTACTAATATGTTTACACCACTAATTCAACCTATTGGTGCTTTGGACATATCAGCATCTATAGT
AGCTGGTGGTATTGTAGCTATCGTAGTAACATGCCTTGCCTACTATTTTATGAGGTTTAGAAGAGCTTTTGGTGAATACAGTCA
TGTAGTTGCCTTTAATACTTTACTATTCCTTATGTCATTCACTGTACTCTGTTTAACACCAGTTTACTCATTCTTACCTGGTGT
TTATTCTGTTATTTACTTGTACTTGACATTTTATCTTACTAATGATGTTTCTTTTTTAGCACATATTCAGTGGATGGTTATGTT
CACACCTTTAGTACCTTTCTGGATAACAATTGCTTATATCATTTGTATTTCCACAAAGCATTTCTATTGGTTCTTTAGTAATTA
CCTAAAGAGACGTGTAGTCTTTAATGGTGTTTCCTTTAGTACTTTTGAAGAAGCTGCGCTGTGCACCTTTTTGTTAAATAAAGA
AATGTATCTAAAGTTGCGTAGTGATGTGCTATTACCTCTTACGCAATATAATAGATACTTAGCTCTTTATAATAAGTACAAGTA
TTTTAGTGGAGCAATGGATACAACTAGCTACAGAGAAGCTGCTTGTTGTCATCTCGCAAAGGCTCTCAATGACTTCAGTAACTC
AGGTTCTGATGTTCTTTACCAACCACCACAAACCTCTATCACCTCAGCTGTTTTGCAGAGTGGTTTTAGAAAAATGGCATTCCC
ATCTGGTAAAGTTGAGGGTTGTATGGTACAAGTAACTTGTGGTACAACTACACTTAACGGTCTTTGGCTTGATGACGTAGTTTA
CTGTCCAAGACATGTGATCTGCACCTCTGAAGACATGCTTAACCCTAATTATGAAGATTTACTCATTCGTAAGTCTAATCATAA
TTTCTTGGTACAGGCTGGTAATGTTCAACTCAGGGTTATTGGACATTCTATGCAAAATTGTGTACTTAAGCTTAAGGTTGATAC
AGCCAATCCTAAGACACCTAAGTATAAGTTTGTTCGCATTCAACCAGGACAGACTTTTTCAGTGTTAGCTTGTTACAATGGTTC
ACCATCTGGTGTTTACCAATGTGCTATGAGGCCCAATTTCACTATTAAGGGTTCATTCCTTAATGGTTCATGTGGTAGTGTTGG
TTTTAACATAGATTATGACTGTGTCTCTTTTTGTTACATGCACCATATGGAATTACCAACTGGAGTTCATGCTGGCACAGACTT
AGAAGGTAACTTTTATGGACCTTTTGTTGACAGGCAAACAGCACAAGCAGCTGGTACGGACACAACTATTACAGTTAATGTTTT
AGCTTGGTTGTACGCTGCTGTTATAAATGGAGACAGGTGGTTTCTCAATCGATTTACCACAACTCTTAATGACTTTAACCTTGT
GGCTATGAAGTACAATTATGAACCTCTAACACAAGACCATGTTGACATACTAGGACCTCTTTCTGCTCAAACTGGAATTGCCGT
TTTAGATATGTGTGCTTCATTAAAAGAATTACTGCAAAATGGTATGAATGGACGTACCATATTGGGTAGTGCTTTATTAGAAGA
TGAATTTACACCTTTTGATGTTGTTAGACAATGCTCAGGTGTTACTTTCCAAAGTGCAGTGAAAAGAACAATCAAGGGTACACA
CCACTGGTTGTTACTCACAATTTTGACTTCACTTTTAGTTTTAGTCCAGAGTACTCAATGGTCTTTGTTCTTTTTTTTGTATGA
AAATGCCTTTTTACCTTTTGCTATGGGTATTATTGCTATGTCTGCTTTTGCAATGATGTTTGTCAAACATAAGCATGCATTTCT
CTGTTTGTTTTTGTTACCTTCTCTTGCCACTGTAGCTTATTTTAATATGGTCTATATGCCTGCTAGTTGGGTGATGCGTATTAT
GACATGGTTGGATATGGTTGATACTAGTTTGTCTGGTTTTAAGCTAAAAGACTGTGTTATGTATGCATCAGCTGTAGTGTTACT
AATCCTTATGACAGCAAGAACTGTGTATGATGATGGTGCTAGGAGAGTGTGGACACTTATGAATGTCTTGACACTCGTTTATAA
AGTTTATTATGGTAATGCTTTAGATCAAGCCATTTCCATGTGGGCTCTTATAATCTCTGTTACTTCTAACTACTCAGGTGTAGT
TACAACTGTCATGTTTTTGGCCAGAGGTATTGTTTTTATGTGTGTTGAGTATTGCCCTATTTTCTTCATAACTGGTAATACACT
TCAGTGTATAATGCTAGTTTATTGTTTCTTAGGCTATTTTTGTACTTGTTACTTTGGCCTCTTTTGTTTACTCAACCGCTACTT
TAGACTGACTCTTGGTGTTTATGATTACTTAGTTTCTACACAGGAGTTTAGATATATGAATTCACAGGGACTACTCCCACCCAA
GAATAGCATAGATGCCTTCAAACTCAACATTAAATTGTTGGGTGTTGGTGGCAAACCTTGTATCAAAGTAGCCACTGTACAGTC
TAAAATGTCAGATGTAAAGTGCACATCAGTAGTCTTACTCTCAGTTTTGCAACAACTCAGAGTAGAATCATCATCTAAATTGTG
GGCTCAATGTGTCCAGTTACACAATGACATTCTCTTAGCTAAAGATACTACTGAAGCCTTTGAAAAAATGGTTTCACTACTTTC
TGTTTTGCTTTCCATGCAGGGTGCTGTAGACATAAACAAGCTTTGTGAAGAAATGCTGGACAACAGGGCAACCTTACAAGCTAT
AGCCTCAGAGTTTAGTTCCCTTCCATCATATGCAGCTTTTGCTACTGCTCAAGAAGCTTATGAGCAGGCTGTTGCTAATGGTGA
TTCTGAAGTTGTTCTTAAAAAGTTGAAGAAGTCTTTGAATGTGGCTAAATCTGAATTTGACCGTGATGCAGCCATGCAACGTAA
GTTGGAAAAGATGGCTGATCAAGCTATGACCCAAATGTATAAACAGGCTAGATCTGAGGACAAGAGGGCAAAAGTTACTAGTGC
TATGCAGACAATGCTTTTCACTATGCTTAGAAAGTTGGATAATGATGCACTCAACAACATTATCAACAATGCAAGAGATGGTTG
TGTTCCCTTGAACATAATACCTCTTACAACAGCAGCCAAACTAATGGTTGTCATACCAGACTATAACACATATAAAAATACGTG
TGATGGTACAACATTTACTTATGCATCAGCATTGTGGGAAATCCAACAGGTTGTAGATGCAGATAGTAAAATTGTTCAACTTAG
TGAAATTAGTATGGACAATTCACCTAATTTAGCATGGCCTCTTATTGTAACAGCTTTAAGGGCCAATTCTGCTGTCAAATTACA
GAATAATGAGCTTAGTCCTGTTGCACTACGACAGATGTCTTGTGCTGCCGGTACTACACAAACTGCTTGCACTGATGACAATGC
GTTAGCTTACTACAACACAACAAAGGGAGGTAGGTTTGTACTTGCACTGTTATCCGATTTACAGGATTTGAAATGGGCTAGATT
CCCTAAGAGTGATGGAACTGGTACTATCTATACAGAACTGGAACCACCTTGTAGGTTTGTTACAGACACACCTAAAGGTCCTAA
AGTGAAGTATTTATACTTTATTAAAGGATTAAACAACCTAAATAGAGGTATGGTACTTGGTAGTTTAGCTGCCACAGTACGTCT
ACAAGCTGGTAATGCAACAGAAGTGCCTGCCAATTCAACTGTATTATCTTTCTGTGCTTTTGCTGTAGATGCTGCTAAAGCTTA
CAAAGATTATCTAGCTAGTGGGGGACAACCAATCACTAATTGTGTTAAGATGTTGTGTACACACACTGGTACTGGTCAGGCAAT
AACAGTTACACCGGAAGCCAATATGGATCAAGAATCCTTTGGTGGTGCATCGTGTTGTCTGTACTGCCGTTGCCACATAGATCA
TCCAAATCCTAAAGGATTTTGTGACTTAAAAGGTAAGTATGTACAAATACCTACAACTTGTGCTAATGACCCTGTGGGTTTTAC
ACTTAAAAACACAGTCTGTACCGTCTGCGGTATGTGGAAAGGTTATGGCTGTAGTTGTGATCAACTCCGCGAACCCATGCTTCA
GTCAGCTGATGCACAATCGTTTTTAAACGGGTTTGCGGTGTAAGTGCAGCCCGTCTTACACCGTGCGGCACAGGCACTAGTACT

GATGTCGTATACAGGGCTTTTGACATCTACAATGATAAAGTAGCTGGTTTTGCTAAATTCCTAAAAACTAATTGTTGTCGCTTC
CAAGAAAAGGACGAAGATGACAATTTAATTGATTCTTACTTTGTAGTTAAGAGACACACTTTCTCTAACTACCAACATGAAGAA
ACAATTTATAATTTACTTAAGGATTGTCCAGCTGTTGCTAAACATGACTTCTTTAAGTTTAGAATAGACGGTGACATGGTACCA
CATATATCACGTCAACGTCTTACTAAATACACAATGGCAGACCTCGTCTATGCTTTAAGGCATTTTGATGAAGGTAATTGTGAC
ACATTAAAAGAAATACTTGTCACATACAATTGTTGTGATGATGATTATTTCAATAAAAAGGACTGGTATGATTTTGTAGAAAAC
CCAGATATATTACGCGTATACGCCAACTTAGGTGAACGTGTACGCCAAGCTTTGTTAAAAACAGTACAATTCTGTGATGCCATG
CGAAATGCTGGTATTGTTGGTGTACTGACATTAGATAATCAAGATCTCAATGGTAACTGGTATGATTTCGGTGATTTCATACAA
ACCACGCCAGGTAGTGGAGTTCCTGTTGTAGATTCTTATTATTCATTGTTAATGCCTATATTAACCTTGACCAGGGCTTTAACT
GCAGAGTCACATGTTGACACTGACTTAACAAAGCCTTACATTAAGTGGGATTTGTTAAAATATGACTTCACGGAAGAGAGGTTA
AAACTCTTTGACCGTTATTTTAAATATTGGGATCAGACATACCACCCAAATTGTGTTAACTGTTTGGATGACAGATGCATTCTG
CATTGTGCAAACTTTAATGTTTTATTCTCTACAGTGTTCCCACCTACAAGTTTTGGACCACTAGTGAGAAAAATATTTGTTGAT
GGTGTTCCATTTGTAGTTTCAACTGGATACCACTTCAGAGAGCTAGGTGTTGTACATAATCAGGATGTAAACTTACATAGCTCT
AGACTTAGTTTTAAGGAATTACTTGTGTATGCTGCTGACCCTGCTATGCACGCTGCTTCTGGTAATCTATTACTAGATAAACGC
ACTACGTGCTTTTCAGTAGCTGCACTTACTAACAATGTTGCTTTTCAAACTGTCAAACCCGGTAATTTTAACAAAGACTTCTAT
GACTTTGCTGTGTCTAAGGGTTTCTTTAAGGAAGGAAGTTCTGTTGAATTAAAACACTTCTTCTTTGCTCAGGATGGTAATGCT
GCTATCAGCGATTATGACTACTATCGTTATAATCTACCAACAATGTGTGATATCAGACAACTACTATTTGTAGTTGAAGTTGTT
GATAAGTACTTTGATTGTTACGATGGTGGCTGTATTAATGCTAACCAAGTCATCGTCAACAACCTAGACAAATCAGCTGGTTTT
CCATTTAATAAATGGGGTAAGGCTAGACTTTATTATGATTCAATGAGTTATGAGGATCAAGATGCACTTTTCGCATATACAAAA
CGTAATGTCATCCCTACTATAACTCAAATGAATCTTAAGTATGCCATTAGTGCAAAGAATAGAGCTCGCACCGTAGCTGGTGTC
TCTATCTGTAGTACTATGACCAATAGACAGTTTCATCAAAAATTATTGAAATCAATAGCCGCCACTAGAGGAGCTACTGTAGTA
ATTGGAACAAGCAAATTCTATGGTGGTTGGCACAACATGTTAAAAACTGTTTATAGTGATGTAGAAAACCCTCACCTTATGGGT
TGGGATTATCCTAAATGTGATAGAGCCATGCCTAACATGCTTAGAATTATGGCCTCACTTGTTCTTGCTCGCAAACATACAACG
TGTTGTAGCTTGTCACACCGTTTCTATAGATTAGCTAATGAGTGTGCTCAAGTATTGAGTGAAATGGTCATGTGTGGCGGTTCA
CTATATGTTAAACCAGGTGGAACCTCATCAGGAGATGCCACAACTGCTTATGCTAATAGTGTTTTTAACATTTGTCAAGCTGTC
ACGGCCAATGTTAATGCACTTTTATCTACTGATGGTAACAAAATTGCCGATAAGTATGTCCGCAATTTACAACACAGACTTTAT
GAGTGTCTCTATAGAAATAGAGATGTTGACACAGACTTTGTGAATGAGTTTTACGCATATTTGCGTAAACATTTCTCAATGATG
ATACTCTCTGACGATGCTGTTGTGTGTTTCAATAGCACTTATGCATCTCAAGGTCTAGTGGCTAGCATAAAGAACTTTAAGTCA
GTTCTTTATTATCAAAACAATGTTTTTATGTCTGAAGCAAAATGTTGGACTGAGACTGACCTTACTAAAGGACCTCATGAATTT
TGCTCTCAACATACAATGCTAGTTAAACAGGGTGATGATTATGTGTACCTTCCTTACCCAGATCCATCAAGAATCCTAGGGGCC
GGCTGTTTTGTAGATGATATCGTAAAAACAGATGGTACACTTATGATTGAACGGTTCGTGTCTTTAGCTATAGATGCTTACCCA
CTTACTAAACATCCTAATCAGGAGTATGCTGATGTCTTTCATTTGTACTTACAATACATAAGAAAGCTACATGATGAGTTAACA
GGACACATGTTAGACATGTATTCTGTTATGCTTACTAATGATAACACTTCAAGGTATTGGGAACCTGAGTTTTATGAGGCTATG
TACACACCGCATACAGTCTTACAGGCTGTTGGGGCTTGTGTTCTTTGCAATTCACAGACTTCATTAAGATGTGGTGCTTGCATA
CGTAGACCATTCTTATGTTGTAAATGCTGTTACGACCATGTCATATCAACATCACATAAATTAGTCTTGTCTGTTAATCCGTAT
GTTTGCAATGCTCCAGGTTGTGATGTCACAGATGTGACTCAACTTTACTTAGGAGGTATGAGCTATTATTGTAAATCACATAAA
CCACCCATTAGTTTTCCATTGTGTGCTAATGGACAAGTTTTTGGTTTATATAAAAATACATGTGTTGGTAGCGATAATGTTACT
GACTTTAATGCAATTGCAACATGTGACTGGACAAATGCTGGTGATTACATTTTAGCTAACACCTGTACTGAAAGACTCAAGCTT
TTTGCAGCAGAAACGCTCAAAGCTACTGAGGAGACATTTAAACTGTCTTATGGTATTGCTACTGTACGTGAAGTGCTGTCTGAC
AGAGAATTACATCTTTCATGGGAAGTTGGTAAACCTAGACCACCACTTAACCGAAATTATGTCTTTACTGGTTATCGTGTAACT
AAAAACAGTAAAGTACAAATAGGAGAGTACACCTTTGAAAAAGGTGACTATGGTGATGCTGTTGTTTACCGAGGTACAACAACT
TACAAATTAAATGTTGGTGATTATTTTGTGCTGACATCACATACAGTAATGCCATTAAGTGCACCTACACTAGTGCCACAAGAG
CACTATGTTAGAATTACTGGCTTATACCCAACACTCAATATCTCAGATGAGTTTTCTAGCAATGTTGCAAATTATCAAAAGGTT
GGTATGCAAAAGTATTCTACACTCCAGGGACCACCTGGTACTGGTAAGAGTCATTTTGCTATTGGCCTAGCTCTCTACTACCCT
TCTGCTCGCATAGTGTATACAGCTTGCTCTCATGCCGCTGTTGATGCACTATGTGAGAAGGCATTAAAATATTTGCCTATAGAT
AAATGTAGTAGAATTATACCTGCACGTGCTCGTGTAGAGTGTTTTGATAAATTCAAAGTGAATTCAACATTAGAACAGTATGTC
TTTTGTACTGTAAATGCATTGCCTGAGACGACAGCAGATATAGTTGTCTTTGATGAAATTTCAATGGCCACAAATTATGATTTG
AGTGTTGTCAATGCCAGATTACGTGCTAAGCACTATGTGTACATTGGCGACCCTGCTCAATTACCTGCACCACGCACATTGCTA
ACTAAGGGCACACTAGAACCAGAATATTTCAATTCAGTGTGTAGACTTATGAAAACTATAGGTCCAGACATGTTCCTCGGAACT
TGTCGGCGTTGTCCTGCTGAAATTGTTGACACTGTGAGTGCTTTGGTTTATGATAATAAGCTTAAAGCACATAAAGACAAATCA
GCTCAATGCTTTAAAATGTTTTATAAGGGTGTTATCACGCATGATGTTTCATCTGCAATTAACAGGCCACAAATAGGCGTGGTA
AGAGAATTCCTTACACGTAACCCTGCTTGGAGAAAAGCTGTCTTTATTTCACCTTATAATTCACAGAATGCTGTAGCCTCAAAG
ATTTTGGGACTACCAACTCAAACTGTTGATTCATCACAGGGCTCAGAATATGACTATGTCATATTCACTCAAACCACTGAAACA
GCTCACTCTTGTAATGTAAACAGATTTAATGTTGCTATTACCAGAGCAAAAGTAGGCATACTTTGCATAATGTCTGATAGAGAC
CTTTATGACAAGTTGCAATTTACAAGTCTTGAAATTCCACGTAGGAATGTGGCAACTTTACAAGCTGAAAATGTAACAGGACTC
TTTAAAGATTGTAGTAAGGTAATCACTGGGTTACATCCTACACAGGCACCTACACACCTCAGTGTTGACACTAAATTCAAAACT
GAAGGTTTATGTGTTGACATACCTGGCATACCTAAGGACATGACCTATAGAAGACTCATCTCTATGATGGGTTTTAAAATGAAT
TATCAAGTTAATGGTTACCCTAACATGTTTATCACCCGCGAAGAAGCTATAAGACATGTACGTGCATGGATTGGCTTCGATGTC
GAGGGGTGTCATGCTACTAGAGAAGCTGTTGGTACCAATTTACCTTTACAGCTAGGTTTTTCTACAGGTGTTAACCTAGTTGCT
GTACCTACAGGTTATGTTGATACACCTAATAATACAGATTTTTCCAGAGTTAGTGCTAAACCACCGCCTGGAGATCAATTTAAA
CACCTCATACCACTTATGTACAAAGGACTTCCTTGGAATGTAGTGCGTATAAAGATTGTACAAATGTTAAGTGACACACTTAAA
AATCTCTCTGACAGAGTCGTATTTGTCTTATGGGCACATGGCTTTGAGTTGACATCTATGAAGTATTTTGTGAAAATAGGACCT

GAGCGCACCTGTTGTCTATGTGATAGACGTGCCACATGCTTTTCCACTGCTTCAGACACTTATGCCTGTTGGCATCATTCTATT
GGATTTGATTACGTCTATAATCCGTTTATGATTGATGTTCAACAATGGGGTTTTACAGGTAACCTACAAAGCAACCATGATCTG
TATTGTCAAGTCCATGGTAATGCACATGTAGCTAGTTGTGATGCAATCATGACTAGGTGTCTAGCTGTCCACGAGTGCTTTGTT
AAGCGTGTTGACTGGACTATTGAATATCCTATAATTGGTGATGAACTGAAGATTAATGCGGCTTGTAGAAAGGTTCAACACATG
GTTGTTAAAGCTGCATTATTAGCAGACAAATTCCCAGTTCTTCACGACATTGGTAACCCTAAAGCTATTAAGTGTGTACCTCAA
GCTGATGTAGAATGGAAGTTCTATGATGCACAGCCTTGTAGTGACAAAGCTTATAAAATAGAAGAATTATTCTATTCTTATGCC
ACACATTCTGACAAATTCACAGATGGTGTATGCCTATTTTGGAATTGCAATGTCGATAGATATCCTGCTAATTCCATTGTTTGT
AGATTTGACACTAGAGTGCTATCTAACCTTAACTTGCCTGGTTGTGATGGTGGCAGTTTGTATGTAAATAAACATGCATTCCAC
ACACCAGCTTTTGATAAAAGTGCTTTTGTTAATTTAAAACAATTACCATTTTTCTATTACTCTGACAGTCCATGTGAGTCTCAT
GGAAAACAAGTAGTGTCAGATATAGATTATGTACCACTAAAGTCTGCTACGTGTATAACACGTTGCAATTTAGGTGGTGCTGTC
TGTAGACATCATGCTAATGAGTACAGATTGTATCTCGATGCTTATAACATGATGATCTCAGCTGGCTTTAGCTTGTGGGTTTAC
AAACAATTTGATACTTATAACCTCTGGAACACTTTTACAAGACTTCAGAGTTTAGAAAATGTGGCTTTTAATGTTGTAAATAAG
GGACACTTTGATGGACAACAGGGTGAAGTACCAGTTTCTATCATTAATAACACTGTTTACACAAAAGTTGATGGTGTTGATGTA
GAATTGTTTGAAAATAAAACAACATTACCTGTTAATGTAGCATTTGAGCTTTGGGCTAAGCGCAACATTAAACCAGTACCAGAG
GTGAAAATACTCAATAATTTGGGTGTGGACATTGCTGCTAATACTGTGATCTGGGACTACAAAAGAGATGCTCCAGCACATATA
TCTACTATTGGTGTTTGTTCTATGACTGACATAGCCAAGAAACCAACTGAAACGATTTGTGCACCACTCACTGTCTTTTTTGAT
GGTAGAGTTGATGGTCAAGTAGACTTATTTAGAAATGCCCGTAATGGTGTTCTTATTACAGAAGGTAGTGTTAAAGGTTTACAA
CCATCTGTAGGTCCCAAACAAGCTAGTCTTAATGGAGTCACATTAATTGGAGAAGCCGTAAAAACACAGTTCAATTATTATAAG
AAAGTTGATGGTGTTGTCCAACAATTACCTGAAACTTACTTTACTCAGAGTAGAAATTTACAAGAATTTAAACCCAGGAGTCAA
ATGGAAATTGATTTCTTAGAATTAGCTATGGATGAATTCATTGAACGGTATAAATTAGAAGGCTATGCCTTCGAACATATCGTT
TATGGAGATTTTAGTCATAGTCAGTTAGGTGGTTTACATCTACTGATTGGACTAGCTAAACGTTTTAAGGAATCACCTTTTGAA
TTAGAAGATTTTATTCCTATGGACAGTACAGTTAAAAACTATTTCATAACAGATGCGCAAACAGGTTCATCTAAGTGTGTGTGT
TCTGTTATTGATTTATTACTTGATGATTTTGTTGAAATAATAAAATCCCAAGATTTATCTGTAGTTTCTAAGGTTGTCAAAGTG
ACTATTGACTATACAGAAATTTCATTTATGCTTTGGTGTAAAGATGGCCATGTAGAAACATTTTACCCAAAATTACAATCTAGT
CAAGCGTGGCAACCGGGTGTTGCTATGCCTAATCTTTACAAAATGCAAAGAATGCTATTAGAAAAGTGTGACCTTCAAAATTAT
GGTGATAGTGCAACATTACCTAAAGGCATAATGATGAATGTCGCAAAATATACTCAACTGTGTCAATATTTAAACACATTAACA
TTAGCTGTACCCTATAATATGAGAGTTATACATTTTGGTGCTGGTTCTGATAAAGGAGTTGCACCAGGTACAGCTGTTTTAAGA
CAGTGGTTGCCTACGGGTACGCTGCTTGTCGATTCAGATCTTAATGACTTTGTCTCTGATGCAGATTCAACTTTGATTGGTGAT
TGTGCAACTGTACATACAGCTAATAAATGGGATCTCATTATTAGTGATATGTACGACCCTAAGACTAAAAATGTTACAAAAGAA
AATGACTCTAAAGAGGGTTTTTTCACTTACATTTGTGGGTTTATACAACAAAAGCTAGCTCTTGGAGGTTCCGTGGCTATAAAG
ATAACAGAACATTCTTGGAATGCTGATCTTTATAAGCTCATGGGACACTTCGCATGGTGGACAGCCTTTGTTACTAATGTGAAT
GCGTCATCATCTGAAGCATTTTTAATTGGATGTAATTATCTTGGCAAACCACGCGAACAAATAGATGGTTATGTCATGCATGCA
AATTACATATTTTGGAGGAATACAAATCCAATTCAGTTGTCTTCCTATTCTTTATTTGACATGAGTAAATTTCCCCTTAAATTA
AGGGGTACTGCTGTTATGTCTTTAAAAGAAGGTCAAATCAATGATATGATTTTATCTCTTCTTAGTAAAGGTAGACTTATAATT
AGAGAAAACAACAGAGTTGTTATTTCTAGTGATGTTCTTGTTAACAACTAAACGAACAATGTTTGTTTTTCTTGTTTTATTGCC
ACTAGTCTCTAGTCAGTGTGTTAATCTTACAACCAGAACTCAATTACCCCCTGCATACACTAATTCTTTCACACGTGGTGTTTA
TTACCCTGACAAAGTTTTCAGATCCTCAGTTTTACATTCAACTCAGGACTTGTTCTTACCTTTCTTTTCCAATGTTACTTGGTT
CCATGCTATACATGTCTCTGGGACCAATGGTACTAAGAGGTTTGATAACCCTGTCCTACCATTTAATGATGGTGTTTATTTTGC
TTCCACTGAGAAGTCTAACATAATAAGAGGCTGGATTTTTGGTACTACTTTAGATTCGAAGACCCAGTCCCTACTTATTGTTAA
TAACGCTACTAATGTTGTTATTAAAGTCTGTGAATTTCAATTTTGTAATGATCCATTTTTGGGTGTTTATTACCACAAAAACAA
CAAAAGTTGGATGGAAAGTGAGTTCAGAGTTTATTCTAGTGCGAATAATTGCACTTTTGAATATGTCTCTCAGCCTTTTCTTAT
GGACCTTGAAGGAAAACAGGGTAATTTCAAAAATCTTAGGGAATTTGTGTTTAAGAATATTGATGGTTATTTTAAAATATATTC
TAAGCACACGCCTATTAATTTAGTGCGTGATCTCCCTCAGGGTTTTTCGGCTTTAGAACCATTGGTAGATTTGCCAATAGGTAT
TAACATCACTAGGTTTCAAACTTTACTTGCTTTACATAGAAGTTATTTGACTCCTGGTGATTCTTCTTCAGGTTGGACAGCTGG
TGCTGCAGCTTATTATGTGGGTTATCTTCAACCTAGGACTTTTCTATTAAAATATAATGAAAATGGAACCATTACAGATGCTGT
AGACTGTGCACTTGACCCTCTCTCAGAAACAAAGTGTACGTTGAAATCCTTCACTGTAGAAAAAGGAATCTATCAAACTTCTAA
CTTTAGAGTCCAACCAACAGAATCTATTGTTAGATTTCCTAATATTACAAACTTGTGCCCTTTTGGTGAAGTTTTTAACGCCAC
CAGATTTGCATCTGTTTATGCTTGGAACAGGAAGAGAATCAGCAACTGTGTTGCTGATTATTCTGTCCTATATAATTCCGCATC
ATTTTCCACTTTTAAGTGTTATGGAGTGTCTCCTACTAAATTAAATGATCTCTGCTTTACTAATGTCTATGCAGATTCATTTGT
AATTAGAGGTGATGAAGTCAGACAAATCGCTCCAGGGCAAACTGGAAAGATTGCTGATTATAATTATAAATTACCAGATGATTT
TACAGGCTGCGTTATAGCTTGGAATTCTAACAATCTTGATTCTAAGGTTGGTGGTAATTATAATTACCTGTATAGATTGTTTAG
GAAGTCTAATCTCAAACCTTTTGAGAGAGATATTTCAACTGAAATCTATCAGGCCGGTAGCACACCTTGTAATGGTGTTGAAGG
TTTTAATTGTTACTTTCCTTTACAATCATATGGTTTCCAACCCACTAATGGTGTTGGTTACCAACCATACAGAGTAGTAGTACT
TTCTTTTGAACTTCTACATGCACCAGCAACTGTTTGTGGACCTAAAAAGTCTACTAATTTGGTTAAAAACAAATGTGTCAATTT
CAACTTCAATGGTTTAACAGGCACAGGTGTTCTTACTGAGTCTAACAAAAAGTTTCTGCCTTTCCAACAATTTGGCAGAGACAT
TGCTGACACTACTGATGCTGTCCGTGATCCACAGACACTTGAGATTCTTGACATTACACCATGTTCTTTTGGTGGTGTCAGTGT
TATAACACCAGGAACAAATACTTCTAACCAGGTTGCTGTTCTTTATCAGGATGTTAACTGCACAGAAGTCCCTGTTGCTATTCA
TGCAGATCAACTTACTCCTACTTGGCGTGTTTATTCTACAGGTTCTAATGTTTTTCAAACACGTGCAGGCTGTTTAATAGGGGC
TGAACATGTCAACAACTCATATGAGTGTGACATACCCATTGGTGCAGGTATATGCGCTAGTTATCAGACTCAGACTAATTCTCC
TCGGCGGGCACGTAGTGTAGCTAGTCAATCCATCATTGCCTACACTATGTCACTTGGTGCAGAAAATTCAGTTGCTTACTCTAA
TAACTCTATTGCCATACCCACAAATTTTACTATTAGTGTTACCACAGAAATTCTACCAGTGTCTATGACCAAGACATCAGTAGA

TTGTACAATGTACATTTGTGGTGATTCAACTGAATGCAGCAATCTTTTGTTGCAATATGGCAGTTTTTGTACACAATTAAACCG
TGCTTTAACTGGAATAGCTGTTGAACAAGACAAAAACACCCAAGAAGTTTTTGCACAAGTCAAACAAATTTACAAAACACCACC
AATTAAAGATTTTGGTGGTTTTAATTTTTCACAAATATTACCAGATCCATCAAAACCAAGCAAGAGGTCATTTATTGAAGATCT
ACTTTTCAACAAAGTGACACTTGCAGATGCTGGCTTCATCAAACAATATGGTGATTGCCTTGGTGATATTGCTGCTAGAGACCT
CATTTGTGCACAAAAGTTTAACGGCCTTACTGTTTTGCCACCTTTGCTCACAGATGAAATGATTGCTCAATACACTTCTGCACT
GTTAGCGGGTACAATCACTTCTGGTTGGACCTTTGGTGCAGGTGCTGCATTACAAATACCATTTGCTATGCAAATGGCTTATAG
GTTTAATGGTATTGGAGTTACACAGAATGTTCTCTATGAGAACCAAAAATTGATTGCCAACCAATTTAATAGTGCTATTGGCAA
AATTCAAGACTCACTTTCTTCCACAGCAAGTGCACTTGGAAAACTTCAAGATGTGGTCAACCAAAATGCACAAGCTTTAAACAC
GCTTGTTAAACAACTTAGCTCCAATTTTGGTGCAATTTCAAGTGTTTTAAATGATATCCTTTCACGTCTTGACAAAGTTGAGGC
TGAAGTGCAAATTGATAGGTTGATCACAGGCAGACTTCAAAGTTTGCAGACATATGTGACTCAACAATTAATTAGAGCTGCAGA
AATCAGAGCTTCTGCTAATCTTGCTGCTACTAAAATGTCAGAGTGTGTACTTGGACAATCAAAAAGAGTTGATTTTTGTGGAAA
GGGCTATCATCTTATGTCCTTCCCTCAGTCAGCACCTCATGGTGTAGTCTTCTTGCATGTGACTTATGTCCCTGCACAAGAAAA
GAACTTCACAACTGCTCCTGCCATTTGTCATGATGGAAAAGCACACTTTCCTCGTGAAGGTGTCTTTGTTTCAAATGGCACACA
CTGGTTTGTAACACAAAGGAATTTTTATGAACCACAAATCATTACTACAGACAACACATTTGTGTCTGGTAACTGTGATGTTGT
AATAGGAATTGTCAACAACACAGTTTATGATCCTTTGCAACCTGAATTAGACTCATTCAAGGAGGAGTTAGATAAATATTTTAA
GAATCATACATCACCAGATGTTGATTTAGGTGACATCTCTGGCATTAATGCTTCAGTTGTAAACATTCAAAAAGAAATTGACCG
CCTCAATGAGGTTGCCAAGAATTTAAATGAATCTCTCATCGATCTCCAAGAACTTGGAAAGTATGAGCAGTATATAAAATGGCC
ATGGTACATTTGGCTAGGTTTTATAGCTGGCTTGATTGCCATAGTAATGGTGACAATTATGCTTTGCTGTATGACCAGTTGCTG
TAGTTGTCTCAAGGGCTGTTGTTCTTGTGGATCCTGCTGCAAATTTGATGAAGACGACTCTGAGCCAGTGCTCAAAGGAGTCAA
ATTACATTACACATAAACGAACTTATGGATTTGTTTATGAGAATCTTCACAATTGGAACTGTAACTTTGAAGCAAGGTGAAATC
AAGGATGCTACTCCTTCAGATTTTGTTCGCGCTACTGCAACGATACCGATACAAGCCTCACTCCCTTTCGGATGGCTTATTGTT
GGCGTTGCACTTCTTGCTGTTTTTCAGAGCGCTTCCAAAATCATAACCCTCAAAAAGAGATGGCAACTAGCACTCTCCAAGGGT
GTTCACTTTGTTTGCAACTTGCTGTTGTTGTTTGTAACAGTTTACTCACACCTTTTGCTCGTTGCTGCTGGCCTTGAAGCCCCT
TTTCTCTATCTTTATGCTTTAGTCTACTTCTTGCAGAGTATAAACTTTGTAAGAATAATAATGAGGCTTTGGCTTTGCTGGAAA
TGCCGTTCCAAAAACCCATTACTTTATGATGCCAACTATTTTCTTTGCTGGCATACTAATTGTTACGACTATTGTATACCTTAC
AATAGTGTAACTTCTTCAATTGTCATTACTTCAGGTGATGGCACAACAAGTCCTATTTCTGAACATGACTACCAGATTGGTGGT
TATACTGAAAAATGGGAATCTGGAGTAAAAGACTGTGTTGTATTACACAGTTACTTCACTTCAGACTATTACCAGCTGTACTCA
ACTCAATTGAGTACAGACACTGGTGTTGAACATGTTACCTTCTTCATCTACAATAAAATTGTTGATGAGCCTGAAGAACATGTC
CAAATTCACACAATCGACGGTTCATCCGGAGTTGTTAATCCAGTAATGGAACCAATTTATGATGAACCGACGACGACTACTAGC
GTGCCTTTGTAAGCACAAGCTGATGAGTACGAACTTATGTACTCATTCGTTTCGGAAGAGACAGGTACGTTAATAGTTAATAGC
GTACTTCTTTTTCTTGCTTTCGTGGTATTCTTGCTAGTTACACTAGCCATCCTTACTGCGCTTCGATTGTGTGCGTACTGCTGC
AATATTGTTAACGTGAGTCTTGTAAAACCTTCTTTTTACGTTTACTCTCGTGTTAAAAATCTGAATTCTTCTAGAGTTCCTGAT
CTTCTGGTCTAAACGAACTAAATATTATATTAGTTTTTCTGTTTGGAACTTTAATTTTAGCCATGGCAGATTCCAACGGTACTA
TTACCGTTGAAGAGCTTAAAAAGCTCCTTGAACAATGGAACCTAGTAATAGGTTTCCTATTCCTTACATGGATTTGTCTTCTAC
AATTTGCCTATGCCAACAGGAATAGGTTTTTGTATATAATTAAGTTAATTTTCCTCTGGCTGTTATGGCCAGTAACTTTAGCTT
GTTTTGTGCTTGCTGCTGTTTACAGAATAAATTGGATCACCGGTGGAATTGCTATCGCAATGGCTTGTCTTGTAGGCTTGATGT
GGCTCAGCTACTTCATTGCTTCTTTCAGACTGTTTGCGCGTACGCGTTCCATGTGGTCATTCAATCCAGAAACTAACATTCTTC
TCAACGTGCCACTCCATGGCACTATTCTGACCAGACCGCTTCTAGAAAGTGAACTCGTAATCGGAGCTGTGATCCTTCGTGGAC
ATCTTCGTATTGCTGGACACCATCTAGGACGCTGTGACATCAAGGACCTGCCTAAAGAAATCACTGTTGCTACATCACGAACGC
TTTCTTATTACAAATTGGGAGCTTCGCAGCGTGTAGCAGGTGACTCAGGTTTTGCTGCATACAGTCGCTACAGGATTGGCAACT
ATAAATTAAACACAGACCATTCCAGTAGCAGTGACAATATTGCTTTGCTTGTACAGTAAGTGACAACAGATGTTTCATCTCGTT
GACTTTCAGGTTACTATAGCAGAGATATTACTAATTATTATGAGGACTTTTAAAGTTTCCATTTGGAATCTTGATTACATCATA
AACCTCATAATTAAAAATTTATCTAAGTCACTAACTGAGAATAAATATTCTCAATTAGATGAAGAGCAACCAATGGAGATTGAT
TAAACGAACATGAAAATTATTCTTTTCTTGGCACTGATAACACTCGCTACTTGTGAGCTTTATCACTACCAAGAGTGTGTTAGA
GGTACAACAGTACTTTTAAAAGAACCTTGCTCTTCTGGAACATACGAGGGCAATTCACCATTTCATCCTCTAGCTGATAACAAA
TTTGCACTGACTTGCTTTAGCACTCAATTTGCTTTTGCTTGTCCTGACGGCGTAAAACACGTCTATCAGTTACGTGCCAGATCA
GTTTCACCTAAACTGTTCATCAGACAAGAGGAAGTTCAAGAACTTTACTCTCCAATTTTTCTTATTGTTGCGGCAATAGTGTTT
ATAACACTTTGCTTCACACTCAAAAGAAAGACAGAATGATTGAACTTTCATTAATTGACTTCTATTTGTGCTTTTTAGCCTTTC
TGCTATTCCTTGTTTTAATTATGCTTATTATCTTTTGGTTCTCACTTGAACTGCAAGATCATAATGAAACTTGTCACGCCTAAA
CGAACATGAAATTTCTTGTTTTCTTAGGAATCATCACAACTGTAGCTGCATTTCACCAAGAATGTAGTTTACAGTCATGTACTC
AACATCAACCATATGTAGTTGATGACCCGTGTCCTATTCACTTCTATTCTAAATGGTATATTAGAGTAGGAGCTAGAAAATCAG
CACCTTTAATTGAATTGTGCGTGGATGAGGCTGGTTCTAAATCACCCATTCAGTACATCGATATCGGTAATTATACAGTTTCCT
GTTTACCTTTTACAATTAATTGCCAGGAACCTAAATTGGGTAGTCTTGTAGTGCGTTGTTCGTTCTATGAAGACTTTTTAGAGT
ATCATGACGTTCGTGTTGTTTTAGATTTCATCTAAACGAACAAACTAAAATGTCTGATAATGGACCCCAAAATCAGCGAAATGC
ACCCCGCATTACGTTTGGTGGACCCTCAGATTCAACTGGCAGTAACCAGAATGGAGAACGCAGTGGGGCGCGATCAAAACAACG
TCGGCCCCAAGGTTTACCCAATAATACTGCGTCTTGGTTCACCGCTCTCACTCAACATGGCAAGGAAGACCTTAAATTCCCTCG
AGGACAAGGCGTTCCAATTAACACCAATAGCAGTCCAGATGACCAAATTGGCTACTACCGAAGAGCTACCAGACGAATTCGTGG
TGGTGACGGTAAAATGAAAGATCTCAGTCCAAGATGGTATTTCTACTACCTAGGAACTGGGCCAGAAGCTGGACTTCCCTATGG
TGCTAACAAAGACGGCATCATATGGGTTGCAACTGAGGGAGCCTTGAATACACCAAAAGATCACATTGGCACCCGCAATCCTGC
TAACAATGCTGCAATCGTGCTACAACTTCCTCAAGGAACAACATTGCCAAAAGGCTTCTACGCAGAAGGGAGCAGAGGCGGCAG
TCAAGCCTCTTCTCGTTCCTCATCACGTAGTCGCAACAGTTCAAGAAATTCAACTCCAGGCAGCAGTAGGGGAACTTCTCCTGC

```
TAGAATGGCTGGCAATGGCGGTGATGCTGCTCTTGCTTTGCTGCTGCTTGACAGATTGAACCAGCTTGAGAGCAAAATGTCTGG
TAAAGGCCAACAACAACAAGGCCAAACTGTCACTAAGAAATCTGCTGCTGAGGCTTCTAAGAAGCCTCGGCAAAAACGTACTGC
CACTAAAGCATACAATGTAACACAAGCTTTCGGCAGACGTGGTCCAGAACAAACCCAAGGAAATTTTGGGGACCAGGAACTAAT
CAGACAAGGAACTGATTACAAACATTGGCCGCAAATTGCACAATTTGCCCCCAGCGCTTCAGCGTTCTTCGGAATGTCGCGCAT
TGGCATGGAAGTCACACCTTCGGGAACGTGGTTGACCTACACAGGTGCCATCAAATTGGATGACAAAGATCCAAATTTCAAAGA
TCAAGTCATTTTGCTGAATAAGCATATTGACGCATACAAAACATTCCCACCAACAGAGCCTAAAAAGGACAAAAAGAAGAAGGC
TGATGAAACTCAAGCCTTACCGCAGAGACAGAAGAAACAGCAAACTGTGACTCTTCTTCCTGCTGCAGATTTGGATGATTTCTC
CAAACAATTGCAACAATCCATGAGCAGTGCTGACTCAACTCAGGCCTAAACTCATGCAGACCACACAAGGCAGATGGGCTATAT
AAACGTTTTCGCTTTTCCGTTTACGATATATAGTCTACTCTTGTGCAGAATGAATTCTCGTAACTACATAGCACAAGTAGATGT
AGTTAACTTTAATCTCACATAGCAATCTTTAATCAGTGTGTAACATTAGGGAGGACTTGAAAGAGCCACCACATTTTCACCGAG
GCCACGCGGAGTACGATCGAGTGTACAGTGAACAATGCTAGGGAGAGCTGCCTATATGGAAGAGCCCTAATGTGTAAAATTAAT
TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Spike protein primer:

```
% DNA sequence that you know starts the gene
spike_FWprimer = 'ATGTTTGTTTTTCTTGTTTTATTG';
```

Codon information - reorganize into a three column cell array for easier accessibility:

```
% NOTE: one cell in codon_list (imported above) is a line of information
% containing 1) the nucleotides, the name, and abbreviation for
% each amino acid. We want each of these three items in a separate column
codon_list = importdata('codonList.txt');
disp(codon_list);
```

```
    {'TTT,Phenylalanine,F'}
    {'TTC,Phenylalanine,F'}
    {'TTA,Leucine,L'      }
    {'TTG,Leucine,L'      }
    {'TCT,Serine,S'       }
    {'TCC,Serine,S'       }
    {'TCA,Serine,S'       }
    {'TCG,Serine,S'       }
    {'TAT,Tyrosine,Y'     }
    {'TAC,Tyrosine,Y'     }
    {'TAA,Stop,X'         }
    {'TAG,Stop,X'         }
    {'TGT,Cysteine,C'     }
    {'TGC,Cysteine,C'     }
    {'TGA,Stop,X'         }
    {'TGG,Tryptophan,W'   }
    {'CTT,Leucine,L'      }
    {'CTC,Leucine,L'      }
    {'CTA,Leucine,L'      }
    {'CTG,Leucine,L'      }
    {'CCT,Proline,P'      }
    {'CCC,Proline,P'      }
    {'CCA,Proline,P'      }
    {'CCG,Proline,P'      }
    {'CAT,Histidine,H'    }
    {'CAC,Histidine,H'    }
    {'CAA,Glutamine,Q'    }
    {'CAG,Glutamine,Q'    }
    {'CGT,Arginine,R'     }
```

```
    {'CGC,Arginine,R'     }
    {'CGA,Arginine,R'     }
    {'CGG,Arginine,R'     }
    {'ATT,Isoleucine,I'   }
    {'ATC,Isoleucine,I'   }
    {'ATA,Isoleucine,I'   }
    {'ATG,Methionine,M'   }
    {'ACT,Threonine,T'    }
    {'ACC,Threonine,T'    }
    {'ACA,Threonine,T'    }
    {'ACG,Threonine,T'    }
    {'AAT,Asparagine,N'   }
    {'AAC,Asparagine,N'   }
    {'AAA,Lysine,K'       }
    {'AAG,Lysine,K'       }
    {'AGT,Serine,S'       }
    {'AGC,Serine,S'       }
    {'AGA,Arginine,R'     }
    {'AGG,Arginine,R'     }
    {'GTT,Valine,V'       }
    {'GTC,Valine,V'       }
    {'GTA,Valine,V'       }
    {'GTG,Valine,V'       }
    {'GCT,Alanine,A'      }
    {'GCC,Alanine,A'      }
    {'GCA,Alanine,A'      }
    {'GCG,Alanine,A'      }
    {'GAT,Aspartate,D'    }
    {'GAC,Aspartate,D'    }
    {'GAA,Glutamate,E'    }
    {'GAG,Glutamate,E'    }
    {'GGT,Glycine,G'      }
    {'GGC,Glycine,G'      }
    {'GGA,Glycine,G'      }
    {'GGG,Glycine,G'      }
```

```
codons = cell(length(codon_list),3);

%process each line and seperate the elements

tempCodon = codon_list{k}
```

```
tempCodon = 'TAC,Tyrosine,Y'
```

```
commaInd = strfind(tempCodon, ',')
```

```
commaInd = 1×2
     4    13
```

```
for k = 1:length(codon_list)
    %First cell codon ATCG
    codons{k,1} = tempCodon(1:commaInd(1)-1);
    %second cell: name of amino acid
    codons{k,2} = tempCodon(1:commaInd(2)-1);
    %third cell: First cell codon ATCG
```

```
    codons{k,1} = tempCodon(commaInd(2)+1:end);

end
```

Note: the codons listed are given in terms of *DNA bases*.

Question 1: how many bases are in the SARS-CoV-2 genome?

```
% Number of bases equals number of elements in the character vector
numBases = length(sarsCov2_genome)
```

```
numBases = 29903
```

Question 2: how many of *each* base appear in the genome?

Note: for this problem we consider only one strand.

```
% Step 1: Determine how many of each base are in the genome
base_A = length(sarsCov2_genome(sarsCov2_genome == 'A'))
```

```
base_A = 8954
```

```
base_T = length(sarsCov2_genome(sarsCov2_genome == 'T'))
```

```
base_T = 9594
```

```
base_C = length(sarsCov2_genome(sarsCov2_genome == 'C'))
```

```
base_C = 5492
```

```
base_G = length(sarsCov2_genome(sarsCov2_genome == 'G'))
```

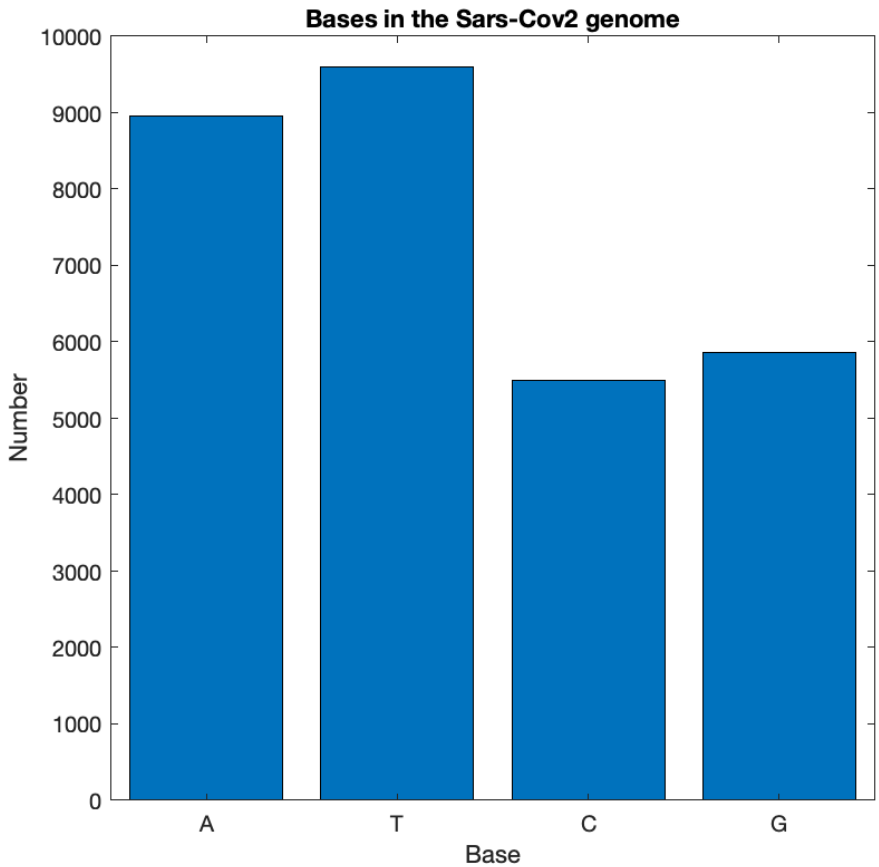
```
base_G = 5863
```

```
% Step 2: Plotting
% Create cell array containing each x label
xval = {'A', 'T', 'C', 'G'}
```

```
xval = 1×4 cell array
    {'A'}    {'T'}    {'C'}    {'G'}
```

```
%Convert cell array to categorical and then reorder to be in the same order
xval2 = categorical(xval);
xval2 = reordercats(xval2, xval)';
```

```
% Make bar graph
bar(xval2, [base_A, base_T, base_C, base_G]);
title('Bases in the Sars-Cov2 genome')
axis square
ylabel('Number');
xlabel('Base');
```



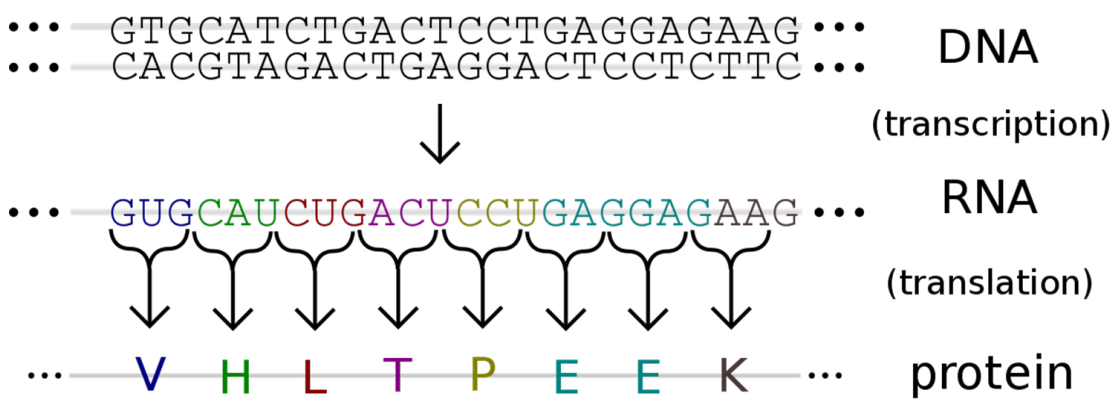
Question 3: given a primer, what is the DNA sequence for the spike glycoprotein? How many bases long is it?

TOPIC 17: Text and structure arrays

ENGR105, 10/29/20

Practice Problem: the central dogma of molecular biology

The **central dogma of molecular biology** is an explanation of the genetic flow of information in a biological system, often stated as "DNA makes RNA, and RNA makes protein." *Depicted visually (from* wikipedia*):*



Key terms for this problem:

- **DNA**: the "genetic blueprint" for life. Strands of DNA are biopolymers consisting of a sequence of monomers, known as nucleotides. Each nucleotide consists of one base (cytosine [**C**], guanine [**G**], adenine [**A**] or thymine [**T**]), a sugar (deoxyribose), and a phosphate group. **Information is stored in the sequence of bases**, and DNA sequences are represented as a sequence of A, T, C, and G. Example: `TCTTGTTTTATTG`
- **mRNA**: the "instructions copied from DNA" dictating how to construct a protein. Like DNA, strands of RNA are biopolymers consisting of a sequence of nucleotide monomers. In this case, each nucleotide consists of one base (cytosine [**C**], guanine [**G**], adenine [**A**] or uracil [**U**]), a sugar (ribose), and a phosphate group. An mRNA strand is **produced from its DNA template**, with all bases matching except with uracils replacing thymines. Information is stored in the sequence of bases, and mRNA sequences are represented as a sequence of A, U, C, and G. Example corresponding to the above: `UCUUGUUUUAUUG`
- **Protein**: the "functional units" produced from an mRNA template. Encoded from mRNA, proteins are biopolymers that consist of a sequence of amino acid monomers. There are 20 naturally occuring amino acids.
- **Codon**: sequence of three bases that specify which amino acid will be added next to *a protein that is being constructed*.

```
% List of codons. Note the use of importdata()
codon_list = importdata('codonList.txt');
disp(codon_list)
```

```
    {'TTT,Phenylalanine,F'}
    {'TTC,Phenylalanine,F'}
    {'TTA,Leucine,L'      }
    {'TTG,Leucine,L'      }
    {'TCT,Serine,S'       }
    {'TCC,Serine,S'       }
    {'TCA,Serine,S'       }
    {'TCG,Serine,S'       }
    {'TAT,Tyrosine,Y'     }
    {'TAC,Tyrosine,Y'     }
    {'TAA,Stop,X'         }
    {'TAG,Stop,X'         }
    {'TGT,Cysteine,C'     }
    {'TGC,Cysteine,C'     }
    {'TGA,Stop,X'         }
```

```
{'TGG,Tryptophan,W'   }
{'CTT,Leucine,L'      }
{'CTC,Leucine,L'      }
{'CTA,Leucine,L'      }
{'CTG,Leucine,L'      }
{'CCT,Proline,P'      }
{'CCC,Proline,P'      }
{'CCA,Proline,P'      }
{'CCG,Proline,P'      }
{'CAT,Histidine,H'    }
{'CAC,Histidine,H'    }
{'CAA,Glutamine,Q'    }
{'CAG,Glutamine,Q'    }
{'CGT,Arginine,R'     }
{'CGC,Arginine,R'     }
{'CGA,Arginine,R'     }
{'CGG,Arginine,R'     }
{'ATT,Isoleucine,I'   }
{'ATC,Isoleucine,I'   }
{'ATA,Isoleucine,I'   }
{'ATG,Methionine,M'   }
{'ACT,Threonine,T'    }
{'ACC,Threonine,T'    }
{'ACA,Threonine,T'    }
{'ACG,Threonine,T'    }
{'AAT,Asparagine,N'   }
{'AAC,Asparagine,N'   }
{'AAA,Lysine,K'       }
{'AAG,Lysine,K'       }
{'AGT,Serine,S'       }
{'AGC,Serine,S'       }
{'AGA,Arginine,R'     }
{'AGG,Arginine,R'     }
{'GTT,Valine,V'       }
{'GTC,Valine,V'       }
{'GTA,Valine,V'       }
{'GTG,Valine,V'       }
{'GCT,Alanine,A'      }
{'GCC,Alanine,A'      }
{'GCA,Alanine,A'      }
{'GCG,Alanine,A'      }
{'GAT,Aspartate,D'    }
{'GAC,Aspartate,D'    }
{'GAA,Glutamate,E'    }
{'GAG,Glutamate,E'    }
{'GGT,Glycine,G'      }
{'GGC,Glycine,G'      }
{'GGA,Glycine,G'      }
{'GGG,Glycine,G'      }
```

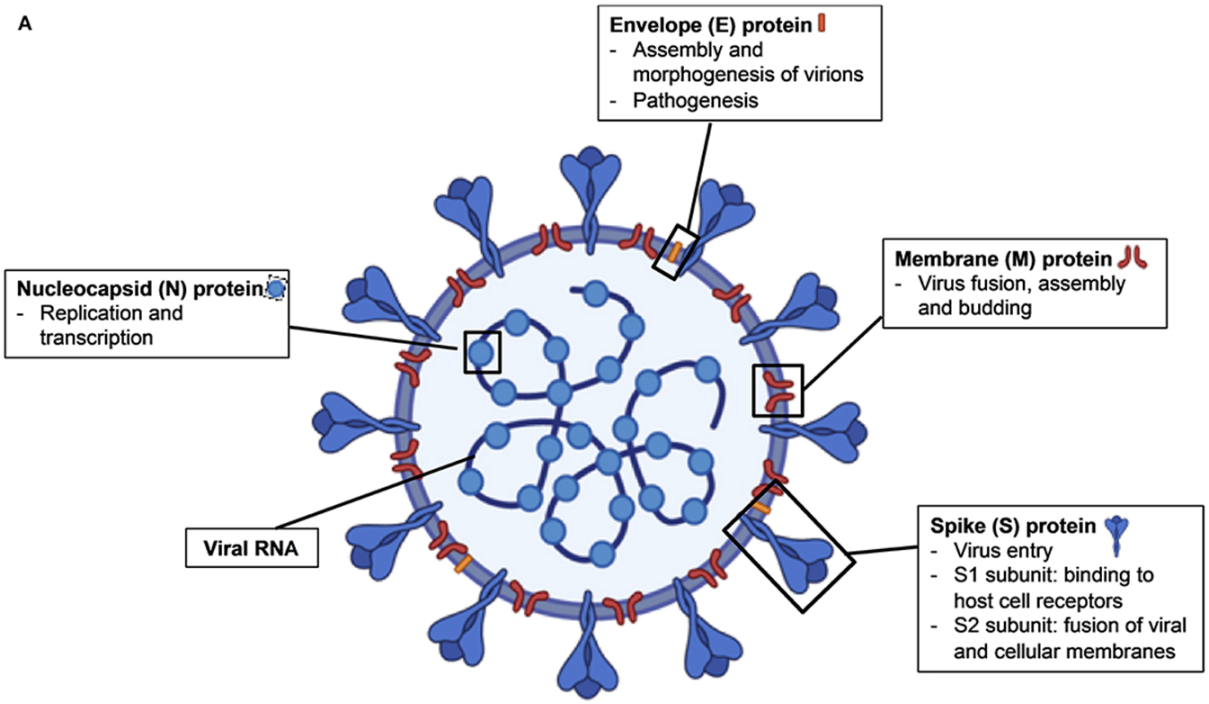
Important notes:

- Genes occur in sets of three bases. One codon (three bases) specifies an amino acid.
- Methionine (DNA base sequence `ATG`) is nearly always the "Start" codon, indicating the start of a sequence for a gene.
- There are three common "Stop" codons that indicate where a gene sequence stops: DNA base sequences `TAA`, `TAG`, `TGA`.
- We are representing these biomolecules as text. Remember that these correspond to *different chemical structures*.

Short summary:

- DNA specifies mRNA, which encodes the protein to be made.
- DNA/mRNA can be represented as text - either ATCG or AUCG
- Protein can be represented as text - a string of one of twenty letters.

In this problem, we will analyze this flow of biological information describing the **"spike"** protein on the membrane of **severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2)**.SARS-CoV-2 is an enveloped, *positive-sense*, single-stranded RNA virus that causes coronavirus disease 2019 (COVID-19). Virus particles include the RNA genetic material and structural proteins needed for invasion of host cells. Once inside the cell the infecting RNA is used to encode structural proteins that make up virus particles, nonstructural proteins that direct virus assembly, transcription, replication and host control and accessory proteins whose function has not been determined.



Source DOI: 10.3389/fimmu.2020.00879

The structural proteins of SARS-CoV-2 include the envelope protein, **spike or surface glycoprotein**, membrane protein and the nucleocapsid protein. **The spike glycoprotein is found on the outside of the virus particle and gives coronavirus viruses their crown-like appearance. This glycoprotein mediates attachment of the virus particle and entry into the host cell.** This protein is an *important target* for vaccine development, antibody therapies and diagnostic antigen-based tests.

Key resources and references:

- Review of SARS-CoV-2 and COVID-19: https://www.nature.com/articles/s41579-020-00459-7
- Genomic information (Wuhan-Hu-1): https://www.ncbi.nlm.nih.gov/nuccore/NC_045512

- General information about coronaviruses: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4369385/

Problem statement

Given a DNA primer (a short sequence of DNA representing the first few bases of a given gene) for the spike glycoprotein and the genome of SARS-CoV-2, answer the following questions:

Question 1: how many bases are in the SARS-CoV-2 genome?

Question 2: how many of *each* base appear in the genome?

Question 3: given a primer, what is the DNA sequence for the spike glycoprotein? How many bases long is it?

Question 4: what is the mRNA transcript of the spike protein?

Question 5: What is the amino acid sequence of the spike protein?

Question 6: How many of each amino acid are in the spike protein?

First thing to do: organize information that is provided

SARS-CoV-2 genome:

```
% Import genomic information as one long character array.
% NOTE: difference from importdata()
sarsCov2_genome = fileread('sarsCov2_genome.txt');  % Read text file
sarsCov2_genome = sarsCov2_genome(isletter(sarsCov2_genome));   % remove spaces
and commas
sarsCov2_genome = upper(sarsCov2_genome);   % Convert to upper case
disp(sarsCov2_genome)
```

```
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTCTCTAAACGAACTTTAAAATC
TGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATAATTAATAACTAATTACTGTCGTTGACAGGACACGAGT
AACTCGTCTATCTTCTGCAGGCTGCTTACGGTTTCGTCCGTGTTGCAGCCGATCATCAGCACATCTAGGTTTCGTCCGGGTGTG
ACCGAAAGGTAAGATGGAGAGCCTTGTCCCTGGTTTCAACGAGAAAACACACGTCCAACTCAGTTTGCCTGTTTTACAGGTTCG
CGACGTGCTCGTACGTGGCTTTGGAGACTCCGTGGAGGAGGTCTTATCAGAGGCACGTCAACATCTTAAAGATGGCACTTGTGG
CTTAGTAGAAGTTGAAAAAGGCGTTTTGCCTCAACTTGAACAGCCCTATGTGTTCATCAAACGTTCGGATGCTCGAACTGCACC
TCATGGTCATGTTATGGTTGAGCTGGTAGCAGAACTCGAAGGCATTCAGTACGGTCGTAGTGGTGAGACACTTGGTGTCCTTGT
CCCTCATGTGGGCGAAATACCAGTGGCTTACCGCAAGGTTCTTCTTCGTAAGAACGGTAATAAAGGAGCTGGTGGCCATAGTTA
CGGCGCCGATCTAAAGTCATTTGACTTAGGCGACGAGCTTGGCACTGATCCTTATGAAGATTTTCAAGAAAACTGGAACACTAA
ACATAGCAGTGGTGTTACCCGTGAACTCATGCGTGAGCTTAACGGAGGGGCATACACTCGCTATGTCGATAACAACTTCTGTGG
CCCTGATGGCTACCCTCTTGAGTGCATTAAAGACCTTCTAGCACGTGCTGGTAAAGCTTCATGCACTTTGTCCGAACAACTGGA
CTTTATTGACACTAAGAGGGGTGTATACTGCTGCCGTGAACATGAGCATGAAATTGCTTGGTACACGGAACGTTCTGAAAAGAG
CTATGAATTGCAGACACCTTTTGAAATTAAATTGGCAAAGAAATTTGACACCTTCAATGGGGAATGTCCAAATTTTGTATTTCC
CTTAAATTCCATAATCAAGACTATTCAACCAAGGGTTGAAAAGAAAAAGCTTGATGGCTTTATGGGTAGAATTCGATCTGTCTA
TCCAGTTGCGTCACCAAATGAATGCAACCAAATGTGCCTTTCAACTCTCATGAAGTGTGATCATTGTGGTGAAACTTCATGGCA
GACGGGCGATTTTGTTAAAGCCACTTGCGAATTTTGTGGCACTGAGAATTTGACTAAAGAAGGTGCCACTACTTGTGGTTACTT
ACCCCAAAATGCTGTTGTTAAAATTTATTGTCCAGCATGTCACAATTCAGAAGTAGGACCTGAGCATAGTCTTGCCGAATACCA
TAATGAATCTGGCTTGAAAACCATTCTTCGTAAGGGTGGTCGCACTATTGCCTTTGGAGGCTGTGTGTTCTCTTATGTTGGTTG
```

CCATAACAAGTGTGCCTATTGGGTTCCACGTGCTAGCGCTAACATAGGTTGTAACCATACAGGTGTTGTTGGAGAAGGTTCCGA
AGGTCTTAATGACAACCTTCTTGAAATACTCCAAAAAGAGAAAGTCAACATCAATATTGTTGGTGACTTTAAACTTAATGAAGA
GATCGCCATTATTTTGGCATCTTTTTCTGCTTCCACAAGTGCTTTTGTGGAAACTGTGAAAGGTTTGGATTATAAAGCATTCAA
ACAAATTGTTGAATCCTGTGGTAATTTTAAAGTTACAAAAGGAAAAGCTAAAAAAGGTGCCTGGAATATTGGTGAACAGAAATC
AATACTGAGTCCTCTTTATGCATTTGCATCAGAGGCTGCTCGTGTTGTACGATCAATTTTCTCCCGCACTCTTGAAACTGCTCA
AAATTCTGTGCGTGTTTTACAGAAGGCCGCTATAACAATACTAGATGGAATTTCACAGTATTCACTGAGACTCATTGATGCTAT
GATGTTCACATCTGATTTGGCTACTAACAATCTAGTTGTAATGGCCTACATTACAGGTGGTGTTGTTCAGTTGACTTCGCAGTG
GCTAACTAACATCTTTGGCACTGTTTATGAAAAACTCAAACCCGTCCTTGATTGGCTTGAAGAGAAGTTTAAGGAAGGTGTAGA
GTTTCTTAGAGACGGTTGGGAAATTGTTAAATTTATCTCAACCTGTGCTTGTGAAATTGTCGGTGGACAAATTGTCACCTGTGC
AAAGGAAATTAAGGAGAGTGTTCAGACATTCTTTAAGCTTGTAAATAAATTTTTGGCTTTGTGTGCTGACTCTATCATTATTGG
TGGAGCTAAACTTAAAGCCTTGAATTTAGGTGAAACATTTGTCACGCACTCAAAGGGATTGTACAGAAAGTGTGTTAAATCCAG
AGAAGAAACTGGCCTACTCATGCCTCTAAAAGCCCCAAAAGAAATTATCTTCTTAGAGGGAGAAACACTTCCCACAGAAGTGTT
AACAGAGGAAGTTGTCTTGAAAACTGGTGATTTACAACCATTAGAACAACCTACTAGTGAAGCTGTTGAAGCTCCATTGGTTGG
TACACCAGTTTGTATTAACGGGCTTATGTTGCTCGAAATCAAAGACACAGAAAAGTACTGTGCCCTTGCACCTAATATGATGGT
AACAAACAATACCTTCACACTCAAAGGCGGTGCACCAACAAAGGTTACTTTTGGTGATGACACTGTGATAGAAGTGCAAGGTTA
CAAGAGTGTGAATATCACTTTTGAACTTGATGAAAGGATTGATAAAGTACTTAATGAGAAGTGCTCTGCCTATACAGTTGAACT
CGGTACAGAAGTAAATGAGTTCGCCTGTGTTGTGGCAGATGCTGTCATAAAAACTTTGCAACCAGTATCTGAATTACTTACACC
ACTGGGCATTGATTTAGATGAGTGGAGTATGGCTACATACTACTTATTTGATGAGTCTGGTGAGTTTAAATTGGCTTCACATAT
GTATTGTTCTTTCTACCCTCCAGATGAGGATGAAGAAGAAGGTGATTGTGAAGAAGAAGAGTTTGAGCCATCAACTCAATATGA
GTATGGTACTGAAGATGATTACCAAGGTAAACCTTTGGAATTTGGTGCCACTTCTGCTGCTCTTCAACCTGAAGAAGAGCAAGA
AGAAGATTGGTTAGATGATGATAGTCAACAAACTGTTGGTCAACAAGACGGCAGTGAGGACAATCAGACAACTACTATTCAAAC
AATTGTTGAGGTTCAACCTCAATTAGAGATGGAACTTACACCAGTTGTTCAGACTATTGAAGTGAATAGTTTTAGTGGTTATTT
AAAACTTACTGACAATGTATACATTAAAAATGCAGACATTGTGGAAGAAGCTAAAAAGGTAAAACCAACAGTGGTTGTTAATGC
AGCCAATGTTTACCTTAAACATGGAGGAGGTGTTGCAGGAGCCTTAAATAAGGCTACTAACAATGCCATGCAAGTTGAATCTGA
TGATTACATAGCTACTAATGGACCACTTAAAGTGGGTGGTAGTTGTGTTTTAAGCGGACACAATCTTGCTAAACACTGTCTTCA
TGTTGTCGGCCCAAATGTTAACAAAGGTGAAGACATTCAACTTCTTAAGAGTGCTTATGAAAATTTTAATCAGCACGAAGTTCT
ACTTGCACCATTATTATCAGCTGGTATTTTTGGTGCTGACCCTATACATTCTTTAAGAGTTTGTGTAGATACTGTTCGCACAAA
TGTCTACTTAGCTGTCTTTGATAAAAATCTCTATGACAAACTTGTTTCAAGCTTTTTGGAAATGAAGAGTGAAAAGCAAGTTGA
ACAAAAGATCGCTGAGATTCCTAAAGAGGAAGTTAAGCCATTTATAACTGAAAGTAAACCTTCAGTTGAACAGAGAAAACAAGA
TGATAAGAAAATCAAAGCTTGTGTTGAAGAAGTTACAACAACTCTGGAAGAAACTAAGTTCCTCACAGAAAACTTGTTACTTTA
TATTGACATTAATGGCAATCTTCATCCAGATTCTGCCACTCTTGTTAGTGACATTGACATCACTTTCTTAAAGAAAGATGCTCC
ATATATAGTGGGTGATGTTGTTCAAGAGGGTGTTTTAACTGCTGTGGTTATACCTACTAAAAAGGCTGGTGGCACTACTGAAAT
GCTAGCGAAAGCTTTGAGAAAAGTGCCAACAGACAATTATATAACCACTTACCCGGGTCAGGGTTTAAATGGTTACACTGTAGA
GGAGGCAAAGACAGTGCTTAAAAAGTGTAAAAGTGCCTTTTACATTCTACCATCTATTATCTCTAATGAGAAGCAAGAAATTCT
TGGAACTGTTTCTTGGAATTTGCGAGAAATGCTTGCACATGCAGAAGAAACACGCAAATTAATGCCTGTCTGTGTGGAAACTAA
AGCCATAGTTTCAACTATACAGCGTAAATATAAGGGTATTAAAATACAAGAGGGTGTGGTTGATTATGGTGCTAGATTTTACTT
TTACACCAGTAAAACAACTGTAGCGTCACTTATCAACACACTTAACGATCTAAATGAAACTCTTGTTACAATGCCACTTGGCTA
TGTAACACATGGCTTAAATTTGGAAGAAGCTGCTCGGTATATGAGATCTCTCAAAGTGCCAGCTACAGTTTCTGTTTCTTCACC
TGATGCTGTTACAGCGTATAATGGTTATCTTACTTCTTCTTCTAAAACACCTGAAGAACATTTTATTGAAACCATCTCACTTGC
TGGTTCCTATAAAGATTGGTCCTATTCTGGACAATCTACACAACTAGGTATAGAATTTCTTAAGAGAGGTGATAAAAGTGTATA
TTACACTAGTAATCCTACCACATTCCACCTAGATGGTGAAGTTATCACCTTTGACAATCTTAAGACACTTCTTTCTTTGAGAGA
AGTGAGGACTATTAAGGTGTTTACAACAGTAGACAACATTAACCTCCACACGCAAGTTGTGGACATGTCAATGACATATGGACA
ACAGTTTGGTCCAACTTATTTGGATGGAGCTGATGTTACTAAAATAAAACCTCATAATTCACATGAAGGTAAAACATTTTATGT
TTTACCTAATGATGACACTCTACGTGTTGAGGCTTTTGAGTACTACCACACAACTGATCCTAGTTTTCTGGGTAGGTACATGTC
AGCATTAAATCACACTAAAAAGTGGAAATACCCACAAGTTAATGGTTTAACTTCTATTAAATGGGCAGATAACAACTGTTATCT
TGCCACTGCATTGTTAACACTCCAACAAATAGAGTTGAAGTTTAATCCACCTGCTCTACAAGATGCTTATTACAGAGCAAGGGC
TGGTGAAGCTGCTAACTTTTGTGCACTTATCTTAGCCTACTGTAATAAGACAGTAGGTGAGTTAGGTGATGTTAGAGAAACAAT
GAGTTACTTGTTTCAACATGCCAATTTAGATTCTTGCAAAAGAGTCTTGAACGTGGTGTGTAAAACTTGTGGACAACAGCAGAC
AACCCTTAAGGGTGTAGAAGCTGTTATGTACATGGGCACACTTTCTTATGAACAATTTAAGAAAGGTGTTCAGATACCTTGTAC
GTGTGGTAAACAAGCTACAAAATATCTAGTACAACAGGAGTCACCTTTTGTTATGATGTCAGCACCACCTGCTCAGTATGAACT
TAAGCATGGTACATTTACTTGTGCTAGTGAGTACACTGGTAATTACCAGTGTGGTCACTATAAACATATAACTTCTAAAGAAAC
TTTGTATTGCATAGACGGTGCTTTACTTACAAAGTCCTCAGAATACAAAGGTCCTATTACGGATGTTTTCTACAAAGAAAACAG
TTACACAACAACCATAAAACCAGTTACTTATAAATTGGATGGTGTTGTTTGTACAGAAATTGACCCTAAGTTGGACAATTATTA
TAAGAAAGACAATTCTTATTTCACAGAGCAACCAATTGATCTTGTACCAAACCAACCATATCCAAACGCAAGCTTCGATAATTT
TAAGTTTGTATGTGATAATATCAAATTTGCTGATGATTTAAACCAGTTAACTGGTTATAAGAAACCTGCTTCAAGAGAGCTTAA
AGTTACATTTTTCCCTGACTTAAATGGTGATGTGGTGGCTATTGATTATAAACACTACACACCCTCTTTTAAGAAAGGAGCTAA
ATTGTTACATAAACCTATTGTTTGGCATGTTAACAATGCAACTAATAAAGCCACGTATAAACCAAATACCTGGTGTATACGTTG
TCTTTGGAGCACAAAACCAGTTGAAACATCAAATTCGTTTGATGTACTGAAGTCAGAGGACGCGCAGGGAATGGATAATCTTGC
CTGCGAAGATCTAAAACCAGTCTCTGAAGAAGTAGTGGAAAATCCTACCATACAGAAAGACGTTCTTGAGTGTAATGTGAAAAC
TACCGAAGTTGTAGGAGACATTATACTTAAACCAGCAAATAATAGTTTAAAAATTACAGAAGAGGTTGGCCACACAGATCTAAT
GGCTGCTTATGTAGACAATTCTAGTCTTACTATTAAGAAACCTAATGAATTATCTAGAGTATTAGGTTTGAAAACCCTTGCTAC

TCATGGTTTAGCTGCTGTTAATAGTGTCCCTTGGGATACTATAGCTAATTATGCTAAGCCTTTTCTTAACAAAGTTGTTAGTAC
AACTACTAACATAGTTACACGGTGTTTAAACCGTGTTTGTACTAATTATATGCCTTATTTCTTTACTTTATTGCTACAATTGTG
TACTTTTACTAGAAGTACAAATTCTAGAATTAAAGCATCTATGCCGACTACTATAGCAAAGAATACTGTTAAGAGTGTCGGTAA
ATTTTGTCTAGAGGCTTCATTTAATTATTTGAAGTCACCTAATTTTTCTAAACTGATAAATATTATAATTTGGTTTTTACTATT
AAGTGTTTGCCTAGGTTCTTTAATCTACTCAACCGCTGCTTTAGGTGTTTTAATGTCTAATTTAGGCATGCCTTCTTACTGTAC
TGGTTACAGAGAAGGCTATTTGAACTCTACTAATGTCACTATTGCAACCTACTGTACTGGTTCTATACCTTGTAGTGTTTGTCT
TAGTGGTTTAGATTCTTTAGACACCTATCCTTCTTTAGAAACTATACAAATTACCATTTCATCTTTTAAATGGGATTTAACTGC
TTTTGGCTTAGTTGCAGAGTGGTTTTTGGCATATATTCTTTTCACTAGGTTTTTCTATGTACTTGGATTGGCTGCAATCATGCA
ATTGTTTTTCAGCTATTTTGCAGTACATTTTATTAGTAATTCTTGGCTTATGTGGTTAATAATTAATCTTGTACAAATGGCCCC
GATTTCAGCTATGGTTAGAATGTACATCTTCTTTGCATCATTTTATTATGTATGGAAAAGTTATGTGCATGTTGTAGACGGTTG
TAATTCATCAACTTGTATGATGTGTTACAAACGTAATAGAGCAACAAGAGTCGAATGTACAACTATTGTTAATGGTGTTAGAAG
GTCCTTTTATGTCTATGCTAATGGAGGTAAAGGCTTTTGCAAACTACACAATTGGAATTGTGTTAATTGTGATACATTCTGTGC
TGGTAGTACATTTATTAGTGATGAAGTTGCGAGAGACTTGTCACTACAGTTTAAAAGACCAATAAATCCTACTGACCAGTCTTC
TTACATCGTTGATAGTGTTACAGTGAAGAATGGTTCCATCCATCTTTACTTTGATAAAGCTGGTCAAAAGACTTATGAAAGACA
TTCTCTCTCTCATTTTGTTAACTTAGACAACCTGAGAGCTAATAACACTAAAGGTTCATTGCCTATTAATGTTATAGTTTTTGA
TGGTAAATCAAAATGTGAAGAATCATCTGCAAAATCAGCGTCTGTTTACTACAGTCAGCTTATGTGTCAACCTATACTGTTACT
AGATCAGGCATTAGTGTCTGATGTTGGTGATAGTGCGGAAGTTGCAGTTAAAATGTTTGATGCTTACGTTAATACGTTTTCATC
AACTTTTAACGTACCAATGGAAAAACTCAAAACACTAGTTGCAACTGCAGAAGCTGAACTTGCAAAGAATGTGTCCTTAGACAA
TGTCTTATCTACTTTTATTTCAGCAGCTCGGCAAGGGTTTGTTGATTCAGATGTAGAAACTAAAGATGTTGTTGAATGTCTTAA
ATTGTCACATCAATCTGACATAGAAGTTACTGGCGATAGTTGTAATAACTATATGCTCACCTATAACAAAGTTGAAAACATGAC
ACCCCGTGACCTTGGTGCTTGTATTGACTGTAGTGCGCGTCATATTAATGCGCAGGTAGCAAAAAGTCACAACATTGCTTTGAT
ATGGAACGTTAAAGATTTCATGTCATTGTCTGAACAACTACGAAAACAAATACGTAGTGCTGCTAAAAAGAATAACTTACCTTT
TAAGTTGACATGTGCAACTACTAGACAAGTTGTTAATGTTGTAACAACAAAGATAGCACTTAAGGGTGGTAAAATTGTTAATAA
TTGGTTGAAGCAGTTAATTAAAGTTACACTTGTGTTCCTTTTTGTTGCTGCTATTTTCTATTTAATAACACCTGTTCATGTCAT
GTCTAAACATACTGACTTTTCAAGTGAAATCATAGGATACAAGGCTATTGATGGTGGTGTCACTCGTGACATAGCATCTACAGA
TACTTGTTTTGCTAACAAACATGCTGATTTTGACACATGGTTTAGCCAGCGTGGTGGTAGTTATACTAATGACAAAGCTTGCCC
ATTGATTGCTGCAGTCATAACAAGAGAAGTGGGTTTTGTCGTGCCTGGTTTGCCTGGCACGATATTACGCACAACTAATGGTGA
CTTTTTGCATTTCTTACCTAGAGTTTTTAGTGCAGTTGGTAACATCTGTTACACACCATCAAAACTTATAGAGTACACTGACTT
TGCAACATCAGCTTGTGTTTTGGCTGCTGAATGTACAATTTTTAAAGATGCTTCTGGTAAGCCAGTACCATATTGTTATGATAC
CAATGTACTAGAAGGTTCTGTTGCTTATGAAAGTTTACGCCCTGACACACGTTATGTGCTCATGGATGGCTCTATTATTCAATT
TCCTAACACCTACCTTGAAGGTTCTGTTAGAGTGGTAACAACTTTTGATTCTGAGTACTGTAGGCACGGCACTTGTGAAAGATC
AGAAGCTGGTGTTTGTGTATCTACTAGTGGTAGATGGGTACTTAACAATGATTATTACAGATCTTTACCAGGAGTTTTCTGTGG
TGTAGATGCTGTAAATTTACTTACTAATATGTTTACACCACTAATTCAACCTATTGGTGCTTTGGACATATCAGCATCTATAGT
AGCTGGTGGTATTGTAGCTATCGTAGTAACATGCCTTGCCTACTATTTTATGAGGTTTAGAAGAGCTTTTGGTGAATACAGTCA
TGTAGTTGCCTTTAATACTTTACTATTCCTTATGTCATTCACTGTACTCTGTTTAACACCAGTTTACTCATTCTTACCTGGTGT
TTATTCTGTTATTTACTTGTACTTGACATTTTATCTTACTAATGATGTTTCTTTTTTAGCACATATTCAGTGGATGGTTATGTT
CACACCTTTAGTACCTTTCTGGATAACAATTGCTTATATCATTTGTATTTCCACAAAGCATTTCTATTGGTTCTTTAGTAATTA
CCTAAAGAGACGTGTAGTCTTTAATGGTGTTTCCTTTAGTACTTTTGAAGAAGCTGCGCTGTGCACCTTTTTGTTAAATAAAGA
AATGTATCTAAAGTTGCGTAGTGATGTGCTATTACCTCTTACGCAATATAATAGATACTTAGCTCTTTATAATAAGTACAAGTA
TTTTAGTGGAGCAATGGATACAACTAGCTACAGAGAAGCTGCTTGTTGTCATCTCGCAAAGGCTCTCAATGACTTCAGTAACTC
AGGTTCTGATGTTCTTTACCAACCACCACAAACCTCTATCACCTCAGCTGTTTTGCAGAGTGGTTTTAGAAAAATGGCATTCCC
ATCTGGTAAAGTTGAGGGTTGTATGGTACAAGTAACTTGTGGTACAACTACACTTAACGGTCTTTGGCTTGATGACGTAGTTTA
CTGTCCAAGACATGTGATCTGCACCTCTGAAGACATGCTTAACCCTAATTATGAAGATTTACTCATTCGTAAGTCTAATCATAA
TTTCTTGGTACAGGCTGGTAATGTTCAACTCAGGGTTATTGGACATTCTATGCAAAATTGTGTACTTAAGCTTAAGGTTGATAC
AGCCAATCCTAAGACACCTAAGTATAAGTTTGTTCGCATTCAACCAGGACAGACTTTTTCAGTGTTAGCTTGTTACAATGGTTC
ACCATCTGGTGTTTACCAATGTGCTATGAGGCCCAATTTCACTATTAAGGGTTCATTCCTTAATGGTTCATGTGGTAGTGTTGG
TTTTAACATAGATTATGACTGTGTCTCTTTTTGTTACATGCACCATATGGAATTACCAACTGGAGTTCATGCTGGCACAGACTT
AGAAGGTAACTTTTATGGACCTTTTGTTGACAGGCAAACAGCACAAGCAGCTGGTACGGACACAACTATTACAGTTAATGTTTT
AGCTTGGTTGTACGCTGCTGTTATAAATGGAGACAGGTGGTTTCTCAATCGATTTACCACAACTCTTAATGACTTTAACCTTGT
GGCTATGAAGTACAATTATGAACCTCTAACACAAGACCATGTTGACATACTAGGACCTCTTTCTGCTCAAACTGGAATTGCCGT
TTTAGATATGTGTGCTTCATTAAAAGAATTACTGCAAAATGGTATGAATGGACGTACCATATTGGGTAGTGCTTTATTAGAAGA
TGAATTTACACCTTTTGATGTTGTTAGACAATGCTCAGGTGTTACTTTCCAAAGTGCAGTGAAAAGAACAATCAAGGGTACACA
CCACTGGTTGTTACTCACAATTTTGACTTCACTTTTAGTTTTAGTCCAGAGTACTCAATGGTCTTTGTTCTTTTTTTTGTATGA
AAATGCCTTTTTACCTTTTGCTATGGGTATTATTGCTATGTCTGCTTTTGCAATGATGTTTGTCAAACATAAGCATGCATTTCT
CTGTTTGTTTTTGTTACCTTCTCTTGCCACTGTAGCTTATTTTAATATGGTCTATATGCCTGCTAGTTGGGTGATGCGTATTAT
GACATGGTTGGATATGGTTGATACTAGTTTGTCTGGTTTTAAGCTAAAAGACTGTGTTATGTATGCATCAGCTGTAGTGTTACT
AATCCTTATGACAGCAAGAACTGTGTATGATGATGGTGCTAGGAGAGTGTGGACACTTATGAATGTCTTGACACTCGTTTATAA
AGTTTATTATGGTAATGCTTTAGATCAAGCCATTTCCATGTGGGCTCTTATAATCTCTGTTACTTCTAACTACTCAGGTGTAGT
TACAACTGTCATGTTTTTGGCCAGAGGTATTGTTTTTATGTGTGTTGAGTATTGCCCTATTTTCTTCATAACTGGTAATACACT
TCAGTGTATAATGCTAGTTTATTGTTTCTTAGGCTATTTTTGTACTTGTTACTTTGGCCTCTTTTGTTTACTCAACCGCTACTT
TAGACTGACTCTTGGTGTTTATGATTACTTAGTTTCTACACAGGAGTTTAGATATATGAATTCACAGGGACTACTCCCACCCAA

GAATAGCATAGATGCCTTCAAACTCAACATTAAATTGTTGGGTGTTGGTGGCAAACCTTGTATCAAAGTAGCCACTGTACAGTC
TAAAATGTCAGATGTAAAGTGCACATCAGTAGTCTTACTCTCAGTTTTGCAACAACTCAGAGTAGAATCATCATCTAAATTGTG
GGCTCAATGTGTCCAGTTACACAATGACATTCTCTTAGCTAAAGATACTACTGAAGCCTTTGAAAAAATGGTTTCACTACTTTC
TGTTTTGCTTTCCATGCAGGGTGCTGTAGACATAAACAAGCTTTGTGAAGAAATGCTGGACAACAGGGCAACCTTACAAGCTAT
AGCCTCAGAGTTTAGTTCCCTTCCATCATATGCAGCTTTTGCTACTGCTCAAGAAGCTTATGAGCAGGCTGTTGCTAATGGTGA
TTCTGAAGTTGTTCTTAAAAAGTTGAAGAAGTCTTTGAATGTGGCTAAATCTGAATTTGACCGTGATGCAGCCATGCAACGTAA
GTTGGAAAAGATGGCTGATCAAGCTATGACCCAAATGTATAAACAGGCTAGATCTGAGGACAAGAGGGCAAAAGTTACTAGTGC
TATGCAGACAATGCTTTTCACTATGCTTAGAAAGTTGGATAATGATGCACTCAACAACATTATCAACAATGCAAGAGATGGTTG
TGTTCCCTTGAACATAATACCTCTTACAACAGCAGCCAAACTAATGGTTGTCATACCAGACTATAACACATATAAAAATACGTG
TGATGGTACAACATTTACTTATGCATCAGCATTGTGGGAAATCCAACAGGTTGTAGATGCAGATAGTAAAATTGTTCAACTTAG
TGAAATTAGTATGGACAATTCACCTAATTTAGCATGGCCTCTTATTGTAACAGCTTTAAGGGCCAATTCTGCTGTCAAATTACA
GAATAATGAGCTTAGTCCTGTTGCACTACGACAGATGTCTTGTGCTGCCGGTACTACACAAACTGCTTGCACTGATGACAATGC
GTTAGCTTACTACAACACAACAAAGGGAGGTAGGTTTGTACTTGCACTGTTATCCGATTTACAGGATTTGAAATGGGCTAGATT
CCCTAAGAGTGATGGAACTGGTACTATCTATACAGAACTGGAACCACCTTGTAGGTTTGTTACAGACACACCTAAAGGTCCTAA
AGTGAAGTATTTATACTTTATTAAAGGATTAAACAACCTAAATAGAGGTATGGTACTTGGTAGTTTAGCTGCCACAGTACGTCT
ACAAGCTGGTAATGCAACAGAAGTGCCTGCCAATTCAACTGTATTATCTTTCTGTGCTTTTGCTGTAGATGCTGCTAAAGCTTA
CAAAGATTATCTAGCTAGTGGGGGACAACCAATCACTAATTGTGTTAAGATGTTGTGTACACACACTGGTACTGGTCAGGCAAT
AACAGTTACACCGGAAGCCAATATGGATCAAGAATCCTTTGGTGGTGCATCGTGTTGTCTGTACTGCCGTTGCCACATAGATCA
TCCAAATCCTAAAGGATTTTGTGACTTAAAAGGTAAGTATGTACAAATACCTACAACTTGTGCTAATGACCCTGTGGGTTTTAC
ACTTAAAAACACAGTCTGTACCGTCTGCGGTATGTGGAAAGGTTATGGCTGTAGTTGTGATCAACTCCGCGAACCCATGCTTCA
GTCAGCTGATGCACAATCGTTTTTAAACGGGTTTGCGGTGTAAGTGCAGCCCGTCTTACACCGTGCGGCACAGGCACTAGTACT
GATGTCGTATACAGGGCTTTTGACATCTACAATGATAAAGTAGCTGGTTTTGCTAAATTCCTAAAAACTAATTGTTGTCGCTTC
CAAGAAAAGGACGAAGATGACAATTTAATTGATTCTTACTTTGTAGTTAAGAGACACACTTTCTCTAACTACCAACATGAAGAA
ACAATTTATAATTTACTTAAGGATTGTCCAGCTGTTGCTAAACATGACTTCTTTAAGTTTAGAATAGACGGTGACATGGTACCA
CATATATCACGTCAACGTCTTACTAAATACACAATGGCAGACCTCGTCTATGCTTTAAGGCATTTTGATGAAGGTAATTGTGAC
ACATTAAAAGAAATACTTGTCACATACAATTGTTGTGATGATGATTATTTCAATAAAAAGGACTGGTATGATTTTGTAGAAAAC
CCAGATATATTACGCGTATACGCCAACTTAGGTGAACGTGTACGCCAAGCTTTGTTAAAAACAGTACAATTCTGTGATGCCATG
CGAAATGCTGGTATTGTTGGTGTACTGACATTAGATAATCAAGATCTCAATGGTAACTGGTATGATTTCGGTGATTTCATACAA
ACCACGCCAGGTAGTGGAGTTCCTGTTGTAGATTCTTATTATTCATTGTTAATGCCTATATTAACCTTGACCAGGGCTTTAACT
GCAGAGTCACATGTTGACACTGACTTAACAAAGCCTTACATTAAGTGGGATTTGTTAAAATATGACTTCACGGAAGAGAGGTTA
AAACTCTTTGACCGTTATTTTAAATATTGGGATCAGACATACCACCCAAATTGTGTTAACTGTTTGGATGACAGATGCATTCTG
CATTGTGCAAACTTTAATGTTTTATTCTCTACAGTGTTCCCACCTACAAGTTTTGGACCACTAGTGAGAAAAATATTTGTTGAT
GGTGTTCCATTTGTAGTTTCAACTGGATACCACTTCAGAGAGCTAGGTGTTGTACATAATCAGGATGTAAACTTACATAGCTCT
AGACTTAGTTTTAAGGAATTACTTGTGTATGCTGCTGACCCTGCTATGCACGCTGCTTCTGGTAATCTATTACTAGATAAACGC
ACTACGTGCTTTTCAGTAGCTGCACTTACTAACAATGTTGCTTTTCAAACTGTCAAACCCGGTAATTTTAACAAAGACTTCTAT
GACTTTGCTGTGTCTAAGGGTTTCTTTAAGGAAGGAAGTTCTGTTGAATTAAAACACTTCTTCTTTGCTCAGGATGGTAATGCT
GCTATCAGCGATTATGACTACTATCGTTATAATCTACCAACAATGTGTGATATCAGACAACTACTATTTGTAGTTGAAGTTGTT
GATAAGTACTTTGATTGTTACGATGGTGGCTGTATTAATGCTAACCAAGTCATCGTCAACAACCTAGACAAATCAGCTGGTTTT
CCATTTAATAAATGGGGTAAGGCTAGACTTTATTATGATTCAATGAGTTATGAGGATCAAGATGCACTTTTCGCATATACAAAA
CGTAATGTCATCCCTACTATAACTCAAATGAATCTTAAGTATGCCATTAGTGCAAAGAATAGAGCTCGCACCGTAGCTGGTGTC
TCTATCTGTAGTACTATGACCAATAGACAGTTTCATCAAAAATTATTGAAATCAATAGCCGCCACTAGAGGAGCTACTGTAGTA
ATTGGAACAAGCAAATTCTATGGTGGTTGGCACAACATGTTAAAAACTGTTTATAGTGATGTAGAAAACCCTCACCTTATGGGT
TGGGATTATCCTAAATGTGATAGAGCCATGCCTAACATGCTTAGAATTATGGCCTCACTTGTTCTTGCTCGCAAACATACAACG
TGTTGTAGCTTGTCACACCGTTTCTATAGATTAGCTAATGAGTGTGCTCAAGTATTGAGTGAAATGGTCATGTGTGGCGGTTCA
CTATATGTTAAACCAGGTGGAACCTCATCAGGAGATGCCACAACTGCTTATGCTAATAGTGTTTTTAACATTTGTCAAGCTGTC
ACGGCCAATGTTAATGCACTTTTATCTACTGATGGTAACAAAATTGCCGATAAGTATGTCCGCAATTTACAACACAGACTTTAT
GAGTGTCTCTATAGAAATAGAGATGTTGACACAGACTTTGTGAATGAGTTTTACGCATATTTGCGTAAACATTTCTCAATGATG
ATACTCTCTGACGATGCTGTTGTGTGTTTCAATAGCACTTATGCATCTCAAGGTCTAGTGGCTAGCATAAAGAACTTTAAGTCA
GTTCTTTATTATCAAAACAATGTTTTTATGTCTGAAGCAAAATGTTGGACTGAGACTGACCTTACTAAAGGACCTCATGAATTT
TGCTCTCAACATACAATGCTAGTTAAACAGGGTGATGATTATGTGTACCTTCCTTACCCAGATCCATCAAGAATCCTAGGGGCC
GGCTGTTTTGTAGATGATATCGTAAAAACAGATGGTACACTTATGATTGAACGGTTCGTGTCTTTAGCTATAGATGCTTACCCA
CTTACTAAACATCCTAATCAGGAGTATGCTGATGTCTTTCATTTGTACTTACAATACATAAGAAAGCTACATGATGAGTTAACA
GGACACATGTTAGACATGTATTCTGTTATGCTTACTAATGATAACACTTCAAGGTATTGGGAACCTGAGTTTTATGAGGCTATG
TACACACCGCATACAGTCTTACAGGCTGTTGGGGCTTGTGTTCTTTGCAATTCACAGACTTCATTAAGATGTGGTGCTTGCATA
CGTAGACCATTCTTATGTTGTAAATGCTGTTACGACCATGTCATATCAACATCACATAAATTAGTCTTGTCTGTTAATCCGTAT
GTTTGCAATGCTCCAGGTTGTGATGTCACAGATGTGACTCAACTTTACTTAGGAGGTATGAGCTATTATTGTAAATCACATAAA
CCACCCATTAGTTTTCCATTGTGTGCTAATGGACAAGTTTTTGGTTTATATAAAAATACATGTGTTGGTAGCGATAATGTTACT
GACTTTAATGCAATTGCAACATGTGACTGGACAAATGCTGGTGATTACATTTTAGCTAACACCTGTACTGAAAGACTCAAGCTT
TTTGCAGCAGAAACGCTCAAAGCTACTGAGGAGACATTTAAACTGTCTTATGGTATTGCTACTGTACGTGAAGTGCTGTCTGAC
AGAGAATTACATCTTTCATGGGAAGTTGGTAAACCTAGACCACCACTTAACCGAAATTATGTCTTTACTGGTTATCGTGTAACT
AAAAACAGTAAAGTACAAATAGGAGAGTACACCTTTGAAAAAGGTGACTATGGTGATGCTGTTGTTTACCGAGGTACAACAACT

TACAAATTAAATGTTGGTGATTATTTTGTGCTGACATCACATACAGTAATGCCATTAAGTGCACCTACACTAGTGCCACAAGAG
CACTATGTTAGAATTACTGGCTTATACCCAACACTCAATATCTCAGATGAGTTTTCTAGCAATGTTGCAAATTATCAAAAGGTT
GGTATGCAAAAGTATTCTACACTCCAGGGACCACCTGGTACTGGTAAGAGTCATTTTGCTATTGGCCTAGCTCTCTACTACCCT
TCTGCTCGCATAGTGTATACAGCTTGCTCTCATGCCGCTGTTGATGCACTATGTGAGAAGGCATTAAAATATTTGCCTATAGAT
AAATGTAGTAGAATTATACCTGCACGTGCTCGTGTAGAGTGTTTTGATAAATTCAAAGTGAATTCAACATTAGAACAGTATGTC
TTTTGTACTGTAAATGCATTGCCTGAGACGACAGCAGATATAGTTGTCTTTGATGAAATTTCAATGGCCACAAATTATGATTTG
AGTGTTGTCAATGCCAGATTACGTGCTAAGCACTATGTGTACATTGGCGACCCTGCTCAATTACCTGCACCACGCACATTGCTA
ACTAAGGGCACACTAGAACCAGAATATTTCAATTCAGTGTGTAGACTTATGAAAACTATAGGTCCAGACATGTTCCTCGGAACT
TGTCGGCGTTGTCCTGCTGAAATTGTTGACACTGTGAGTGCTTTGGTTTATGATAATAAGCTTAAAGCACATAAAGACAAATCA
GCTCAATGCTTTAAAATGTTTTATAAGGGTGTTATCACGCATGATGTTTCATCTGCAATTAACAGGCCACAAATAGGCGTGGTA
AGAGAATTCCTTACACGTAACCCTGCTTGGAGAAAAGCTGTCTTTATTTCACCTTATAATTCACAGAATGCTGTAGCCTCAAAG
ATTTTGGGACTACCAACTCAAACTGTTGATTCATCACAGGGCTCAGAATATGACTATGTCATATTCACTCAAACCACTGAAACA
GCTCACTCTTGTAATGTAAACAGATTTAATGTTGCTATTACCAGAGCAAAAGTAGGCATACTTTGCATAATGTCTGATAGAGAC
CTTTATGACAAGTTGCAATTTACAAGTCTTGAAATTCCACGTAGGAATGTGGCAACTTTACAAGCTGAAAATGTAACAGGACTC
TTTAAAGATTGTAGTAAGGTAATCACTGGGTTACATCCTACACAGGCACCTACACACCTCAGTGTTGACACTAAATTCAAAACT
GAAGGTTTATGTGTTGACATACCTGGCATACCTAAGGACATGACCTATAGAAGACTCATCTCTATGATGGGTTTTAAAATGAAT
TATCAAGTTAATGGTTACCCTAACATGTTTATCACCCGCGAAGAAGCTATAAGACATGTACGTGCATGGATTGGCTTCGATGTC
GAGGGGTGTCATGCTACTAGAGAAGCTGTTGGTACCAATTTACCTTTACAGCTAGGTTTTTCTACAGGTGTTAACCTAGTTGCT
GTACCTACAGGTTATGTTGATACACCTAATAATACAGATTTTTCCAGAGTTAGTGCTAAACCACCGCCTGGAGATCAATTTAAA
CACCTCATACCACTTATGTACAAAGGACTTCCTTGGAATGTAGTGCGTATAAAGATTGTACAAATGTTAAGTGACACACTTAAA
AATCTCTCTGACAGAGTCGTATTTGTCTTATGGGCACATGGCTTTGAGTTGACATCTATGAAGTATTTTGTGAAAATAGGACCT
GAGCGCACCTGTTGTCTATGTGATAGACGTGCCACATGCTTTTCCACTGCTTCAGACACTTATGCCTGTTGGCATCATTCTATT
GGATTTGATTACGTCTATAATCCGTTTATGATTGATGTTCAACAATGGGGTTTTACAGGTAACCTACAAAGCAACCATGATCTG
TATTGTCAAGTCCATGGTAATGCACATGTAGCTAGTTGTGATGCAATCATGACTAGGTGTCTAGCTGTCCACGAGTGCTTTGTT
AAGCGTGTTGACTGGACTATTGAATATCCTATAATTGGTGATGAACTGAAGATTAATGCGGCTTGTAGAAAGGTTCAACACATG
GTTGTTAAAGCTGCATTATTAGCAGACAAATTCCCAGTTCTTCACGACATTGGTAACCCTAAAGCTATTAAGTGTGTACCTCAA
GCTGATGTAGAATGGAAGTTCTATGATGCACAGCCTTGTAGTGACAAAGCTTATAAAATAGAAGAATTATTCTATTCTTATGCC
ACACATTCTGACAAATTCACAGATGGTGTATGCCTATTTTGGAATTGCAATGTCGATAGATATCCTGCTAATTCCATTGTTTGT
AGATTTGACACTAGAGTGCTATCTAACCTTAACTTGCCTGGTTGTGATGGTGGCAGTTTGTATGTAAATAAACATGCATTCCAC
ACACCAGCTTTTGATAAAAGTGCTTTTGTTAATTTAAAACAATTACCATTTTTCTATTACTCTGACAGTCCATGTGAGTCTCAT
GGAAAACAAGTAGTGTCAGATATAGATTATGTACCACTAAAGTCTGCTACGTGTATAACACGTTGCAATTTAGGTGGTGCTGTC
TGTAGACATCATGCTAATGAGTACAGATTGTATCTCGATGCTTATAACATGATGATCTCAGCTGGCTTTAGCTTGTGGGTTTAC
AAACAATTTGATACTTATAACCTCTGGAACACTTTTACAAGACTTCAGAGTTTAGAAAATGTGGCTTTTAATGTTGTAAATAAG
GGACACTTTGATGGACAACAGGGTGAAGTACCAGTTTCTATCATTAATAACACTGTTTACACAAAAGTTGATGGTGTTGATGTA
GAATTGTTTGAAAATAAAACAACATTACCTGTTAATGTAGCATTTGAGCTTTGGGCTAAGCGCAACATTAAACCAGTACCAGAG
GTGAAAATACTCAATAATTTGGGTGTGGACATTGCTGCTAATACTGTGATCTGGGACTACAAAAGAGATGCTCCAGCACATATA
TCTACTATTGGTGTTTGTTCTATGACTGACATAGCCAAGAAACCAACTGAAACGATTTGTGCACCACTCACTGTCTTTTTTGAT
GGTAGAGTTGATGGTCAAGTAGACTTATTTAGAAATGCCCGTAATGGTGTTCTTATTACAGAAGGTAGTGTTAAAGGTTTACAA
CCATCTGTAGGTCCCAAACAAGCTAGTCTTAATGGAGTCACATTAATTGGAGAAGCCGTAAAAACACAGTTCAATTATTATAAG
AAAGTTGATGGTGTTGTCCAACAATTACCTGAAACTTACTTTACTCAGAGTAGAAATTTACAAGAATTTAAACCCAGGAGTCAA
ATGGAAATTGATTTCTTAGAATTAGCTATGGATGAATTCATTGAACGGTATAAATTAGAAGGCTATGCCTTCGAACATATCGTT
TATGGAGATTTTAGTCATAGTCAGTTAGGTGGTTTACATCTACTGATTGGACTAGCTAAACGTTTTAAGGAATCACCTTTTGAA
TTAGAAGATTTTATTCCTATGGACAGTACAGTTAAAAACTATTTCATAACAGATGCGCAAACAGGTTCATCTAAGTGTGTGTGT
TCTGTTATTGATTTATTACTTGATGATTTTGTTGAAATAATAAAATCCCAAGATTTATCTGTAGTTTCTAAGGTTGTCAAAGTG
ACTATTGACTATACAGAAATTTCATTTATGCTTTGGTGTAAAGATGGCCATGTAGAAACATTTTACCCAAAATTACAATCTAGT
CAAGCGTGGCAACCGGGTGTTGCTATGCCTAATCTTTACAAAATGCAAAGAATGCTATTAGAAAAGTGTGACCTTCAAAATTAT
GGTGATAGTGCAACATTACCTAAAGGCATAATGATGAATGTCGCAAAATATACTCAACTGTGTCAATATTTAAACACATTAACA
TTAGCTGTACCCTATAATATGAGAGTTATACATTTTGGTGCTGGTTCTGATAAAGGAGTTGCACCAGGTACAGCTGTTTTAAGA
CAGTGGTTGCCTACGGGTACGCTGCTTGTCGATTCAGATCTTAATGACTTTGTCTCTGATGCAGATTCAACTTTGATTGGTGAT
TGTGCAACTGTACATACAGCTAATAAATGGGATCTCATTATTAGTGATATGTACGACCCTAAGACTAAAAATGTTACAAAAGAA
AATGACTCTAAAGAGGGTTTTTTCACTTACATTTGTGGGTTTATACAACAAAAGCTAGCTCTTGGAGGTTCCGTGGCTATAAAG
ATAACAGAACATTCTTGGAATGCTGATCTTTATAAGCTCATGGGACACTTCGCATGGTGGACAGCCTTTGTTACTAATGTGAAT
GCGTCATCATCTGAAGCATTTTTAATTGGATGTAATTATCTTGGCAAACCACGCGAACAAATAGATGGTTATGTCATGCATGCA
AATTACATATTTTGGAGGAATACAAATCCAATTCAGTTGTCTTCCTATTCTTTATTTGACATGAGTAAATTTCCCCTTAAATTA
AGGGGTACTGCTGTTATGTCTTTAAAAGAAGGTCAAATCAATGATATGATTTTATCTCTTCTTAGTAAAGGTAGACTTATAATT
AGAGAAAACAACAGAGTTGTTATTTCTAGTGATGTTCTTGTTAACAACTAAACGAACAATGTTTGTTTTTCTTGTTTTATTGCC
ACTAGTCTCTAGTCAGTGTGTTAATCTTACAACCAGAACTCAATTACCCCCTGCATACACTAATTCTTTCACACGTGGTGTTTA
TTACCCTGACAAAGTTTTCAGATCCTCAGTTTTACATTCAACTCAGGACTTGTTCTTACCTTTCTTTTCCAATGTTACTTGGTT
CCATGCTATACATGTCTCTGGGACCAATGGTACTAAGAGGTTTGATAACCCTGTCCTACCATTTAATGATGGTGTTTATTTTGC
TTCCACTGAGAAGTCTAACATAATAAGAGGCTGGATTTTTGGTACTACTTTAGATTCGAAGACCCAGTCCCTACTTATTGTTAA
TAACGCTACTAATGTTGTTATTAAAGTCTGTGAATTTCAATTTTGTAATGATCCATTTTTGGGTGTTTATTACCACAAAAACAA

CAAAAGTTGGATGGAAAGTGAGTTCAGAGTTTATTCTAGTGCGAATAATTGCACTTTTGAATATGTCTCTCAGCCTTTTCTTAT
GGACCTTGAAGGAAAACAGGGTAATTTCAAAAATCTTAGGGAATTTGTGTTTAAGAATATTGATGGTTATTTTAAAATATATTC
TAAGCACACGCCTATTAATTTAGTGCGTGATCTCCCTCAGGGTTTTTCGGCTTTAGAACCATTGGTAGATTTGCCAATAGGTAT
TAACATCACTAGGTTTCAAACTTTACTTGCTTTACATAGAAGTTATTTGACTCCTGGTGATTCTTCTTCAGGTTGGACAGCTGG
TGCTGCAGCTTATTATGTGGGTTATCTTCAACCTAGGACTTTTCTATTAAAATATAATGAAAATGGAACCATTACAGATGCTGT
AGACTGTGCACTTGACCCTCTCTCAGAAACAAAGTGTACGTTGAAATCCTTCACTGTAGAAAAAGGAATCTATCAAACTTCTAA
CTTTAGAGTCCAACCAACAGAATCTATTGTTAGATTTCCTAATATTACAAACTTGTGCCCTTTTGGTGAAGTTTTTAACGCCAC
CAGATTTGCATCTGTTTATGCTTGGAACAGGAAGAGAATCAGCAACTGTGTTGCTGATTATTCTGTCCTATATAATTCCGCATC
ATTTTCCACTTTTAAGTGTTATGGAGTGTCTCCTACTAAATTAAATGATCTCTGCTTTACTAATGTCTATGCAGATTCATTTGT
AATTAGAGGTGATGAAGTCAGACAAATCGCTCCAGGGCAAACTGGAAAGATTGCTGATTATAATTATAAATTACCAGATGATTT
TACAGGCTGCGTTATAGCTTGGAATTCTAACAATCTTGATTCTAAGGTTGGTGGTAATTATAATTACCTGTATAGATTGTTTAG
GAAGTCTAATCTCAAACCTTTTGAGAGAGATATTTCAACTGAAATCTATCAGGCCGGTAGCACACCTTGTAATGGTGTTGAAGG
TTTTAATTGTTACTTTCCTTTACAATCATATGGTTTCCAACCCACTAATGGTGTTGGTTACCAACCATACAGAGTAGTAGTACT
TTCTTTTGAACTTCTACATGCACCAGCAACTGTTTGTGGACCTAAAAAGTCTACTAATTTGGTTAAAAACAAATGTGTCAATTT
CAACTTCAATGGTTTAACAGGCACAGGTGTTCTTACTGAGTCTAACAAAAAGTTTCTGCCTTTCCAACAATTTGGCAGAGACAT
TGCTGACACTACTGATGCTGTCCGTGATCCACAGACACTTGAGATTCTTGACATTACACCATGTTCTTTTGGTGGTGTCAGTGT
TATAACACCAGGAACAAATACTTCTAACCAGGTTGCTGTTCTTTATCAGGATGTTAACTGCACAGAAGTCCCTGTTGCTATTCA
TGCAGATCAACTTACTCCTACTTGGCGTGTTTATTCTACAGGTTCTAATGTTTTTCAAACACGTGCAGGCTGTTTAATAGGGGC
TGAACATGTCAACAACTCATATGAGTGTGACATACCCATTGGTGCAGGTATATGCGCTAGTTATCAGACTCAGACTAATTCTCC
TCGGCGGGCACGTAGTGTAGCTAGTCAATCCATCATTGCCTACACTATGTCACTTGGTGCAGAAAATTCAGTTGCTTACTCTAA
TAACTCTATTGCCATACCCACAAATTTTACTATTAGTGTTACCACAGAAATTCTACCAGTGTCTATGACCAAGACATCAGTAGA
TTGTACAATGTACATTTGTGGTGATTCAACTGAATGCAGCAATCTTTTGTTGCAATATGGCAGTTTTTGTACACAATTAAACCG
TGCTTTAACTGGAATAGCTGTTGAACAAGACAAAAACACCCAAGAAGTTTTTGCACAAGTCAAACAAATTTACAAAACACCACC
AATTAAAGATTTTGGTGGTTTTAATTTTTCACAAATATTACCAGATCCATCAAAACCAAGCAAGAGGTCATTTATTGAAGATCT
ACTTTTCAACAAAGTGACACTTGCAGATGCTGGCTTCATCAAACAATATGGTGATTGCCTTGGTGATATTGCTGCTAGAGACCT
CATTTGTGCACAAAAGTTTAACGGCCTTACTGTTTTGCCACCTTTGCTCACAGATGAAATGATTGCTCAATACACTTCTGCACT
GTTAGCGGGTACAATCACTTCTGGTTGGACCTTTGGTGCAGGTGCTGCATTACAAATACCATTTGCTATGCAAATGGCTTATAG
GTTTAATGGTATTGGAGTTACACAGAATGTTCTCTATGAGAACCAAAAATTGATTGCCAACCAATTTAATAGTGCTATTGGCAA
AATTCAAGACTCACTTTCTTCCACAGCAAGTGCACTTGGAAAACTTCAAGATGTGGTCAACCAAAATGCACAAGCTTTAAACAC
GCTTGTTAAACAACTTAGCTCCAATTTTGGTGCAATTTCAAGTGTTTTAAATGATATCCTTTCACGTCTTGACAAAGTTGAGGC
TGAAGTGCAAATTGATAGGTTGATCACAGGCAGACTTCAAAGTTTGCAGACATATGTGACTCAACAATTAATTAGAGCTGCAGA
AATCAGAGCTTCTGCTAATCTTGCTGCTACTAAAATGTCAGAGTGTGTACTTGGACAATCAAAAAGAGTTGATTTTTGTGGAAA
GGGCTATCATCTTATGTCCTTCCCTCAGTCAGCACCTCATGGTGTAGTCTTCTTGCATGTGACTTATGTCCCTGCACAAGAAAA
GAACTTCACAACTGCTCCTGCCATTTGTCATGATGGAAAAGCACACTTTCCTCGTGAAGGTGTCTTTGTTTCAAATGGCACACA
CTGGTTTGTAACACAAAGGAATTTTTATGAACCACAAATCATTACTACAGACAACACATTTGTGTCTGGTAACTGTGATGTTGT
AATAGGAATTGTCAACAACACAGTTTATGATCCTTTGCAACCTGAATTAGACTCATTCAAGGAGGAGTTAGATAAATATTTTAA
GAATCATACATCACCAGATGTTGATTTAGGTGACATCTCTGGCATTAATGCTTCAGTTGTAAACATTCAAAAAGAAATTGACCG
CCTCAATGAGGTTGCCAAGAATTTAAATGAATCTCTCATCGATCTCCAAGAACTTGGAAAGTATGAGCAGTATATAAAATGGCC
ATGGTACATTTGGCTAGGTTTTATAGCTGGCTTGATTGCCATAGTAATGGTGACAATTATGCTTTGCTGTATGACCAGTTGCTG
TAGTTGTCTCAAGGGCTGTTGTTCTTGTGGATCCTGCTGCAAATTTGATGAAGACGACTCTGAGCCAGTGCTCAAAGGAGTCAA
ATTACATTACACATAAACGAACTTATGGATTTGTTTATGAGAATCTTCACAATTGGAACTGTAACTTTGAAGCAAGGTGAAATC
AAGGATGCTACTCCTTCAGATTTTGTTCGCGCTACTGCAACGATACCGATACAAGCCTCACTCCCTTTCGGATGGCTTATTGTT
GGCGTTGCACTTCTTGCTGTTTTTCAGAGCGCTTCCAAAATCATAACCCTCAAAAAGAGATGGCAACTAGCACTCTCCAAGGGT
GTTCACTTTGTTTGCAACTTGCTGTTGTTGTTTGTAACAGTTTACTCACACCTTTTGCTCGTTGCTGCTGGCCTTGAAGCCCCT
TTTCTCTATCTTTATGCTTTAGTCTACTTCTTGCAGAGTATAAACTTTGTAAGAATAATAATGAGGCTTTGGCTTTGCTGGAAA
TGCCGTTCCAAAAACCCATTACTTTATGATGCCAACTATTTTCTTTGCTGGCATACTAATTGTTACGACTATTGTATACCTTAC
AATAGTGTAACTTCTTCAATTGTCATTACTTCAGGTGATGGCACAACAAGTCCTATTTCTGAACATGACTACCAGATTGGTGGT
TATACTGAAAAATGGGAATCTGGAGTAAAAGACTGTGTTGTATTACACAGTTACTTCACTTCAGACTATTACCAGCTGTACTCA
ACTCAATTGAGTACAGACACTGGTGTTGAACATGTTACCTTCTTCATCTACAATAAAATTGTTGATGAGCCTGAAGAACATGTC
CAAATTCACACAATCGACGGTTCATCCGGAGTTGTTAATCCAGTAATGGAACCAATTTATGATGAACCGACGACGACTACTAGC
GTGCCTTTGTAAGCACAAGCTGATGAGTACGAACTTATGTACTCATTCGTTTCGGAAGAGACAGGTACGTTAATAGTTAATAGC
GTACTTCTTTTTCTTGCTTTCGTGGTATTCTTGCTAGTTACACTAGCCATCCTTACTGCGCTTCGATTGTGTGCGTACTGCTGC
AATATTGTTAACGTGAGTCTTGTAAAACCTTCTTTTTACGTTTACTCTCGTGTTAAAAATCTGAATTCTTCTAGAGTTCCTGAT
CTTCTGGTCTAAACGAACTAAATATTATATTAGTTTTTCTGTTTGGAACTTTAATTTTAGCCATGGCAGATTCCAACGGTACTA
TTACCGTTGAAGAGCTTAAAAAGCTCCTTGAACAATGGAACCTAGTAATAGGTTTCCTATTCCTTACATGGATTTGTCTTCTAC
AATTTGCCTATGCCAACAGGAATAGGTTTTTGTATATAATTAAGTTAATTTTCCTCTGGCTGTTATGGCCAGTAACTTTAGCTT
GTTTTGTGCTTGCTGCTGTTTACAGAATAAATTGGATCACCGGTGGAATTGCTATCGCAATGGCTTGTCTTGTAGGCTTGATGT
GGCTCAGCTACTTCATTGCTTCTTTCAGACTGTTTGCGCGTACGCGTTCCATGTGGTCATTCAATCCAGAAACTAACATTCTTC
TCAACGTGCCACTCCATGGCACTATTCTGACCAGACCGCTTCTAGAAAGTGAACTCGTAATCGGAGCTGTGATCCTTCGTGGAC
ATCTTCGTATTGCTGGACACCATCTAGGACGCTGTGACATCAAGGACCTGCCTAAAGAAATCACTGTTGCTACATCACGAACGC
TTTCTTATTACAAATTGGGAGCTTCGCAGCGTGTAGCAGGTGACTCAGGTTTTGCTGCATACAGTCGCTACAGGATTGGCAACT

```
ATAAATTAAACACAGACCATTCCAGTAGCAGTGACAATATTGCTTTGCTTGTACAGTAAGTGACAACAGATGTTTCATCTCGTT
GACTTTCAGGTTACTATAGCAGAGATATTACTAATTATTATGAGGACTTTTAAAGTTTCCATTTGGAATCTTGATTACATCATA
AACCTCATAATTAAAAATTTATCTAAGTCACTAACTGAGAATAAATATTCTCAATTAGATGAAGAGCAACCAATGGAGATTGAT
TAAACGAACATGAAAATTATTCTTTTCTTGGCACTGATAACACTCGCTACTTGTGAGCTTTATCACTACCAAGAGTGTGTTAGA
GGTACAACAGTACTTTTAAAAGAACCTTGCTCTTCTGGAACATACGAGGGCAATTCACCATTTCATCCTCTAGCTGATAACAAA
TTTGCACTGACTTGCTTTAGCACTCAATTTGCTTTTGCTTGTCCTGACGGCGTAAAACACGTCTATCAGTTACGTGCCAGATCA
GTTTCACCTAAACTGTTCATCAGACAAGAGGAAGTTCAAGAACTTTACTCTCCAATTTTTCTTATTGTTGCGGCAATAGTGTTT
ATAACACTTTGCTTCACACTCAAAAGAAAGACAGAATGATTGAACTTTCATTAATTGACTTCTATTTGTGCTTTTTAGCCTTTC
TGCTATTCCTTGTTTTAATTATGCTTATTATCTTTTGGTTCTCACTTGAACTGCAAGATCATAATGAAACTTGTCACGCCTAAA
CGAACATGAAATTTCTTGTTTTCTTAGGAATCATCACAACTGTAGCTGCATTTCACCAAGAATGTAGTTTACAGTCATGTACTC
AACATCAACCATATGTAGTTGATGACCCGTGTCCTATTCACTTCTATTCTAAATGGTATATTAGAGTAGGAGCTAGAAAATCAG
CACCTTTAATTGAATTGTGCGTGGATGAGGCTGGTTCTAAATCACCCATTCAGTACATCGATATCGGTAATTATACAGTTTCCT
GTTTACCTTTTACAATTAATTGCCAGGAACCTAAATTGGGTAGTCTTGTAGTGCGTTGTTCGTTCTATGAAGACTTTTTAGAGT
ATCATGACGTTCGTGTTGTTTTAGATTTCATCTAAACGAACAAACTAAAATGTCTGATAATGGACCCCAAAATCAGCGAAATGC
ACCCCGCATTACGTTTGGTGGACCCTCAGATTCAACTGGCAGTAACCAGAATGGAGAACGCAGTGGGGCGCGATCAAAACAACG
TCGGCCCCAAGGTTTACCCAATAATACTGCGTCTTGGTTCACCGCTCTCACTCAACATGGCAAGGAAGACCTTAAATTCCCTCG
AGGACAAGGCGTTCCAATTAACACCAATAGCAGTCCAGATGACCAAATTGGCTACTACCGAAGAGCTACCAGACGAATTCGTGG
TGGTGACGGTAAAATGAAAGATCTCAGTCCAAGATGGTATTTCTACTACCTAGGAACTGGGCCAGAAGCTGGACTTCCCTATGG
TGCTAACAAAGACGGCATCATATGGGTTGCAACTGAGGGAGCCTTGAATACACCAAAAGATCACATTGGCACCCGCAATCCTGC
TAACAATGCTGCAATCGTGCTACAACTTCCTCAAGGAACAACATTGCCAAAAGGCTTCTACGCAGAAGGGAGCAGAGGCGGCAG
TCAAGCCTCTTCTCGTTCCTCATCACGTAGTCGCAACAGTTCAAGAAATTCAACTCCAGGCAGCAGTAGGGGAACTTCTCCTGC
TAGAATGGCTGGCAATGGCGGTGATGCTGCTCTTGCTTTGCTGCTGCTTGACAGATTGAACCAGCTTGAGAGCAAAATGTCTGG
TAAAGGCCAACAACAACAAGGCCAAACTGTCACTAAGAAATCTGCTGCTGAGGCTTCTAAGAAGCCTCGGCAAAAACGTACTGC
CACTAAAGCATACAATGTAACACAAGCTTTCGGCAGACGTGGTCCAGAACAAACCCAAGGAAATTTTGGGGACCAGGAACTAAT
CAGACAAGGAACTGATTACAAACATTGGCCGCAAATTGCACAATTTGCCCCCAGCGCTTCAGCGTTCTTCGGAATGTCGCGCAT
TGGCATGGAAGTCACACCTTCGGGAACGTGGTTGACCTACACAGGTGCCATCAAATTGGATGACAAAGATCCAAATTTCAAAGA
TCAAGTCATTTTGCTGAATAAGCATATTGACGCATACAAAACATTCCCACCAACAGAGCCTAAAAAGGACAAAAAGAAGAAGGC
TGATGAAACTCAAGCCTTACCGCAGAGACAGAAGAAACAGCAAACTGTGACTCTTCTTCCTGCTGCAGATTTGGATGATTTCTC
CAAACAATTGCAACAATCCATGAGCAGTGCTGACTCAACTCAGGCCTAAACTCATGCAGACCACACAAGGCAGATGGGCTATAT
AAACGTTTTCGCTTTTCCGTTTACGATATATAGTCTACTCTTGTGCAGAATGAATTCTCGTAACTACATAGCACAAGTAGATGT
AGTTAACTTTAATCTCACATAGCAATCTTTAATCAGTGTGTAACATTAGGGAGGACTTGAAAGAGCCACCACATTTTCACCGAG
GCCACGCGGAGTACGATCGAGTGTACAGTGAACAATGCTAGGGAGAGCTGCCTATATGGAAGAGCCCTAATGTGTAAAATTAAT
TTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Spike protein primer:

```
% DNA sequence that you know starts the gene
spike_FWprimer = 'ATGTTTGTTTTTCTTGTTTTATTG';
```

Codon information - reorganize into a three column cell array for easier accessibility:

```
% NOTE: one cell in codon_list (imported above) is a line of information
% containing 1) the nucleotides, the name, and abbreviation for
% each amino acid. We want each of these three items in a separate column

% Preallocate cell array
codons = cell(length(codon_list),3);

% Loop through each entry in the list of codons
for k = 1:length(codon_list)
    % Read character arrays from the k-th row
    codon_row = codon_list{k};
```

```
    % Determine indices where commas occur
    ind = strfind(codon_row,',');

    % First column: bases
    codons{k,1} = codon_row(1:ind(1)-1);
    % Second column: corresponding codon name
    codons{k,2} = codon_row(ind(1)+1:ind(2)-1);
    % Third column: Amino acid
    codons{k,3} = codon_row(ind(2)+1:end);
end
```

Note: the codons listed are given in terms of *DNA bases*.

Question 1: how many bases are in the SARS-CoV-2 genome?

```
% Number of bases equals number of elements in the character vector
numBases = length(sarsCov2_genome)
```

```
numBases = 29903
```

Question 2: how many of *each* base appear in the genome?

Note: for this problem we consider only one strand.

```
% Step 1: Determine how many of each base are in the genome
base_A = length(strfind(sarsCov2_genome, 'A'))
```

```
base_A = 8954
```

```
base_T = length(strfind(sarsCov2_genome, 'T'))
```

```
base_T = 9594
```

```
base_C = length(strfind(sarsCov2_genome, 'C'))
```

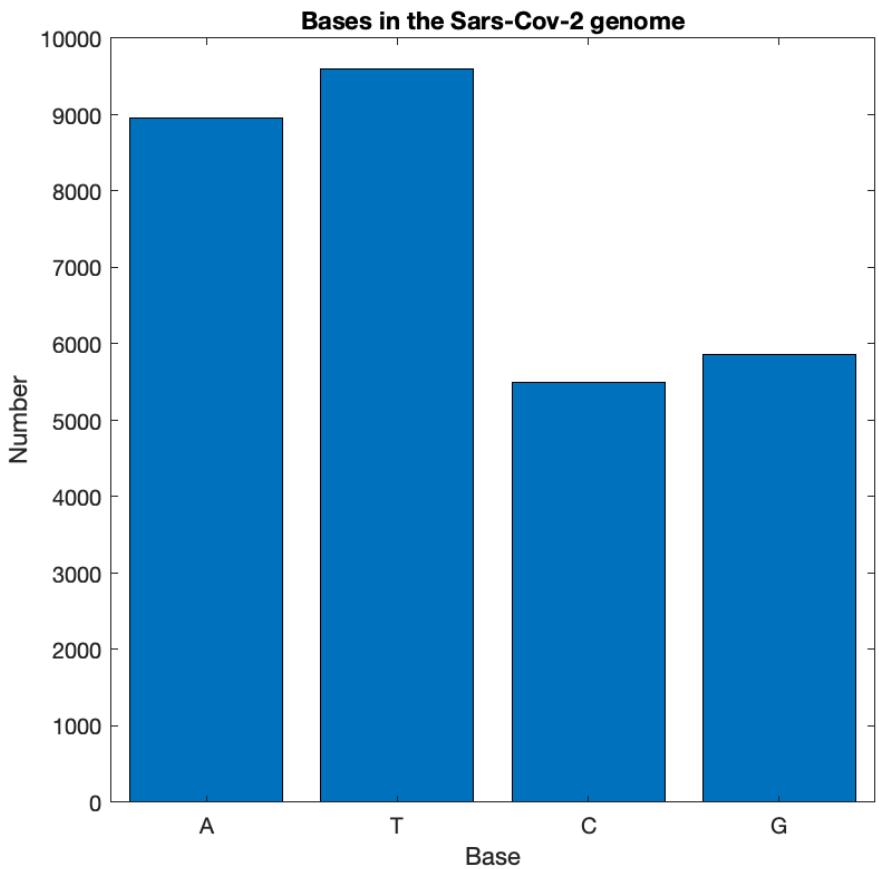
```
base_C = 5492
```

```
base_G = length(strfind(sarsCov2_genome, 'G'))
```

```
base_G = 5863
```

```
% Step 2: Plotting
% Create cell array containing each x label
xval = {'A','T','C','G'};
% Convert cell array to categorical and then reorder to be in the same
% order
xval2 = categorical(xval);
```

```
xval2 = reordercats(xval2,xval)';

% Make bar graph
bar(xval2,[base_A,base_T,base_C,base_G]);
title('Bases in the Sars-Cov-2 genome')
axis square
ylabel('Number');
xlabel('Base');
```



Question 3: given a primer, what is the DNA sequence for the spike glycoprotein? How many bases long is it?

```
% Step 1: Find the index in the genome where the gene starts
DNA_start = strfind(sarsCov2_genome, spike_FWprimer)
```

```
DNA_start = 21563
```

```
% Define the DNA sequence starting at taht position
spike_DNA = sarsCov2_genome(DNA_start:end)
```

```
spike_DNA =
'ATGTTTGTTTTTCTTGTTTTATTGCCACTAGTCTCTAGTCAGTGTGTTAATCTTACAACCAGAACTCAATTACCCCCTGCATA
```

CACTAATTCTTTCACACGTGGTGTTTATTACCCTGACAAAGTTTTCAGATCCTCAGTTTTACATTCAACTCAGGACTTGTTCTT
ACCTTTCTTTTCCAATGTTACTTGGTTCCATGCTATACATGTCTCTGGGACCAATGGTACTAAGAGGTTTGATAACCCTGTCCT
ACCATTTAATGATGGTGTTTATTTTGCTTCCACTGAGAAGTCTAACATAATAAGAGGCTGGATTTTTGGTACTACTTTAGATTC
GAAGACCCAGTCCCTACTTATTGTTAATAACGCTACTAATGTTGTTATTAAAGTCTGTGAATTTCAATTTTGTAATGATCCATT
TTTGGGTGTTTATTACCACAAAAACAACAAAAGTTGGATGGAAAGTGAGTTCAGAGTTTATTCTAGTGCGAATAATTGCACTTT
TGAATATGTCTCTCAGCCTTTTCTTATGGACCTTGAAGGAAAACAGGGTAATTTCAAAAATCTTAGGGAATTTGTGTTTAAGAA
TATTGATGGTTATTTTAAAATATATTCTAAGCACACGCCTATTAATTTAGTGCGTGATCTCCCTCAGGGTTTTTCGGCTTTAGA
ACCATTGGTAGATTTGCCAATAGGTATTAACATCACTAGGTTTCAAACTTTACTTGCTTTACATAGAAGTTATTTGACTCCTGG
TGATTCTTCTTCAGGTTGGACAGCTGGTGCTGCAGCTTATTATGTGGGTTATCTTCAACCTAGGACTTTTCTATTAAAATATAA
TGAAAATGGAACCATTACAGATGCTGTAGACTGTGCACTTGACCCTCTCTCAGAAACAAAGTGTACGTTGAAATCCTTCACTGT
AGAAAAAGGAATCTATCAAACTTCTAACTTTAGAGTCCAACCAACAGAATCTATTGTTAGATTTCCTAATATTACAAACTTGTG
CCCTTTTGGTGAAGTTTTTAACGCCACCAGATTTGCATCTGTTTATGCTTGGAACAGGAAGAGAATCAGCAACTGTGTTGCTGA
TTATTCTGTCCTATATAATTCCGCATCATTTTCCACTTTTAAGTGTTATGGAGTGTCTCCTACTAAATTAAATGATCTCTGCTT
TACTAATGTCTATGCAGATTCATTTGTAATTAGAGGTGATGAAGTCAGACAAATCGCTCCAGGGCAAACTGGAAAGATTGCTGA
TTATAATTATAAATTACCAGATGATTTTACAGGCTGCGTTATAGCTTGGAATTCTAACAATCTTGATTCTAAGGTTGGTGGTAA
TTATAATTACCTGTATAGATTGTTTAGGAAGTCTAATCTCAAACCTTTTGAGAGAGATATTTCAACTGAAATCTATCAGGCCGG
TAGCACACCTTGTAATGGTGTTGAAGGTTTTAATTGTTACTTTCCTTTACAATCATATGGTTTCCAACCCACTAATGGTGTTGG
TTACCAACCATACAGAGTAGTAGTACTTTCTTTTGAACTTCTACATGCACCAGCAACTGTTTGTGGACCTAAAAAGTCTACTAA
TTTGGTTAAAAACAAATGTGTCAATTTCAACTTCAATGGTTTAACAGGCACAGGTGTTCTTACTGAGTCTAACAAAAAGTTTCT
GCCTTTCCAACAATTTGGCAGAGACATTGCTGACACTACTGATGCTGTCCGTGATCCACAGACACTTGAGATTCTTGACATTAC
ACCATGTTCTTTTGGTGGTGTCAGTGTTATAACACCAGGAACAAATACTTCTAACCAGGTTGCTGTTCTTTATCAGGATGTTAA
CTGCACAGAAGTCCCTGTTGCTATTCATGCAGATCAACTTACTCCTACTTGGCGTGTTTATTCTACAGGTTCTAATGTTTTTCA
AACACGTGCAGGCTGTTTAATAGGGGCTGAACATGTCAACAACTCATATGAGTGTGACATACCCATTGGTGCAGGTATATGCGC
TAGTTATCAGACTCAGACTAATTCTCCTCGGCGGGCACGTAGTGTAGCTAGTCAATCCATCATTGCCTACACTATGTCACTTGG
TGCAGAAAATTCAGTTGCTTACTCTAATAACTCTATTGCCATACCCACAAATTTTACTATTAGTGTTACCACAGAAATTCTACC
AGTGTCTATGACCAAGACATCAGTAGATTGTACAATGTACATTTGTGGTGATTCAACTGAATGCAGCAATCTTTTGTTGCAATA
TGGCAGTTTTTGTACACAATTAAACCGTGCTTTAACTGGAATAGCTGTTGAACAAGACAAAAACACCCAAGAAGTTTTTGCACA
AGTCAAACAAATTTACAAAACACCACCAATTAAAGATTTTGGTGGTTTTAATTTTTCACAAATATTACCAGATCCATCAAAACC
AAGCAAGAGGTCATTTATTGAAGATCTACTTTTCAACAAAGTGACACTTGCAGATGCTGGCTTCATCAAACAATATGGTGATTG
CCTTGGTGATATTGCTGCTAGAGACCTCATTTGTGCACAAAAGTTTAACGGCCTTACTGTTTTGCCACCTTTGCTCACAGATGA
AATGATTGCTCAATACACTTCTGCACTGTTAGCGGGTACAATCACTTCTGGTTGGACCTTTGGTGCAGGTGCTGCATTACAAAT
ACCATTTGCTATGCAAATGGCTTATAGGTTTAATGGTATTGGAGTTACACAGAATGTTCTCTATGAGAACCAAAAATTGATTGC
CAACCAATTTAATAGTGCTATTGGCAAAATTCAAGACTCACTTTCTTCCACAGCAAGTGCACTTGGAAAACTTCAAGATGTGGT
CAACCAAAATGCACAAGCTTTAAACACGCTTGTTAAACAACTTAGCTCCAATTTTGGTGCAATTTCAAGTGTTTTAAATGATAT
CCTTTCACGTCTTGACAAAGTTGAGGCTGAAGTGCAAATTGATAGGTTGATCACAGGCAGACTTCAAAGTTTGCAGACATATGT
GACTCAACAATTAATTAGAGCTGCAGAAATCAGAGCTTCTGCTAATCTTGCTGCTACTAAAATGTCAGAGTGTGTACTTGGACA
ATCAAAAAGAGTTGATTTTTGTGGAAAGGGCTATCATCTTATGTCCTTCCCTCAGTCAGCACCTCATGGTGTAGTCTTCTTGCA
TGTGACTTATGTCCCTGCACAAGAAAAGAACTTCACAACTGCTCCTGCCATTTGTCATGATGGAAAAGCACACTTTCCTCGTGA
AGGTGTCTTTGTTTCAAATGGCACACACTGGTTTGTAACACAAAGGAATTTTTATGAACCACAAATCATTACTACAGACAACAC
ATTTGTGTCTGGTAACTGTGATGTTGTAATAGGAATTGTCAACAACACAGTTTATGATCCTTTGCAACCTGAATTAGACTCATT
CAAGGAGGAGTTAGATAAATATTTTAAGAATCATACATCACCAGATGTTGATTTAGGTGACATCTCTGGCATTAATGCTTCAGT
TGTAAACATTCAAAAAGAAATTGACCGCCTCAATGAGGTTGCCAAGAATTTAAATGAATCTCTCATCGATCTCCAAGAACTTGG
AAAGTATGAGCAGTATATAAAATGGCCATGGTACATTTGGCTAGGTTTTATAGCTGGCTTGATTGCCATAGTAATGGTGACAAT
TATGCTTTGCTGTATGACCAGTTGCTGTAGTTGTCTCAAGGGCTGTTGTTCTTGTGGATCCTGCTGCAAATTTGATGAAGACGA
CTCTGAGCCAGTGCTCAAAGGAGTCAAATTACATTACACATAAACGAACTTATGGATTTGTTTATGAGAATCTTCACAATTGGA
ACTGTAACTTTGAAGCAAGGTGAAATCAAGGATGCTACTCCTTCAGATTTTGTTCGCGCTACTGCAACGATACCGATACAAGCC
TCACTCCCTTTCGGATGGCTTATTGTTGGCGTTGCACTTCTTGCTGTTTTTCAGAGCGCTTCCAAAATCATAACCCTCAAAAAG
AGATGGCAACTAGCACTCTCCAAGGGTGTTCACTTTGTTTGCAACTTGCTGTTGTTGTTTGTAACAGTTTACTCACACCTTTTG
CTCGTTGCTGCTGGCCTTGAAGCCCCTTTTCTCTATCTTTATGCTTTAGTCTACTTCTTGCAGAGTATAAACTTTGTAAGAATA
ATAATGAGGCTTTGGCTTTGCTGGAAATGCCGTTCCAAAAACCCATTACTTTATGATGCCAACTATTTTCTTTGCTGGCATACT
AATTGTTACGACTATTGTATACCTTACAATAGTGTAACTTCTTCAATTGTCATTACTTCAGGTGATGGCACAACAAGTCCTATT
TCTGAACATGACTACCAGATTGGTGGTTATACTGAAAAATGGGAATCTGGAGTAAAAGACTGTGTTGTATTACACAGTTACTTC
ACTTCAGACTATTACCAGCTGTACTCAACTCAATTGAGTACAGACACTGGTGTTGAACATGTTACCTTCTTCATCTACAATAAA
ATTGTTGATGAGCCTGAAGAACATGTCCAAATTCACACAATCGACGGTTCATCCGGAGTTGTTAATCCAGTAATGGAACCAATT
TATGATGAACCGACGACGACTACTAGCGTGCCTTTGTAAGCACAAGCTGATGAGTACGAACTTATGTACTCATTCGTTTCGGAA
GAGACAGGTACGTTAATAGTTAATAGCGTACTTCTTTTTCTTGCTTTCGTGGTATTCTTGCTAGTTACACTAGCCATCCTTACT
GCGCTTCGATTGTGTGCGTACTGCTGCAATATTGTTAACGTGAGTCTTGTAAAACCTTCTTTTTACGTTTACTCTCGTGTTAAA
AATCTGAATTCTTCTAGAGTTCCTGATCTTCTGGTCTAAACGAACTAAATATTATATTAGTTTTTCTGTTTGGAACTTTAATTT
TAGCCATGGCAGATTCCAACGGTACTATTACCGTTGAAGAGCTTAAAAAGCTCCTTGAACAATGGAACCTAGTAATAGGTTTCC
TATTCCTTACATGGATTTGTCTTCTACAATTTGCCTATGCCAACAGGAATAGGTTTTTGTATATAATTAAGTTAATTTTCCTCT
GGCTGTTATGGCCAGTAACTTTAGCTTGTTTTGTGCTTGCTGCTGTTTACAGAATAAATTGGATCACCGGTGGAATTGCTATCG

```
CAATGGCTTGTCTTGTAGGCTTGATGTGGCTCAGCTACTTCATTGCTTCTTTCAGACTGTTTGCGCGTACGCGTTCCATGTGGT
CATTCAATCCAGAAACTAACATTCTTCTCAACGTGCCACTCCATGGCACTATTCTGACCAGACCGCTTCTAGAAAGTGAACTCG
TAATCGGAGCTGTGATCCTTCGTGGACATCTTCGTATTGCTGGACACCATCTAGGACGCTGTGACATCAAGGACCTGCCTAAAG
AAATCACTGTTGCTACATCACGAACGCTTTCTTATTACAAATTGGGAGCTTCGCAGCGTGTAGCAGGTGACTCAGGTTTTGCTG
CATACAGTCGCTACAGGATTGGCAACTATAAATTAAACACAGACCATTCCAGTAGCAGTGACAATATTGCTTTGCTTGTACAGT
AAGTGACAACAGATGTTTCATCTCGTTGACTTTCAGGTTACTATAGCAGAGATATTACTAATTATTATGAGGACTTTTAAAGTT
TCCATTTGGAATCTTGATTACATCATAAACCTCATAATTAAAAATTTATCTAAGTCACTAACTGAGAATAAATATTCTCAATTA
GATGAAGAGCAACCAATGGAGATTGATTAAACGAACATGAAAATTATTCTTTTCTTGGCACTGATAACACTCGCTACTTGTGAG
CTTTATCACTACCAAGAGTGTGTTAGAGGTACAACAGTACTTTTAAAAGAACCTTGCTCTTCTGGAACATACGAGGGCAATTCA
CCATTTCATCCTCTAGCTGATAACAAATTTGCACTGACTTGCTTTAGCACTCAATTTGCTTTTGCTTGTCCTGACGGCGTAAAA
CACGTCTATCAGTTACGTGCCAGATCAGTTTCACCTAAACTGTTCATCAGACAAGAGGAAGTTCAAGAACTTTACTCTCCAATT
TTTCTTATTGTTGCGGCAATAGTGTTTATAACACTTTGCTTCACACTCAAAAGAAAGACAGAATGATTGAACTTTCATTAATTG
ACTTCTATTTGTGCTTTTTAGCCTTTCTGCTATTCCTTGTTTTAATTATGCTTATTATCTTTTGGTTCTCACTTGAACTGCAAG
ATCATAATGAAACTTGTCACGCCTAAACGAACATGAAATTTCTTGTTTTCTTAGGAATCATCACAACTGTAGCTGCATTTCACC
AAGAATGTAGTTTACAGTCATGTACTCAACATCAACCATATGTAGTTGATGACCCGTGTCCTATTCACTTCTATTCTAAATGGT
ATATTAGAGTAGGAGCTAGAAAATCAGCACCTTTAATTGAATTGTGCGTGGATGAGGCTGGTTCTAAATCACCCATTCAGTACA
TCGATATCGGTAATTATACAGTTTCCTGTTTACCTTTTACAATTAATTGCCAGGAACCTAAATTGGGTAGTCTTGTAGTGCGTT
GTTCGTTCTATGAAGACTTTTTAGAGTATCATGACGTTCGTGTTGTTTTAGATTTCATCTAAACGAACAAACTAAAATGTCTGA
TAATGGACCCCAAAATCAGCGAAATGCACCCCGCATTACGTTTGGTGGACCCTCAGATTCAACTGGCAGTAACCAGAATGGAGA
ACGCAGTGGGGCGCGATCAAAACAACGTCGGCCCCAAGGTTTACCCAATAATACTGCGTCTTGGTTCACCGCTCTCACTCAACA
TGGCAAGGAAGACCTTAAATTCCCTCGAGGACAAGGCGTTCCAATTAACACCAATAGCAGTCCAGATGACCAAATTGGCTACTA
CCGAAGAGCTACCAGACGAATTCGTGGTGGTGACGGTAAAATGAAAGATCTCAGTCCAAGATGGTATTTCTACTACCTAGGAAC
TGGGCCAGAAGCTGGACTTCCCTATGGTGCTAACAAAGACGGCATCATATGGGTTGCAACTGAGGGAGCCTTGAATACACCAAA
AGATCACATTGGCACCCGCAATCCTGCTAACAATGCTGCAATCGTGCTACAACTTCCTCAAGGAACAACATTGCCAAAAGGCTT
CTACGCAGAAGGGAGCAGAGGCGGCAGTCAAGCCTCTTCTCGTTCCTCATCACGTAGTCGCAACAGTTCAAGAAATTCAACTCC
AGGCAGCAGTAGGGGAACTTCTCCTGCTAGAATGGCTGGCAATGGCGGTGATGCTGCTCTTGCTTTGCTGCTGCTTGACAGATT
GAACCAGCTTGAGAGCAAAATGTCTGGTAAAGGCCAACAACAACAAGGCCAAACTGTCACTAAGAAATCTGCTGCTGAGGCTTC
TAAGAAGCCTCGGCAAAAACGTACTGCCACTAAAGCATACAATGTAACACAAGCTTTCGGCAGACGTGGTCCAGAACAAACCCA
AGGAAATTTTGGGGACCAGGAACTAATCAGACAAGGAACTGATTACAAACATTGGCCGCAAATTGCACAATTTGCCCCCAGCGC
TTCAGCGTTCTTCGGAATGTCGCGCATTGGCATGGAAGTCACACCTTCGGGAACGTGGTTGACCTACACAGGTGCCATCAAATT
GGATGACAAAGATCCAAATTTCAAAGATCAAGTCATTTTGCTGAATAAGCATATTGACGCATACAAAACATTCCCACCAACAGA
GCCTAAAAAGGACAAAAAGAAGAAGGCTGATGAAACTCAAGCCTTACCGCAGAGACAGAAGAAACAGCAAACTGTGACTCTTCT
TCCTGCTGCAGATTTGGATGATTTCTCCAAACAATTGCAACAATCCATGAGCAGTGCTGACTCAACTCAGGCCTAAACTCATGC
AGACCACACAAGGCAGATGGGCTATATAAACGTTTTCGCTTTTCCGTTTACGATATATAGTCTACTCTTGTGCAGAATGAATTC
TCGTAACTACATAGCACAAGTAGATGTAGTTAACTTTAATCTCACATAGCAATCTTTAATCAGTGTGTAACATTAGGGAGGACT
TGAAAGAGCCACCACATTTTCACCGAGGCCACGCGGAGTACGATCGAGTGTACAGTGAACAATGCTAGGGAGAGCTGCCTATAT
GGAAGAGCCCTAATGTGTAAAATTAATTTTAGTAGTGCTATCCCCATGTGATTTTAATAGCTTCTTAGGAGAATGACAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAA'
```

```
% Step 2: determine length of gene
% Goal: find each instance of a "possible" stop codon. In this case, the
% first one that occurs after a "multiple of three bases" will be it.
possibleStops = [];
stopCodons = ['TAA'; 'TAG'; 'TGA']
```

```
stopCodons = 3×3 char array
    'TAA'
    'TAG'
    'TGA'
```

```
%Loop through sequence to find potential stop codons

for k = 1:3
    % stop codon to be considered
    tempStop = stopCodons(k,:);
```

```
    %find and add potential stop locations
    possibleStops = [possibleStops, strfind(spike_DNA, tempStop)];
end

%find the first stop position that is consisitent
possibleStops = sort(possibleStops, 'ascend')
```

```
possibleStops = 1×427
    29    36    48    87   117   228   237   240   258   261   285   294
299   302   329   360   363   372   384   393   408   411   465   483
492   504   537   552   567   582
```

```
%find positions in modulus of 3
lastCodon = possibleStops(mod(possibleStops,3) == 1);
lastBase = lastCodon(1) + 2;
% DNA sequence of spike protein
spike_DNA = spike_DNA(1:lastBase)
```

```
spike_DNA =
'ATGTTTGTTTTTCTTGTTTTATTGCCACTAGTCTCTAGTCAGTGTGTTAATCTTACAACCAGAACTCAATTACCCCCTGCATA
CACTAATTCTTTCACACGTGGTGTTTATTACCCTGACAAAGTTTTCAGATCCTCAGTTTTACATTCAACTCAGGACTTGTTCTT
ACCTTTCTTTTCCAATGTTACTTGGTTCCATGCTATACATGTCTCTGGGACCAATGGTACTAAGAGGTTTGATAACCCTGTCCT
ACCATTTAATGATGGTGTTTATTTTGCTTCCACTGAGAAGTCTAACATAATAAGAGGCTGGATTTTTGGTACTACTTTAGATTC
GAAGACCCAGTCCCTACTTATTGTTAATAACGCTACTAATGTTGTTATTAAAGTCTGTGAATTTCAATTTTGTAATGATCCATT
TTTGGGTGTTTATTACCACAAAAACAACAAAAGTTGGATGGAAAGTGAGTTCAGAGTTTATTCTAGTGCGAATAATTGCACTTT
TGAATATGTCTCTCAGCCTTTTCTTATGGACCTTGAAGGAAAACAGGGTAATTTCAAAAATCTTAGGGAATTTGTGTTTAAGAA
TATTGATGGTTATTTTAAAATATATTCTAAGCACACGCCTATTAATTTAGTGCGTGATCTCCCTCAGGGTTTTTCGGCTTTAGA
ACCATTGGTAGATTTGCCAATAGGTATTAACATCACTAGGTTTCAAACTTTACTTGCTTTACATAGAAGTTATTTGACTCCTGG
TGATTCTTCTTCAGGTTGGACAGCTGGTGCTGCAGCTTATTATGTGGGTTATCTTCAACCTAGGACTTTTCTATTAAAATATAA
TGAAAATGGAACCATTACAGATGCTGTAGACTGTGCACTTGACCCTCTCTCAGAAACAAAGTGTACGTTGAAATCCTTCACTGT
AGAAAAAGGAATCTATCAAACTTCTAACTTTAGAGTCCAACCAACAGAATCTATTGTTAGATTTCCTAATATTACAAACTTGTG
CCCTTTTGGTGAAGTTTTTAACGCCACCAGATTTGCATCTGTTTATGCTTGGAACAGGAAGAGAATCAGCAACTGTGTTGCTGA
TTATTCTGTCCTATATAATTCCGCATCATTTTCCACTTTTAAGTGTTATGGAGTGTCTCCTACTAAATTAAATGATCTCTGCTT
TACTAATGTCTATGCAGATTCATTTGTAATTAGAGGTGATGAAGTCAGACAAATCGCTCCAGGGCAAACTGGAAAGATTGCTGA
TTATAATTATAAATTACCAGATGATTTTACAGGCTGCGTTATAGCTTGGAATTCTAACAATCTTGATTCTAAGGTTGGTGGTAA
TTATAATTACCTGTATAGATTGTTTAGGAAGTCTAATCTCAAACCTTTTGAGAGAGATATTTCAACTGAAATCTATCAGGCCGG
TAGCACACCTTGTAATGGTGTTGAAGGTTTTAATTGTTACTTTCCTTTACAATCATATGGTTTCCAACCCACTAATGGTGTTGG
TTACCAACCATACAGAGTAGTAGTACTTTCTTTTGAACTTCTACATGCACCAGCAACTGTTTGTGGACCTAAAAAGTCTACTAA
TTTGGTTAAAAACAAATGTGTCAATTTCAACTTCAATGGTTTAACAGGCACAGGTGTTCTTACTGAGTCTAACAAAAAGTTTCT
GCCTTTCCAACAATTTGGCAGAGACATTGCTGACACTACTGATGCTGTCCGTGATCCACAGACACTTGAGATTCTTGACATTAC
ACCATGTTCTTTTGGTGGTGTCAGTGTTATAACACCAGGAACAAATACTTCTAACCAGGTTGCTGTTCTTTATCAGGATGTTAA
CTGCACAGAAGTCCCTGTTGCTATTCATGCAGATCAACTTACTCCTACTTGGCGTGTTTATTCTACAGGTTCTAATGTTTTTCA
AACACGTGCAGGCTGTTTAATAGGGGCTGAACATGTCAACAACTCATATGAGTGTGACATACCCATTGGTGCAGGTATATGCGC
TAGTTATCAGACTCAGACTAATTCTCCTCGGCGGGCACGTAGTGTAGCTAGTCAATCCATCATTGCCTACACTATGTCACTTGG
TGCAGAAAATTCAGTTGCTTACTCTAATAACTCTATTGCCATACCCACAAATTTTACTATTAGTGTTACCACAGAAATTCTACC
AGTGTCTATGACCAAGACATCAGTAGATTGTACAATGTACATTTGTGGTGATTCAACTGAATGCAGCAATCTTTTGTTGCAATA
TGGCAGTTTTTGTACACAATTAAACCGTGCTTTAACTGGAATAGCTGTTGAACAAGACAAAAACACCCAAGAAGTTTTTGCACA
AGTCAAACAAATTTACAAAACACCACCAATTAAAGATTTTGGTGGTTTTAATTTTTCACAAATATTACCAGATCCATCAAAACC
AAGCAAGAGGTCATTTATTGAAGATCTACTTTTCAACAAAGTGACACTTGCAGATGCTGGCTTCATCAAACAATATGGTGATTG
CCTTGGTGATATTGCTGCTAGAGACCTCATTTGTGCACAAAAGTTTAACGGCCTTACTGTTTTGCCACCTTTGCTCACAGATGA
AATGATTGCTCAATACACTTCTGCACTGTTAGCGGGTACAATCACTTCTGGTTGGACCTTTGGTGCAGGTGCTGCATTACAAAT
ACCATTTGCTATGCAAATGGCTTATAGGTTTAATGGTATTGGAGTTACACAGAATGTTCTCTATGAGAACCAAAAATTGATTGC
CAACCAATTTAATAGTGCTATTGGCAAAATTCAAGACTCACTTTCTTCCACAGCAAGTGCACTTGGAAAACTTCAAGATGTGGT
CAACCAAAATGCACAAGCTTTAAACACGCTTGTTAAACAACTTAGCTCCAATTTTGGTGCAATTTCAAGTGTTTTAAATGATAT
```

```
CCTTTCACGTCTTGACAAAGTTGAGGCTGAAGTGCAAATTGATAGGTTGATCACAGGCAGACTTCAAAGTTTGCAGACATATGT
GACTCAACAATTAATTAGAGCTGCAGAAATCAGAGCTTCTGCTAATCTTGCTGCTACTAAAATGTCAGAGTGTGTACTTGGACA
ATCAAAAAGAGTTGATTTTTGTGGAAAGGGCTATCATCTTATGTCCTTCCCTCAGTCAGCACCTCATGGTGTAGTCTTCTTGCA
TGTGACTTATGTCCCTGCACAAGAAAAGAACTTCACAACTGCTCCTGCCATTTGTCATGATGGAAAAGCACACTTTCCTCGTGA
AGGTGTCTTTGTTTCAAATGGCACACACTGGTTTGTAACACAAAGGAATTTTTATGAACCACAAATCATTACTACAGACAACAC
ATTTGTGTCTGGTAACTGTGATGTTGTAATAGGAATTGTCAACAACACAGTTTATGATCCTTTGCAACCTGAATTAGACTCATT
CAAGGAGGAGTTAGATAAATATTTTAAGAATCATACATCACCAGATGTTGATTTAGGTGACATCTCTGGCATTAATGCTTCAGT
TGTAAACATTCAAAAAGAAATTGACCGCCTCAATGAGGTTGCCAAGAATTTAAATGAATCTCTCATCGATCTCCAAGAACTTGG
AAAGTATGAGCAGTATATAAAATGGCCATGGTACATTTGGCTAGGTTTTATAGCTGGCTTGATTGCCATAGTAATGGTGACAAT
TATGCTTTGCTGTATGACCAGTTGCTGTAGTTGTCTCAAGGGCTGTTGTTCTTGTGGATCCTGCTGCAAATTTGATGAAGACGA
CTCTGAGCCAGTGCTCAAAGGAGTCAAATTACATTACACATAA'
```

```
disp(spike_DNA)
```

```
ATGTTTGTTTTTCTTGTTTTATTGCCACTAGTCTCTAGTCAGTGTGTTAATCTTACAACCAGAACTCAATTACCCCCTGCATAC
ACTAATTCTTTCACACGTGGTGTTTATTACCCTGACAAAGTTTTCAGATCCTCAGTTTTACATTCAACTCAGGACTTGTTCTTA
CCTTTCTTTTCCAATGTTACTTGGTTCCATGCTATACATGTCTCTGGGACCAATGGTACTAAGAGGTTTGATAACCCTGTCCTA
CCATTTAATGATGGTGTTTATTTTGCTTCCACTGAGAAGTCTAACATAATAAGAGGCTGGATTTTTGGTACTACTTTAGATTCG
AAGACCCAGTCCCTACTTATTGTTAATAACGCTACTAATGTTGTTATTAAAGTCTGTGAATTTCAATTTTGTAATGATCCATTT
TTGGGTGTTTATTACCACAAAAACAACAAAAGTTGGATGGAAAGTGAGTTCAGAGTTTATTCTAGTGCGAATAATTGCACTTTT
GAATATGTCTCTCAGCCTTTTCTTATGGACCTTGAAGGAAAACAGGGTAATTTCAAAAATCTTAGGGAATTTGTGTTTAAGAAT
ATTGATGGTTATTTTAAAATATATTCTAAGCACACGCCTATTAATTTAGTGCGTGATCTCCCTCAGGGTTTTTCGGCTTTAGAA
CCATTGGTAGATTTGCCAATAGGTATTAACATCACTAGGTTTCAAACTTTACTTGCTTTACATAGAAGTTATTTGACTCCTGGT
GATTCTTCTTCAGGTTGGACAGCTGGTGCTGCAGCTTATTATGTGGGTTATCTTCAACCTAGGACTTTTCTATTAAAATATAAT
GAAAATGGAACCATTACAGATGCTGTAGACTGTGCACTTGACCCTCTCTCAGAAACAAAGTGTACGTTGAAATCCTTCACTGTA
GAAAAAGGAATCTATCAAACTTCTAACTTTAGAGTCCAACCAACAGAATCTATTGTTAGATTTCCTAATATTACAAACTTGTGC
CCTTTTGGTGAAGTTTTTAACGCCACCAGATTTGCATCTGTTTATGCTTGGAACAGGAAGAGAATCAGCAACTGTGTTGCTGAT
TATTCTGTCCTATATAATTCCGCATCATTTTCCACTTTTAAGTGTTATGGAGTGTCTCCTACTAAATTAAATGATCTCTGCTTT
ACTAATGTCTATGCAGATTCATTTGTAATTAGAGGTGATGAAGTCAGACAAATCGCTCCAGGGCAAACTGGAAAGATTGCTGAT
TATAATTATAAATTACCAGATGATTTTACAGGCTGCGTTATAGCTTGGAATTCTAACAATCTTGATTCTAAGGTTGGTGGTAAT
TATAATTACCTGTATAGATTGTTTAGGAAGTCTAATCTCAAACCTTTTGAGAGAGATATTTCAACTGAAATCTATCAGGCCGGT
AGCACACCTTGTAATGGTGTTGAAGGTTTTAATTGTTACTTTCCTTTACAATCATATGGTTTCCAACCCACTAATGGTGTTGGT
TACCAACCATACAGAGTAGTAGTACTTTCTTTTGAACTTCTACATGCACCAGCAACTGTTTGTGGACCTAAAAAGTCTACTAAT
TTGGTTAAAAACAAATGTGTCAATTTCAACTTCAATGGTTTAACAGGCACAGGTGTTCTTACTGAGTCTAACAAAAAGTTTCTG
CCTTTCCAACAATTTGGCAGAGACATTGCTGACACTACTGATGCTGTCCGTGATCCACAGACACTTGAGATTCTTGACATTACA
CCATGTTCTTTTGGTGGTGTCAGTGTTATAACACCAGGAACAAATACTTCTAACCAGGTTGCTGTTCTTTATCAGGATGTTAAC
TGCACAGAAGTCCCTGTTGCTATTCATGCAGATCAACTTACTCCTACTTGGCGTGTTTATTCTACAGGTTCTAATGTTTTTCAA
ACACGTGCAGGCTGTTTAATAGGGGCTGAACATGTCAACAACTCATATGAGTGTGACATACCCATTGGTGCAGGTATATGCGCT
AGTTATCAGACTCAGACTAATTCTCCTCGGCGGGCACGTAGTGTAGCTAGTCAATCCATCATTGCCTACACTATGTCACTTGGT
GCAGAAAATTCAGTTGCTTACTCTAATAACTCTATTGCCATACCCACAAATTTTACTATTAGTGTTACCACAGAAATTCTACCA
GTGTCTATGACCAAGACATCAGTAGATTGTACAATGTACATTTGTGGTGATTCAACTGAATGCAGCAATCTTTTGTTGCAATAT
GGCAGTTTTTGTACACAATTAAACCGTGCTTTAACTGGAATAGCTGTTGAACAAGACAAAAACACCCAAGAAGTTTTTGCACAA
GTCAAACAAATTTACAAAACACCACCAATTAAAGATTTTGGTGGTTTTAATTTTTCACAAATATTACCAGATCCATCAAAACCA
AGCAAGAGGTCATTTATTGAAGATCTACTTTTCAACAAAGTGACACTTGCAGATGCTGGCTTCATCAAACAATATGGTGATTGC
CTTGGTGATATTGCTGCTAGAGACCTCATTTGTGCACAAAAGTTTAACGGCCTTACTGTTTTGCCACCTTTGCTCACAGATGAA
ATGATTGCTCAATACACTTCTGCACTGTTAGCGGGTACAATCACTTCTGGTTGGACCTTTGGTGCAGGTGCTGCATTACAAATA
CCATTTGCTATGCAAATGGCTTATAGGTTTAATGGTATTGGAGTTACACAGAATGTTCTCTATGAGAACCAAAAATTGATTGCC
AACCAATTTAATAGTGCTATTGGCAAAATTCAAGACTCACTTTCTTCCACAGCAAGTGCACTTGGAAAACTTCAAGATGTGGTC
AACCAAAATGCACAAGCTTTAAACACGCTTGTTAAACAACTTAGCTCCAATTTTGGTGCAATTTCAAGTGTTTTAAATGATATC
CTTTCACGTCTTGACAAAGTTGAGGCTGAAGTGCAAATTGATAGGTTGATCACAGGCAGACTTCAAAGTTTGCAGACATATGTG
ACTCAACAATTAATTAGAGCTGCAGAAATCAGAGCTTCTGCTAATCTTGCTGCTACTAAAATGTCAGAGTGTGTACTTGGACAA
TCAAAAAGAGTTGATTTTTGTGGAAAGGGCTATCATCTTATGTCCTTCCCTCAGTCAGCACCTCATGGTGTAGTCTTCTTGCAT
GTGACTTATGTCCCTGCACAAGAAAAGAACTTCACAACTGCTCCTGCCATTTGTCATGATGGAAAAGCACACTTTCCTCGTGAA
GGTGTCTTTGTTTCAAATGGCACACACTGGTTTGTAACACAAAGGAATTTTTATGAACCACAAATCATTACTACAGACAACACA
TTTGTGTCTGGTAACTGTGATGTTGTAATAGGAATTGTCAACAACACAGTTTATGATCCTTTGCAACCTGAATTAGACTCATTC
AAGGAGGAGTTAGATAAATATTTTAAGAATCATACATCACCAGATGTTGATTTAGGTGACATCTCTGGCATTAATGCTTCAGTT
GTAAACATTCAAAAAGAAATTGACCGCCTCAATGAGGTTGCCAAGAATTTAAATGAATCTCTCATCGATCTCCAAGAACTTGGA
AAGTATGAGCAGTATATAAAATGGCCATGGTACATTTGGCTAGGTTTTATAGCTGGCTTGATTGCCATAGTAATGGTGACAATT
ATGCTTTGCTGTATGACCAGTTGCTGTAGTTGTCTCAAGGGCTGTTGTTCTTGTGGATCCTGCTGCAAATTTGATGAAGACGAC
TCTGAGCCAGTGCTCAAAGGAGTCAAATTACATTACACATAA
```

Question 4: what is the mRNA transcript of the spike protein?

```
% First, initialize the RNA sequence from the DNA sequence
spike_RNA = spike_DNA;

% The difference between DNA and RNA is that RNA contains uracil bases int
% he place of thymine
spike_RNA(spike_DNA == 'T') = 'U';
length(spike_RNA)
```

```
ans = 3822
```

```
disp(spike_RNA)
```

```
AUGUUUGUUUUUCUUGUUUUAUUGCCACUAGUCUCUAGUCAGUGUGUUAAUCUUACAACCAGAACUCAAUUACCCCCUGCAUAC
ACUAAUUCUUUCACACGUGGUGUUUAUUACCCUGACAAAGUUUUCAGAUCCUCAGUUUUACAUUCAACUCAGGACUUGUUCUUA
CCUUUCUUUUCCAAUGUUACUUGGUUCCAUGCUAUACAUGUCUCUGGGACCAAUGGUACUAAGAGGUUUGAUAACCCUGUCCUA
CCAUUUAAUGAUGGUGUUUAUUUUGCUUCCACUGAGAAGUCUAACAUAAUAAGAGGCUGGAUUUUUGGUACUACUUUAGAUUCG
AAGACCCAGUCCCUACUUAUUGUUAAUAACGCUACUAAUGUUGUUAUUAAAGUCUGUGAAUUUCAAUUUUGUAAUGAUCCAUUU
UUGGGUGUUUAUUACCACAAAAACAACAAAAGUUGGAUGGAAAGUGAGUUCAGAGUUUAUUCUAGUGCGAAUAAUUGCACUUUU
GAAUAUGUCUCUCAGCCUUUUCUUAUGGACCUUGAAGGAAAACAGGGUAAUUUCAAAAAUCUUAGGGAAUUUGUGUUUAAGAAU
AUUGAUGGUUAUUUUAAAAUAUAUUCUAAGCACACGCCUAUUAAUUUAGUGCGUGAUCUCCCUCAGGGUUUUUCGGCUUUAGAA
CCAUUGGUAGAUUUGCCAAUAGGUAUUAACAUCACUAGGUUUCAAACUUUACUUGCUUUACAUAGAAGUUAUUUGACUCCUGGU
GAUUCUUCUUCAGGUUGGACAGCUGGUGCUGCAGCUUAUUAUGUGGGUUAUCUUCAACCUAGGACUUUUCUAUUAAAAUAUAAU
GAAAAUGGAACCAUUACAGAUGCUGUAGACUGUGCACUUGACCCUCUCUCAGAAACAAAGUGUACGUUGAAAUCCUUCACUGUA
GAAAAAGGAAUCUAUCAAACUUCUAACUUUAGAGUCCAACCAACAGAAUCUAUUGUUAGAUUUCCUAAUAUUACAAACUUGUGC
CCUUUUGGUGAAGUUUUUAACGCCACCAGAUUUGCAUCUGUUUAUGCUUGGAACAGGAAGAGAAUCAGCAACUGUGUUGCUGAU
UAUUCUGUCCUAUAUAAUUCCGCAUCAUUUUCCACUUUUAAGUGUUAUGGAGUGUCUCCUACUAAAUUAAAUGAUCUCUGCUUU
ACUAAUGUCUAUGCAGAUUCAUUUGUAAUUAGAGGUGAUGAAGUCAGACAAAUCGCUCCAGGGCAAACUGGAAAGAUUGCUGAU
UAUAAUUAUAAAUUACCAGAUGAUUUUACAGGCUGCGUUAUAGCUUGGAAUUCUAACAAUCUUGAUUCUAAGGUUGGUGGUAAU
UAUAAUUACCUGUAUAGAUUGUUUAGGAAGUCUAAUCUCAAACCUUUUGAGAGAGAUAUUUCAACUGAAAUCUAUCAGGCCGGU
AGCACACCUUGUAAUGGUGUUGAAGGUUUUAAUUGUUACUUUCCUUUACAAUCAUAUGGUUUCCAACCCACUAAUGGUGUUGGU
UACCAACCAUACAGAGUAGUAGUACUUUCUUUUGAACUUCUACAUGCACCAGCAACUGUUUGUGGACCUAAAAAGUCUACUAAU
UUGGUUAAAAACAAAUGUGUCAAUUUCAACUUCAAUGGUUUAACAGGCACAGGUGUUCUUACUGAGUCUAACAAAAAGUUUCUG
CCUUUCCAACAAUUUGGCAGAGACAUUGCUGACACUACUGAUGCUGUCCGUGAUCCACAGACACUUGAGAUUCUUGACAUUACA
CCAUGUUCUUUUGGUGGUGUCAGUGUUAUAACACCAGGAACAAAUACUUCUAACCAGGUUGCUGUUCUUUAUCAGGAUGUUAAC
UGCACAGAAGUCCCUGUUGCUAUUCAUGCAGAUCAACUUACUCCUACUUGGCGUGUUUAUUCUACAGGUUCUAAUGUUUUUCAA
ACACGUGCAGGCUGUUUAAUAGGGGCUGAACAUGUCAACAACUCAUAUGAGUGUGACAUACCCAUUGGUGCAGGUAUAUGCGCU
AGUUAUCAGACUCAGACUAAUUCUCCUCGGCGGGCACGUAGUGUAGCUAGUCAAUCCAUCAUUGCCUACACUAUGUCACUUGGU
GCAGAAAAUUCAGUUGCUUACUCUAAUAACUCUAUUGCCAUACCCACAAAUUUUACUAUUAGUGUUACCACAGAAAUUCUACCA
GUGUCUAUGACCAAGACAUCAGUAGAUUGUACAAUGUACAUUUGUGGUGAUUCAACUGAAUGCAGCAAUCUUUUGUUGCAAUAU
GGCAGUUUUUGUACACAAUUAAACCGUGCUUUAACUGGAAUAGCUGUUGAACAAGACAAAAACACCCAAGAAGUUUUUGCACAA
GUCAAACAAAUUUACAAAACACCACCAAUUAAAGAUUUUGGUGGUUUUAAUUUUUCACAAAUAUUACCAGAUCCAUCAAAACCA
AGCAAGAGGUCAUUUAUUGAAGAUCUACUUUUCAACAAAGUGACACUUGCAGAUGCUGGCUUCAUCAAACAAUAUGGUGAUUGC
CUUGGUGAUAUUGCUGCUAGAGACCUCAUUUGUGCACAAAAGUUUAACGGCCUUACUGUUUUGCCACCUUUGCUCACAGAUGAA
AUGAUUGCUCAAUACACUUCUGCACUGUUAGCGGGUACAAUCACUUCUGGUUGGACCUUUGGUGCAGGUGCUGCAUUACAAAUA
CCAUUUGCUAUGCAAAUGGCUUAUAGGUUUAAUGGUAUUGGAGUUACACAGAAUGUUCUCUAUGAGAACCAAAAAUUGAUUGCC
AACCAAUUUAAUAGUGCUAUUGGCAAAAUUCAAGACUCACUUUCUUCCACAGCAAGUGCACUUGGAAAACUUCAAGAUGUGGUC
AACCAAAAUGCACAAGCUUUAAACACGCUUGUUAAACAACUUAGCUCCAAUUUUGGUGCAAUUUCAAGUGUUUUAAAUGAUAUC
CUUUCACGUCUUGACAAAGUUGAGGCUGAAGUGCAAAUUGAUAGGUUGAUCACAGGCAGACUUCAAAGUUUGCAGACAUAUGUG
ACUCAACAAUUAAUUAGAGCUGCAGAAAUCAGAGCUUCUGCUAAUCUUGCUGCUACUAAAAUGUCAGAGUGUGUACUUGGACAA
UCAAAAAGAGUUGAUUUUUGUGGAAAGGGCUAUCAUCUUAUGUCCUUCCCUCAGUCAGCACCUCAUGGUGUAGUCUUCUUGCAU
GUGACUUAUGUCCCUGCACAAGAAAAGAACUUCACAACUGCUCCUGCCAUUUGUCAUGAUGGAAAAGCACACUUUCCUCGUGAA
GGUGUCUUUGUUUCAAAUGGCACACACUGGUUUGUAACACAAAGGAAUUUUUAUGAACCACAAAUCAUUACUACAGACAACACA
UUUGUGUCUGGUAACUGUGAUGUUGUAAUAGGAAUUGUCAACAACACAGUUUAUGAUCCUUUGCAACCUGAAUUAGACUCAUUC
AAGGAGGAGUUAGAUAAAUAUUUUAAGAAUCAUACAUCACCAGAUGUUGAUUUAGGUGACAUCUCUGGCAUUAAUGCUUCAGUU
GUAAACAUUCAAAAAGAAAUUGACCGCCUCAAUGAGGUUGCCAAGAAUUUAAAUGAAUCUCUCAUCGAUCUCCAAGAACUUGGA
AAGUAUGAGCAGUAUAUAAAAUGGCCAUGGUACAUUUGGCUAGGUUUUAUAGCUGGCUUGAUUGCCAUAGUAAUGGUGACAAUU
```

```
AUGCUUUGCUGUAUGACCAGUUGCUGUAGUUGUCUCAAGGGCUGUUGUUCUUGUGGAUCCUGCUGCAAAUUUGAUGAAGACGAC
UCUGAGCCAGUGCUCAAAGGAGUCAAAUUACAUUACACAUAA
```

Question 5: What is the amino acid sequence of the spike protein? That is, what is the product of protein *translation*.

```
% Step 1: Preallocate character array to contain protein sequence. Note:
% stop codon is not included in the aminoa cid sequence
spike_protein = blanks(length(spike_DNA)/3-1);


% Step 2: compare each three bases to list of codons to determine
% appropriate amino acod.
% String array: each line corresponds to a codon. To be used for
% identifying codon
codon_bases = string(codons(:,1));

% Note: we do not consider the Stop Codon
for k = 1:length(spike_protein)-3
    % Based on based pairs in codon, identify the index of the codon we are
    % interested in
    codon_index = codon_bases == string(spike_DNA((1:3)+(k-1)*3));

    % Retrive amino acid corresponding to codon and store in protein
    % sequence
    spike_Protein(k) = codons{codon_index,3};
end
length(spike_Protein)
```

```
ans = 1270
```

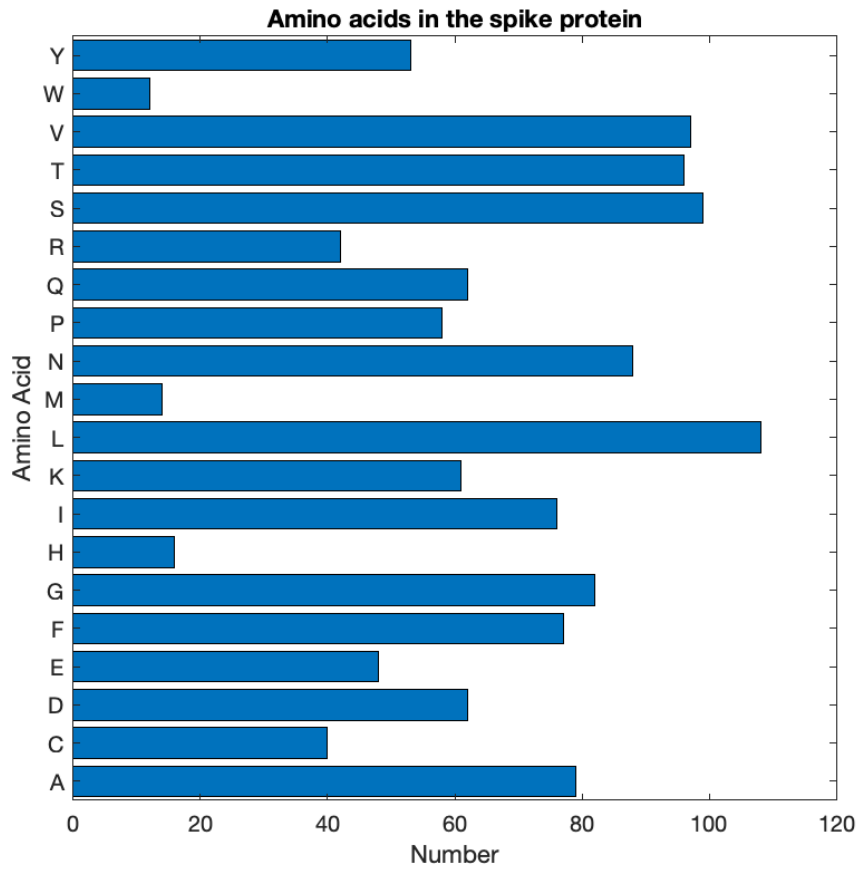
```
disp(spike_Protein)
```

```
MFVFLVLLPLVSSQCVNLTTRTQLPPAYTNSFTRGVYYPDKVFRSSVLHSTQDLFLPFFSNVTWFHAIHVSGTNGTKRFDNPVL
PFNDGVYFASTEKSNIIRGWIFGTTLDSKTQSLLIVNNATNVVIKVCEFQFCNDPFLGVYYHKNNKSWMESEFRVYSSANNCTF
EYVSQPFLMDLEGKQGNFKNLREFVFKNIDGYFKIYSKHTPINLVRDLPQGFSALEPLVDLPIGINITRFQTLLALHRSYLTPG
DSSSGWTAGAAAYYVGYLQPRTFLLKYNENGTITDAVDCALDPLSETKCTLKSFTVEKGIYQTSNFRVQPTESIVRFPNITNLC
PFGEVFNATRFASVYAWNRKRISNCVADYSVLYNSASFSTFKCYGVSPTKLNDLCFTNVYADSFVIRGDEVRQIAPGQTGKIAD
YNYKLPDDFTGCVIAWNSNNLDSKVGGNYNYLYRLFRKSNLKPFERDISTEIYQAGSTPCNGVEGFNCYFPLQSYGFQPTNGVG
YQPYRVVVLSFELLHAPATVCGPKKSTNLVKNKCVNFNFNGLTGTGVLTESNKKFLPFQQFGRDIADTTDAVRDPQTLEILDIT
PCSFGGVSVITPGTNTSNQVAVLYQDVNCTEVPVAIHADQLTPTWRVYSTGSNVFQTRAGCLIGAEHVNNSYECDIPIGAGICA
SYQTQTNSPRRARSVASQSIIAYTMSLGAENSVAYSNNSIAIPTNFTISVTTEILPVSMTKTSVDCTMYICGDSTECSNLLLQY
GSFCTQLNRALTGIAVEQDKNTQEVFAQVKQIYKTPPIKDFGGFNFSQILPDPSKPSKRSFIEDLLFNKVTLADAGFIKQYGDC
LGDIAARDLICAQKFNGLTVLPPLLTDEMIAQYTSALLAGTITSGWTFGAGAALQIPFAMQMAYRFNGIGVTQNVLYENQKLIA
NQFNSAIGKIQDSLSSTASALGKLQDVVNQNAQALNTLVKQLSSNFGAISSVLNDILSRLDKVEAEVQIDRLITGRLQSLQTYV
TQQLIRAAEIRASANLAATKMSECVLGQSKRVDFCGKGYHLMSFPQSAPHGVVFLHVTYVPAQEKNFTTAPAICHDGKAHFPRE
GVFVSNGTHWFVTQRNFYEPQIITTDNTFVSGNCDVVIGIVNNTVYDPLQPELDSFKEELDKYFKNHTSPDVDLGDISGINASV
VNIQKEIDRLNEVAKNLNESLIDLQELGKYEQYIKWPWYIWLGFIAGLIAIVMVTIMLCCMTSCCSCLKGCCSCGSCCKFDEDD
SEPVLKGVKL
```

Question 6: How many of each amino acid are in the spike protein?

```
% To solve this: create a cell array containing "unique" amino acids and a
% corresponding vector that specifies if that amino acid has been so far

% Create cell array to store unique amino acids
AA_list = {spike_Protein(1)};

%Create a vector to store the corresponding counts
AA_count = 1;

% Loop through each amino acid of protien
for k = 2:length(spike_Protein)
    % Create a logical variable
    % True: amino acid has been observed before in protein sequence
    %False: no match -> new amino acid
    match = false;

    % For each identified "unique" amino acid
    for m = 1:length(AA_list)
        % If the amino acids match, increment the number of times it
        % appears
        if strcmp(AA_list{m}, spike_Protein(k))
            AA_count(m) = AA_count(m)+1;
            match = true;
        end
    end

    % If the current word did not match, then we have found a new amino
    % acid!
    if ~match
        AA_list{end+1} = spike_Protein(k);
        AA_count(end+1) = 1;
    end
end

% Step 2: Plot the distribution
% Convert cell array to categorical
xval_Q6 = categorical(AA_list);

% Make bar graph
h1 = bar(xval_Q6, AA_count);
```

```
title('Amino acids in the spike protein')
axis square
ylabel('Number');
xlabel('Amino Acid')
set(h1, 'horizontal', 'on')
```



Writing text data to a file

- Simple: Write a **cell array** to a **delimitted file** using `writecell`. Any delimiter may be used but comma-delimitted is the default.

```
% Write codons to a text file
writecell(codons,'myCodonFile.txt');
```

- More involved: use built-in MATLAB **file manipulation commands**: `fopen`, `fprintf`, `fclose`

```
% Create and open a new file with specified name
fileID = fopen('S_ProteinSequence.txt','w');
% Write character array to file
fprintf(fileID,spike_Protein);
% Close file so that it can be read elsewhere
fclose(fileID);
```

Structure arrays

Structure arrays are useful for storing data in a **tree-like structure** and are common when importing data into MATLAB. A *structure array* is a data type that groups related data using data containers called *fields*. Each field can contain any type of data.

You have worked with structure arrays (and analogous entities) already:

- In **lecture 14**, we recorded my position as we walked through the halls of Towne building and, afterwards, imported that data into MATLAB. The imported data was a structure array with one field, `Position`:

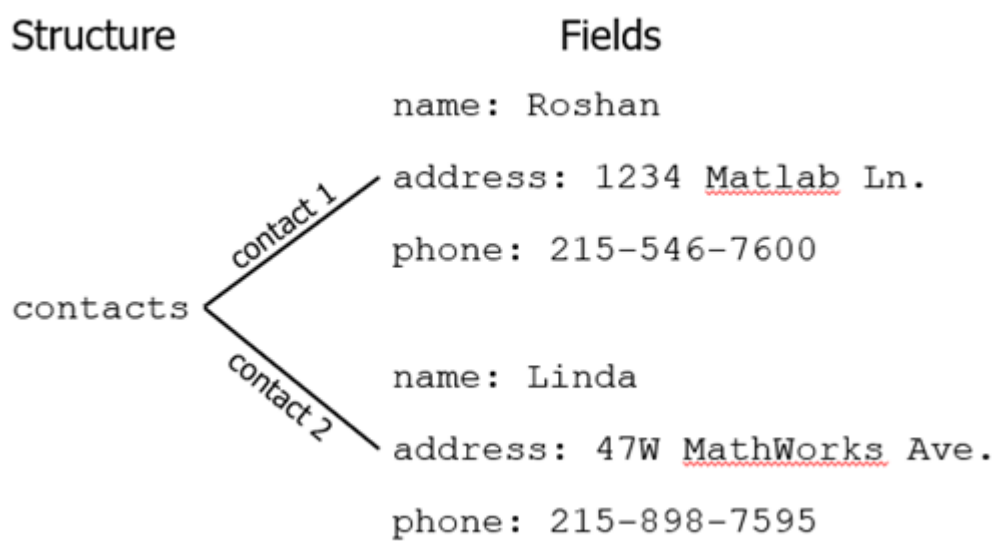
```
% Load the data we just collected
rawData = load('sensorData5.mat')
```

```
rawData = struct with fields:
    Position: [42×6 timetable]
```

- We have done a lot of formatting of plots and graphics. While these are graphics objects, there are parallels because each is a **base object** with **parameters** or **fields** associated with it.

```
>> h1

h1 =

  Bar with properties:

    BarLayout: 'grouped'
     BarWidth: 0.8000
    FaceColor: [0 0.4470 0.7410]
    EdgeColor: [0 0 0]
    BaseValue: 0
        XData: [1×20 categorical]
        YData: [1×20 double]
```

Tree analogy: the base object (known as the *structure*) is the trunk, the *fields/parameters* are the branches. Values in a field are associated with the "higher order" structure.



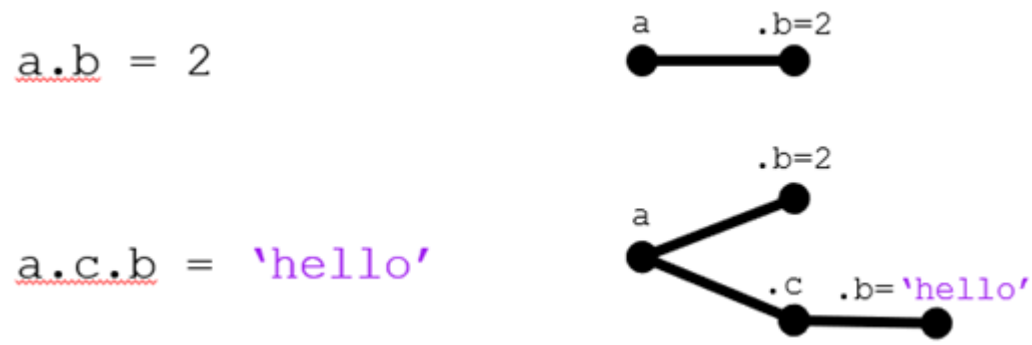
Creating a structure array

Fields in a structure array can be created or accessed using dot notation (`.`).

```
% Create two fields for a structure array, a:
% a.b = 2;
% a.c.b = 'hello';
% a
```

The above structure array can be represented visually below. The fields `b` and `c` "belong" to structure array `a`.

Note: `a.c` is a structure array itself!

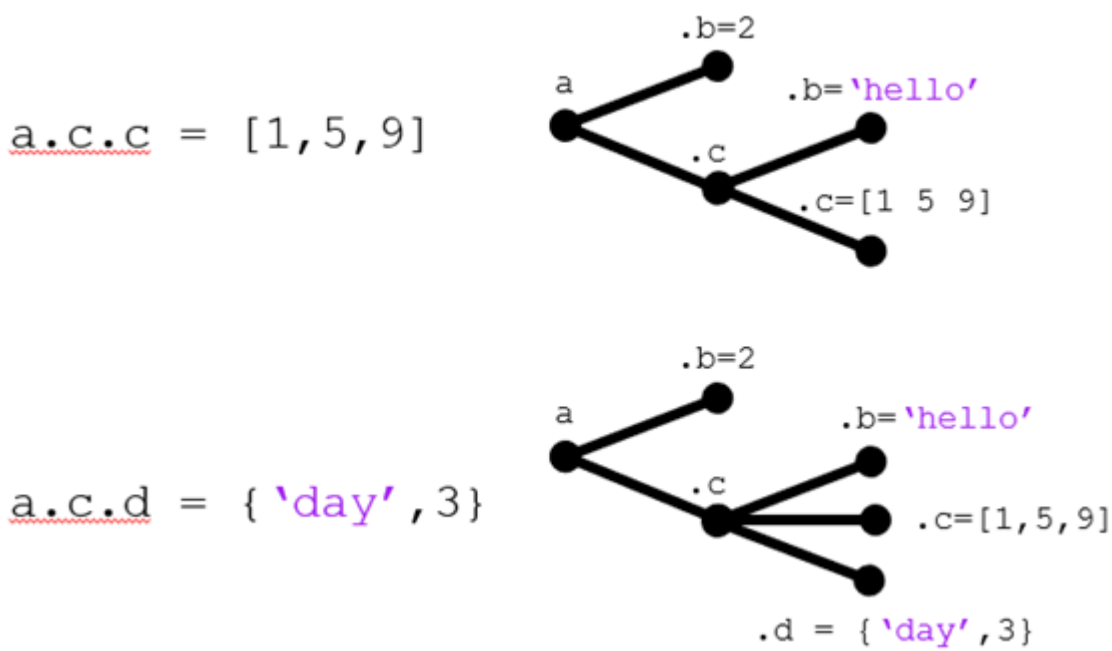



```
% Access field values
% a.b
% a.c
% a.c.b
```

Any data type can be added to a structure array. In the following, we add a numerical vector, a cell array, and a character array.

```
% a.c.c = [1,5,9];
% a.c.d = {'day',3};
```

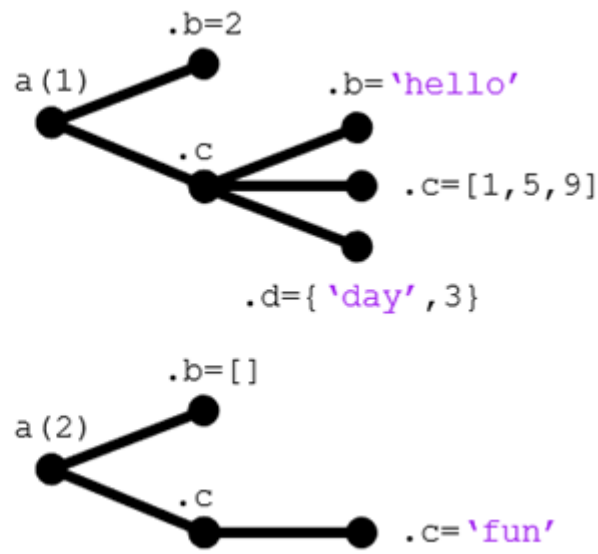
Visually:



Furthermore, structure arrays can have a dimension greater than 1x1:

```
% a(2).c.c = 'fun';
```

```
a(2).c.c = 'fun'
```

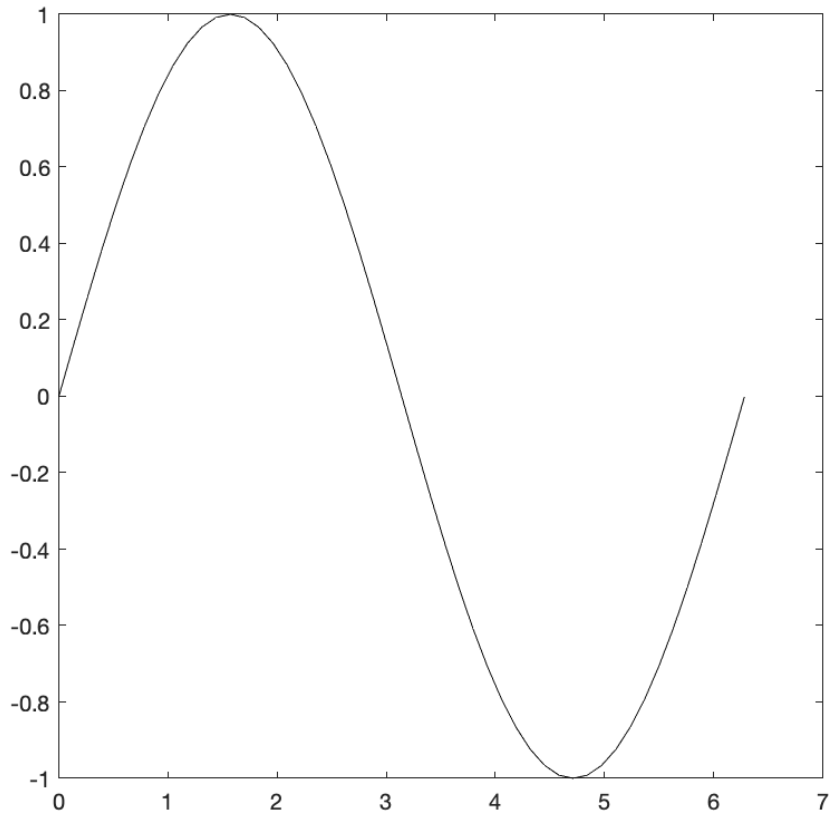


<u>Takeaway</u>: structure arrays are a useful way to organize data. A structure array and all fields within may be passed to a function or command by simply referring to the structure.

Relation to graphics objects

So far, we have used `get()` to visualize the parameters for a graphics object (e.g. plot) and `set()` to change the parameters. You can also use **dot notation to view or modify a parameter value**.

```
close all
x = 0:pi/24:2*pi;
y = sin(x);
h = plot(x,y,'k-');
axis square
```



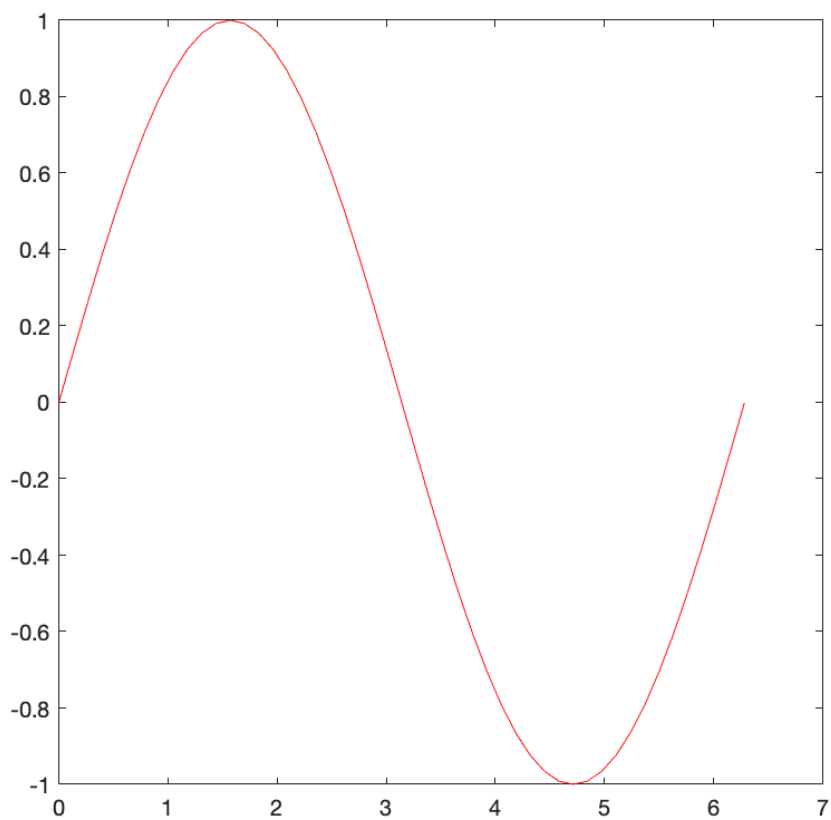

```
get(h)
```

```
    AlignVertexCenters: off
            Annotation: [1×1 matlab.graphics.eventdata.Annotation]
          BeingDeleted: off
            BusyAction: 'queue'
         ButtonDownFcn: ''
              Children: [0×0 GraphicsPlaceholder]
              Clipping: on
                 Color: [0 0 0]
             ColorMode: 'manual'
           ContextMenu: [0×0 GraphicsPlaceholder]
             CreateFcn: ''
       DataTipTemplate: [1×1 matlab.graphics.datatip.DataTipTemplate]
             DeleteFcn: ''
           DisplayName: ''
      HandleVisibility: 'on'
               HitTest: on
         Interruptible: on
              LineJoin: 'round'
             LineStyle: '-'
         LineStyleMode: 'manual'
             LineWidth: 0.5000
                Marker: 'none'
       MarkerEdgeColor: 'auto'
       MarkerFaceColor: 'none'
```

```
      MarkerIndices: [1×49 uint64]
         MarkerMode: 'manual'
         MarkerSize: 6
             Parent: [1×1 Axes]
      PickableParts: 'visible'
           Selected: off
 SelectionHighlight: on
        SeriesIndex: 1
                Tag: ''
               Type: 'line'
           UserData: []
            Visible: on
              XData: [1×49 double]
          XDataMode: 'manual'
        XDataSource: ''
              YData: [1×49 double]
        YDataSource: ''
              ZData: [1×0 double]
        ZDataSource: ''
```

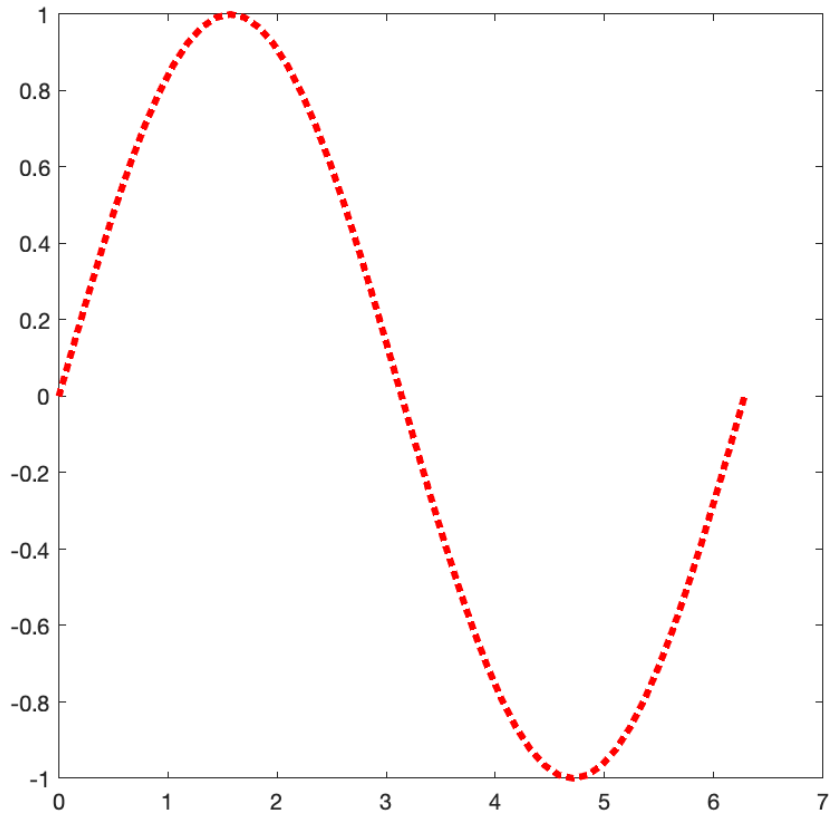
- Modification using `set()/get()`

```
% Change line color
set(h,'Color','r')
```



- Using dot notation on the structure/fields:

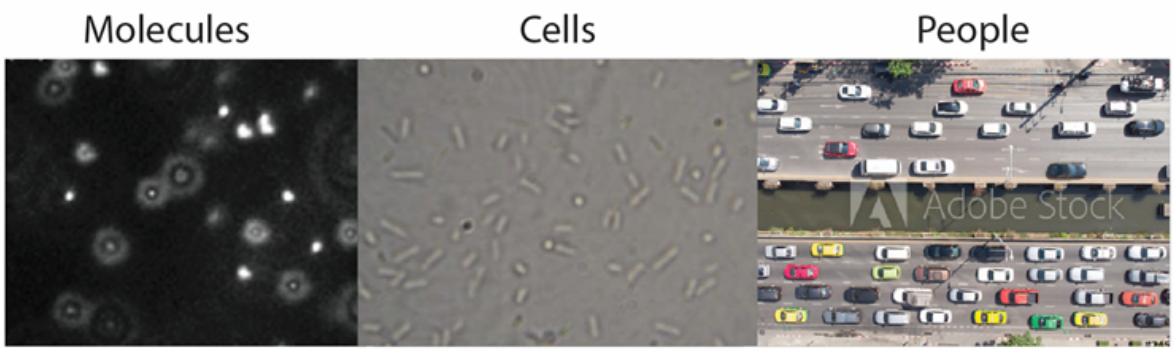
```
% Change line width
h.LineWidth = 3;
% Change line style
h.LineStyle = ':';
```



<u>Note</u>: even if you prefer dot notation, get() is still useful to identify potential parameters to change.

Example: visualizing "particle" movement

There are many cases in science and engineering where you may track the position of *some entity* over time. You could record the motion of fluorescent molecules, the run and tumble trajectories of bacteria, group dynamics of animal swarms, or human migration. In this problem: **given some recorded data, produce a visualization of the "particle" movement**.



Tips and notes:

- `importdata()`, used above, can be used to incorporate mixed data (*e.g.* text and numbers).
- `pause` will halt MATLAB command execution for the specified time, in seconds.
- `rectangle` can be used to produce a square, rectangle, oval, or circle on the current axes.

```
% Import data as a structure array
myData = importdata('L18_TrackingData.txt')
```

```
myData = struct with fields:
        data: [20×3 double]
    textdata: {21×4 cell}
```

```
% Numerical data
myData.data
```

```
ans = 20×3
    3.0000    3.0000    1.0000
    3.0000    4.0000    1.5000
    4.0000    4.0000    2.0000
    4.0000    5.0000    2.5000
    5.0000    5.0000    3.0000
    6.0000    5.0000    2.5000
    7.0000    5.0000    2.0000
    8.0000    6.0000    1.5000
    8.0000    7.0000    1.0000
    7.0000    7.0000    0.5000
```

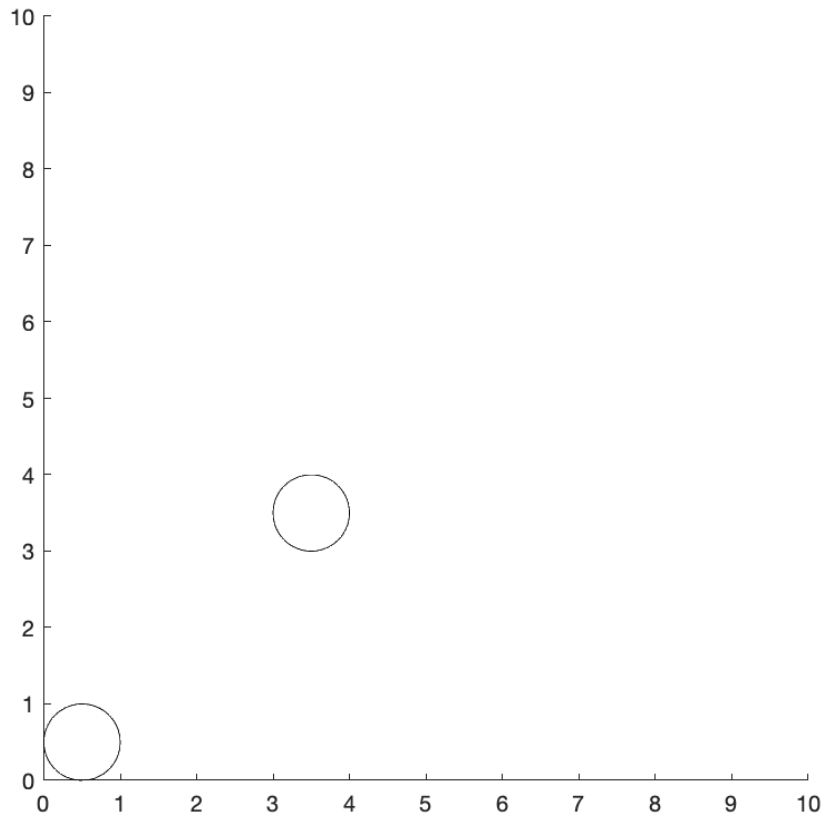
```
% Text data
myData.textdata
```

```
ans = 21×4 cell array
    {'Marker'}    {'X'     }    {'Y'     }    {'Radius'}
    {'A'     }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'     }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'     }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'     }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'     }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'     }    {0×0 char}    {0×0 char}    {0×0 char}
```

```
    {'A'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'A'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
    {'B'      }    {0×0 char}    {0×0 char}    {0×0 char}
```

<u>Step 1</u>: start simple. Write pseudocode and figure out how to plot "circles" for a given position

```
close all % Close previous plots

% Create visualization: note
h1 = rectangle('Position',[0,0,1,1],'Curvature',[1,1]);
h2 = rectangle('Position',[3,3,1,1],'Curvature',[1,1]);

% Specify plotting parameters
axis square
xlim([0,10])
ylim([0,10])
```



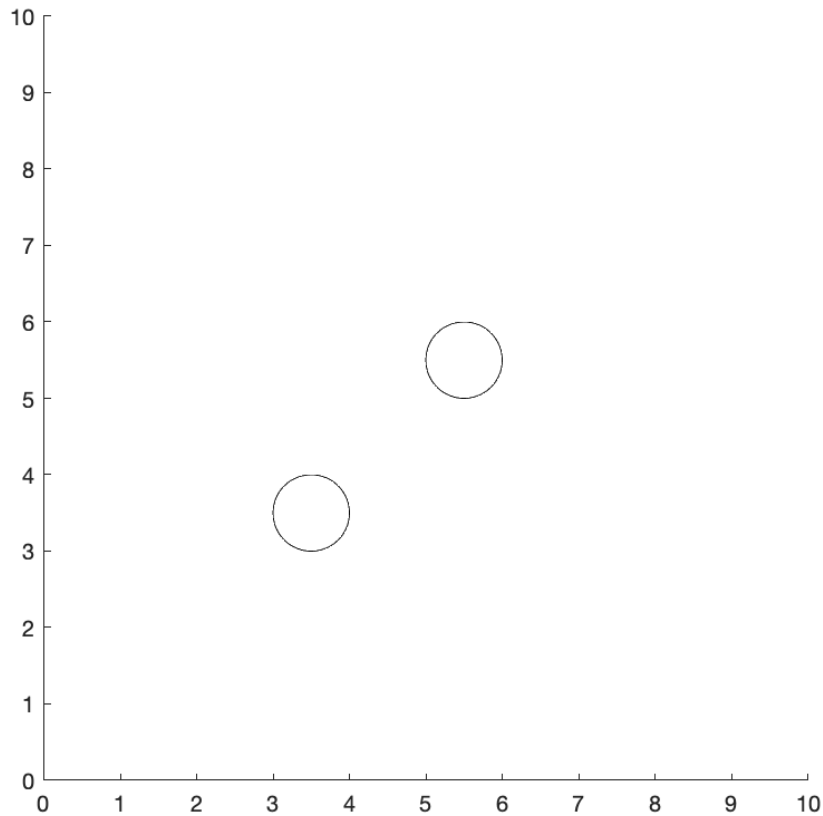
<u>Step 2</u>: Convert data into a convenient format. In this case, a structure array is good.

```
% Step 1: Create cell array of just particle identifiers
particle_ID = myData.textdata(2:end,1);

% Step 2: Create subarrays unique to each particle
A_data = myData.data(strcmp(particle_ID,'A'),:);
B_data = myData.data(strcmp(particle_ID,'B'),:);

% Step 3: Populate structure array with data for both particles
particle(1).ID = 'A';
particle(1).pos = A_data(:,1:2);
particle(1).dia = A_data(:,3);

particle(2).ID = 'B';
particle(2).pos = B_data(:,1:2);
particle(2).dia = B_data(:,3);

particle
```

```
particle = 1×2 struct array with fields:
    ID
    pos
    dia
```

<u>Step 3</u>: Refine plotting code for one time point

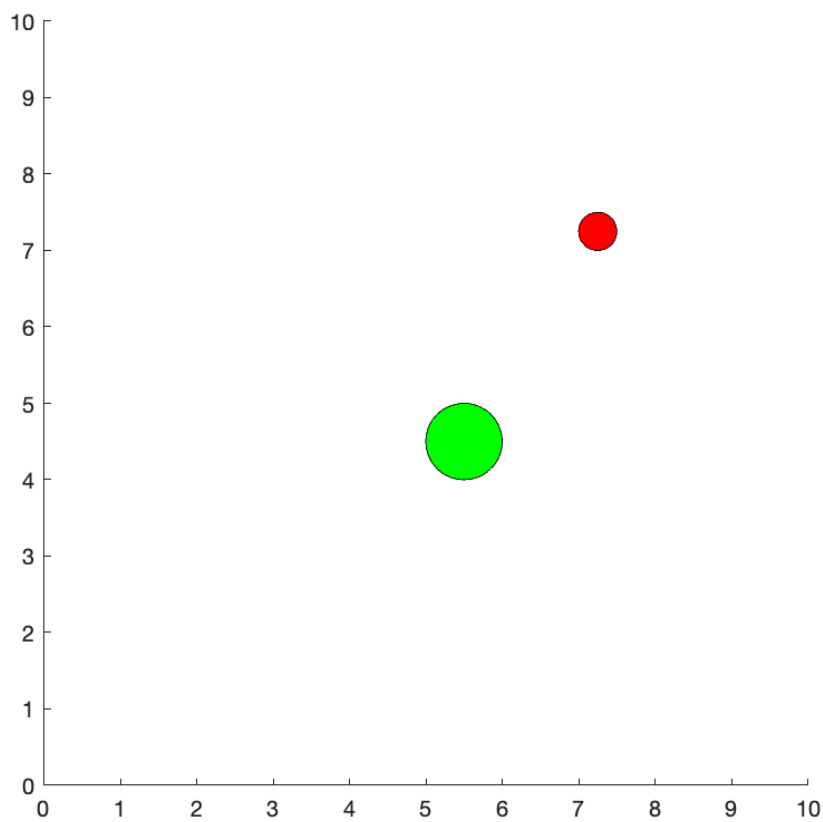
```
close all % Close previous plots

% Goal: loop through all time points, i.e. all rows int he positional
% arrays

% Create axes and specify graphical parameters
g = axes;
axis square
xlim([0,10])
ylim([0,10])

% Create visualization: note the values of each position
h(1) =
rectangle('Position',[particle(1).pos(1,:),particle(1).dia(1)*[1,1]],'Curvature'
,[1,1]);
h(2) =
rectangle('Position',[particle(2).pos(1,:),particle(2).dia(1)*[1,1]],'Curvature'
,[1,1]);
```



<u>Step 4</u>: Refine plotting code for all time point

```
close all % Close previous plots

% Goal: loop through all time points, i.e. all rows in the positional
% arrays.

% Time points
timePts = size(particle(1).pos,1);

% Create axes and specify graphical paramters
axis square
xlim([0,10])
ylim([0,10])

%Create Initial position of particles
```

```
 h(1) =
rectangle('Position',[particle(1).pos(1,:),particle(1).dia(1)*[1,1]],'Curvature'
,[1,1]);
 h(2) =
rectangle('Position',[particle(2).pos(1,:),particle(2).dia(1)*[1,1]],'Curvature'
,[1,1]);
 % Specify colors
 h(1).FaceColor = 'r';
 h(2).FaceColor = 'g';

 % Loop through all time points
 for k = 2:timePts
     % Pause for 1 second
     pause(1)

     % Update position and radius for each particle
     for m = 1:2
         h(m).Position = [particle(m).pos(k,:),particle(m).dia(k)*[1,1]];
     end
 end
```



```
disp('Done')
```

```
Done
```

TOPIC 18: Intro to audio and image processing

ENGR105, 11/3/20

Numeric data, audio, and images

In the example at the end of class on Thursday, we loaded and processed a file that contained both text and numerical information. In your courses, research, job, you will likely need to load and analyze other numerical data as well.

Remember: to open a file by specifying only its name, the file must be located in the current directory

Importing data and reading files

These are some of the most commonly used functions to load files and data into MATLAB.

Note: "comma-delimitted" file means a text file in which each entry in a give row is separated by commas. You may be able to load files with **other delimitters** by changing an input to the following functions (please see the corresponding `help` documentation).

Numerical data

- `importdata` - read numerical/text data files as a **structure array**
- `readcell` - read a comma delimitted file (numerical, text, or both) into a **cell array**
- `csvread` - read **numerical only** csv files into an **array** (I use this a lot because it is "easy")

```
% Read CSV file: Additional inputs specify "offset" to row/column to start
% reading from
data1 = csvread('L18_TrackingData.txt',1,1)
```

```
data1 = 20×3
    3.0000    3.0000    1.0000
    3.0000    4.0000    1.5000
    4.0000    4.0000    2.0000
    4.0000    5.0000    2.5000
    5.0000    5.0000    3.0000
    6.0000    5.0000    2.5000
    7.0000    5.0000    2.0000
    8.0000    6.0000    1.5000
    8.0000    7.0000    1.0000
    7.0000    7.0000    0.5000
```

Note: the offsets are necessary to ensure that MATLAB *only* reads the numerical data. The function will **error** if it tries to read text data.

Audio files

- `audioread` - read audio files to a double array and **output sampling frequency**

Image files

- `imread` - read an image to an integer array

Video files

- `VideoReader` - creates a VideoReader object used to read a given video file
- `readFrame` - reads a frame of the VideoReader object

Exporting data and writing files

Numerical data

- `csvwrite` - write numerical only csv files
- `writecell` - write a cell to a csv file

Audio files

- `audiowrite` - write an array to an audio file based on the sampling frequency

Image files

- `imwrite` - write an integer array to an image file

Video files

- `VideoWriter` - creates a VideoWriter object used to write a given video file
- `writeVideo` - writes a frame of the VideoWriter object

What is a "signal?"

In engineering, a signal is simply a function that conveys information about a phenomenon. This could be a voltage that changes over *time*, a measured intensity that changes over *position*, the *spatiotemporal* concentration profile in a system, etc.

This lecture will introduce two common signals in signal processing:

1) An **audio** signal (represented by a "voltage" or acoustic pressure over time)

2) An **image** (represented by gray values at different pixel locations)

Introduction to audio processing

An audio file may be read using the `audioread()` function. There are two outputs when using this function: the **audio signal** and the **sampling frequency**.

```
% Read audio file
[y_signal,Fs] = audioread('clip0.mp4');

% Sampling frequency (Hz)
disp(Fs)
```

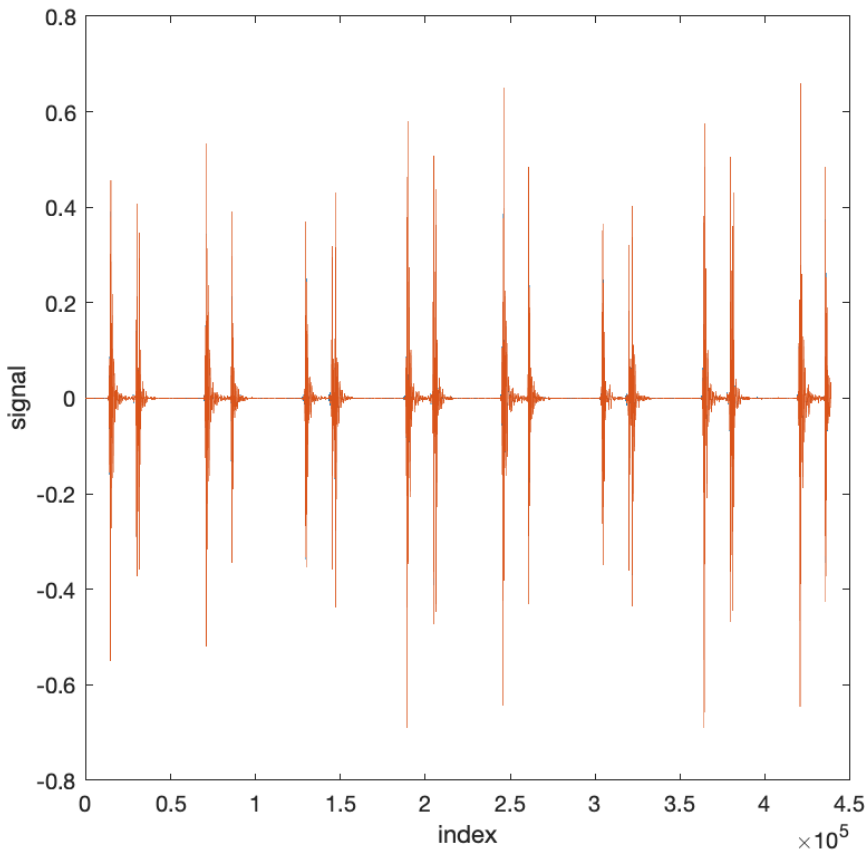
```
        44100
```

Audio signal: this is simply an `Nx2` array of type **double**. In MATLAB, each entry in the audio signal must fall in the interval `[-1,1]`.

```
% Size of audio signal
size(y_signal)
```

```
 ans = 1×2
       439232           2
```

```
% Plot audio signal
plot(y_signal);
axis square
xlabel('index')
ylabel('signal')
```



You can think of the y values as analagous to a voltage. Where you see "beats" corresponds to changes in this voltage. When such a voltage signal is applied to a speaker, e.g. in your headphones, it will cause some electricomechanical component to move (as defined by the time-varying voltages) and therby create vibrations (local air pressure fluctuations, compressions, expansions) in the surround air. We perceive these expansions and contractions as **sound**.

Note: this is the first audio signal you will deal with on HW9.

You can play a given audio signal in MATLAB using `sound()`. To play it, you need to input both the audio signal and the sampling frequency.

```
% Listen to sound
sound(y_signal,Fs)
```

Sampling frequency: this refers to the number of points in a **signal that is continuous in time** that are discretized.

In "real life," many signals may *seem* continuous -- when you hear someone speak, it seems like a continuous sound. That is, it seems like the sound *just* be represented as a function of time, $f(t)$.

However, recall that computation is inherently discrete.

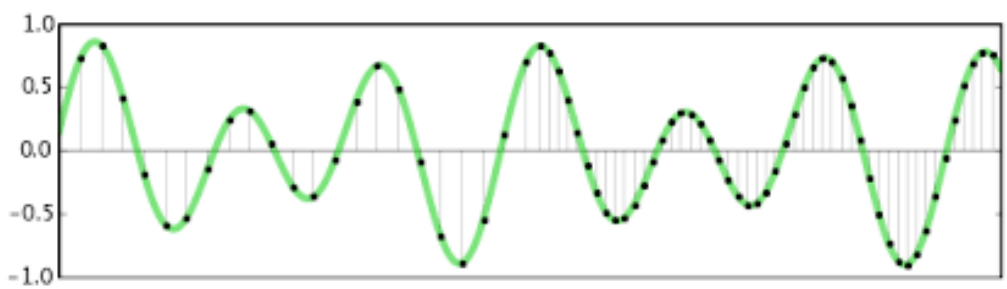
- Computation occurs at discrete physical locations
- Memory is stored at discrete physical locations
- The numbers that we use in MATLAB (class double) are 64-bit numbers that have some minimum and maximum possible values, and only *so many* possible decimal points. That is, the numbers and vectors are **discrete**.(see topic 14 / lecture 15).

To process signals, we have to "approximate" continuous signals as a sequence of discrete points to **perform any useful operations** (to store them, to analyze them, to play them, etc.).

- Consider a continuous signal, $f(t)$, represented by the green line. This signal could have an explicity formula, such as:

$$f(t) = \sin(t) + \sin(2t) + \cos(3t) + \sin(5t)$$

- The function only depends upon t and is defined for every possible t value -- we could in theory evaluate the function at arbitrary precision.



Source: Audacity

- To convert this signal into a discrete signal, we **sample** points along the function. That is, we select discrete points and evaluate the function at those points to approximate the function. **The number of such points that we evaluate per one second of the second is called the sampling frequency**.

The left side: lower sampling frequency -- less accurate representation of the signal

The right side: higher sampling frequency -- more accurate representaiton of the signal

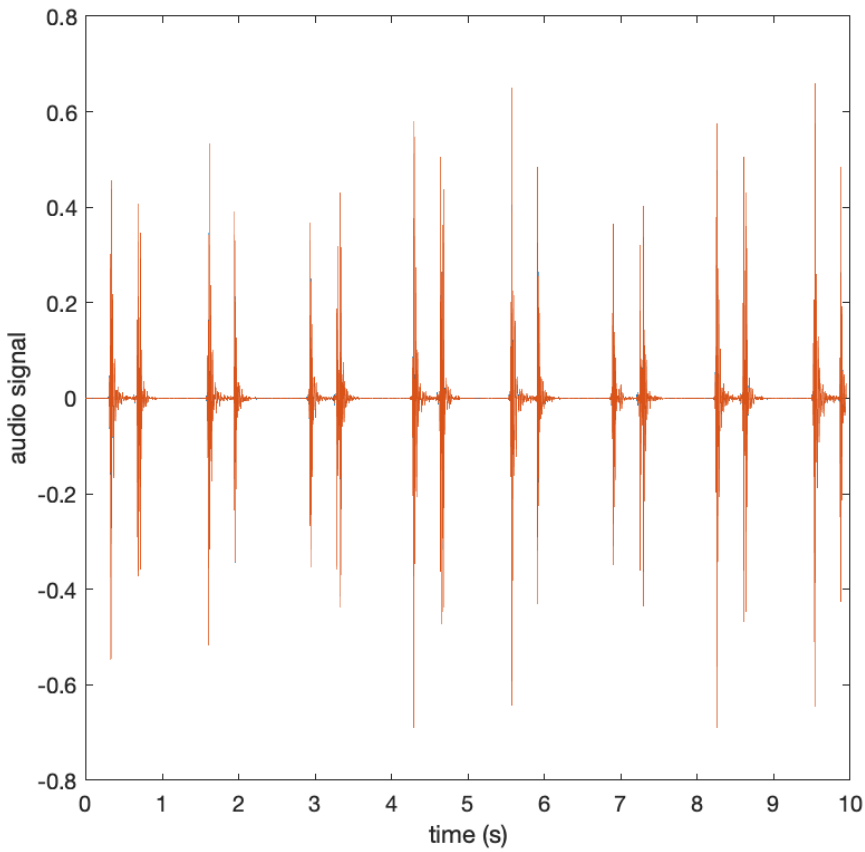
*There is a tradeoff: A **small** sampling interval results in a **better quality** sound file but **increases the size** of the audio file*

Note: most computers have an audio sampling frequency of 44.1 kHz, as above. That, there are 44,100 discrete points that represent the signal per second.

The sampling frequency contains information:

- The **number** of audio samples per second.
- Since the audio signal is an array where each entry corresponds to a sample, the **duration** of an audio signal (in units of seconds) can be easily determined.

```
% Calculate a time vector using the sampling frequency
time = 1/Fs*(1:size(y_signal,1)); % seconds

% Plot audio signal
plot(time,y_signal)
axis square
xlabel('time (s)')
ylabel('audio signal')
```



Stereo vs. mono sound

The size of the imported audio data often contains two "channels," or columns. This corresponds to **stereo** audio: the first column is output to the "left" speaker and the second column is output to the "right" speaker.

```
% Depict size of the data array
size(y_signal)
```

```
ans = 1×2
      439232           2
```

If only one column is input, it will be played as **mono** audio: both speakers will play the same thing.

```
% Listen to the first channel in mono
sound(y_signal(:,1),Fs)
```

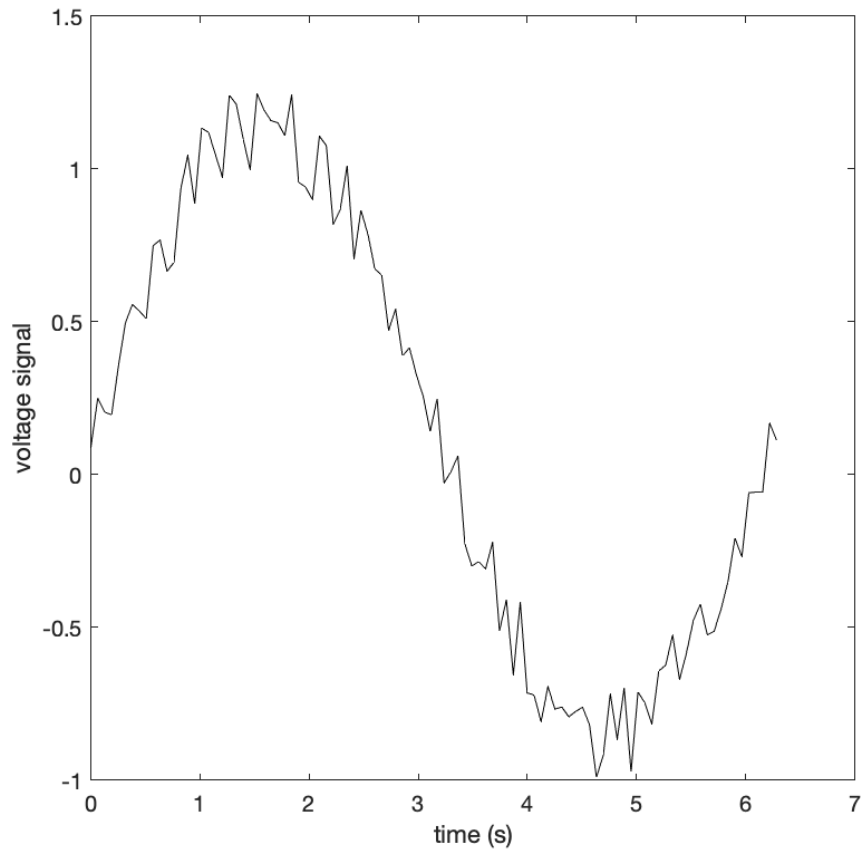
If only two columns are input, but one is all zeros, you will only hear output from one of the speakers

```
% Listen to only the first channel in stereo: you should only hear it in
% one side of your headphones/speakers
sound([y_signal(:,1),zeros(size(y_signal(:,1)))],Fs)
% Listen to only the first channel in stereo: you should only hear it in
% the other side of your headphones/speakers
sound([zeros(size(y_signal(:,1))),y_signal(:,1)],Fs)
```

Filtering a 1-D signal

A goal of signal processing is to obtain some information from a signal. The "raw data" of signal is often very noisy and difficult to obtain quantitative information from. For example, consider the following to be output from instrumentation that you have interfaced with MATLAB:

```
% Simulate noisy output from instrumentation
t = linspace(0,2*pi,100);
x = sin(t) + 0.3*rand(size(t));
% Plot noisy signal
plot(t,x,'k-')
axis square
xlabel('time (s)')
ylabel('voltage signal')
```



You are really just interested in the *low frequency* signal, that is the *general sinusoidal* shape.

- You wish to disregard the noise that is occuring because that could just be due to the instrumentation and not the signal that you care about.
- To get rid of the high frequency noise, you could apply a **low pass filter**.
- One type of low pass filter is a **rolling mean filter**.

<u>Note</u>: what is a "filter?" A filter is just some mathematical operation or computational algorithm that you **apply to an input signal** to produce a new output.

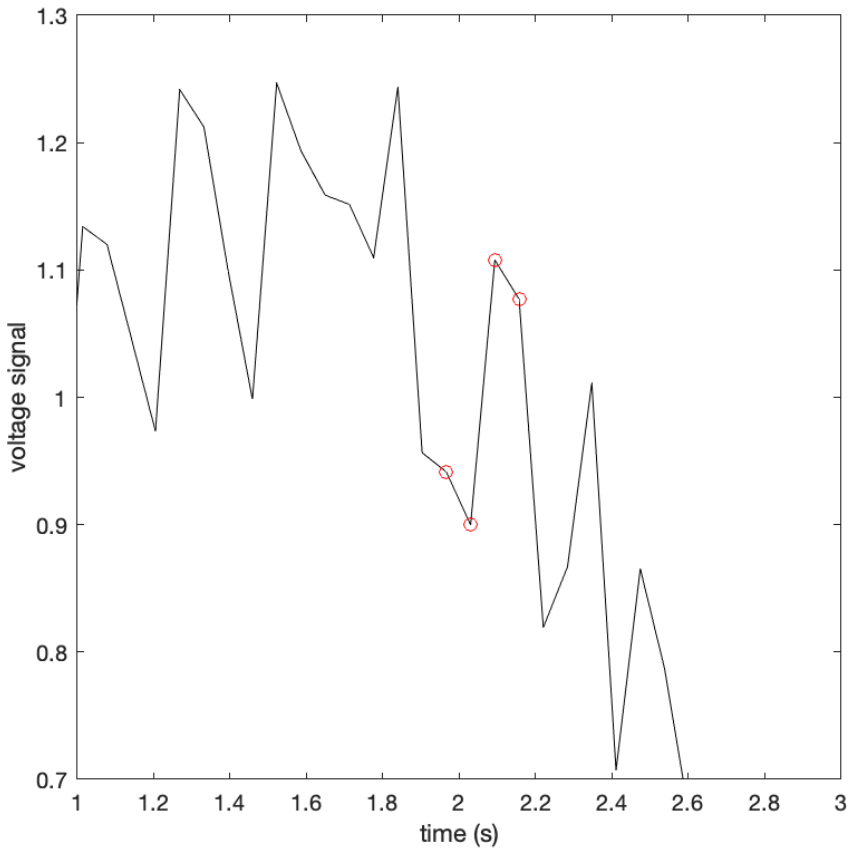
<u>A rolling mean filter</u>: averages the recent points and takes that average as the new data point in the filtered output.

$$y(n) = \frac{1}{\text{windowSize}}(x(n) + x(n-1) + \ldots + x(n - \text{windowSize} + 1))$$

- $y(n)$: the filtered output at point $x = n$
- $x(n)$: the raw, unfilterd signal at $x = n$.

```
% Example: rolling filter applied to the data point n = 36 with a window
% size of 4. In this plot, the four data points will be averaged together,
```

```
% and the "filtered output" at the THIS data point will be this average.

% Plot noisy signal
plot(t,x,'k-')
hold on
plot(t(32:35),x(32:35),'ro')
hold off
axis square
xlabel('time (s)')
ylabel('voltage signal')
xlim([1,3])
ylim([0.7,1.3])
```



1-D rolling mean filters can be created using the `filter()` function in MATLAB.

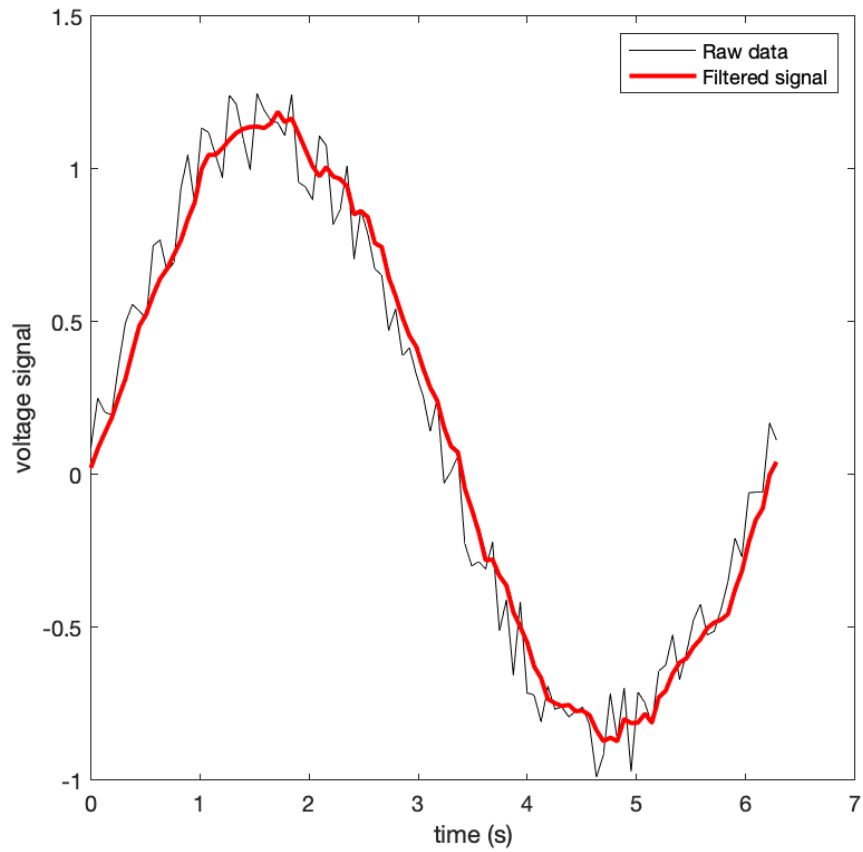
$$y(n) = \frac{1}{\text{windowSize}}\left(x(n) + x(n-1) + \ldots + x(n - \text{windowSize} + 1)\right)$$

The filter takes inputs based on the desired **transfer function**. This is outside of the scope of this class and is related to the process of ***convolution***. For now, just note the parallels with the desired equation giving the output.

```
% Define the window
windowSize = 4;
% The numerator of the transfer function
% Note: 1/windowSize is the coefficient. Based on a window size of 4, we
% want to analyze x(n) ... x(n - 4 + 1). That is, we want to analyze the
% four "most recent" x values. This information is contained within the
% ones vector
b = ones(1,windowSize);
% The denominator of the transfer function
a = windowSize;
```

Applying the filter:

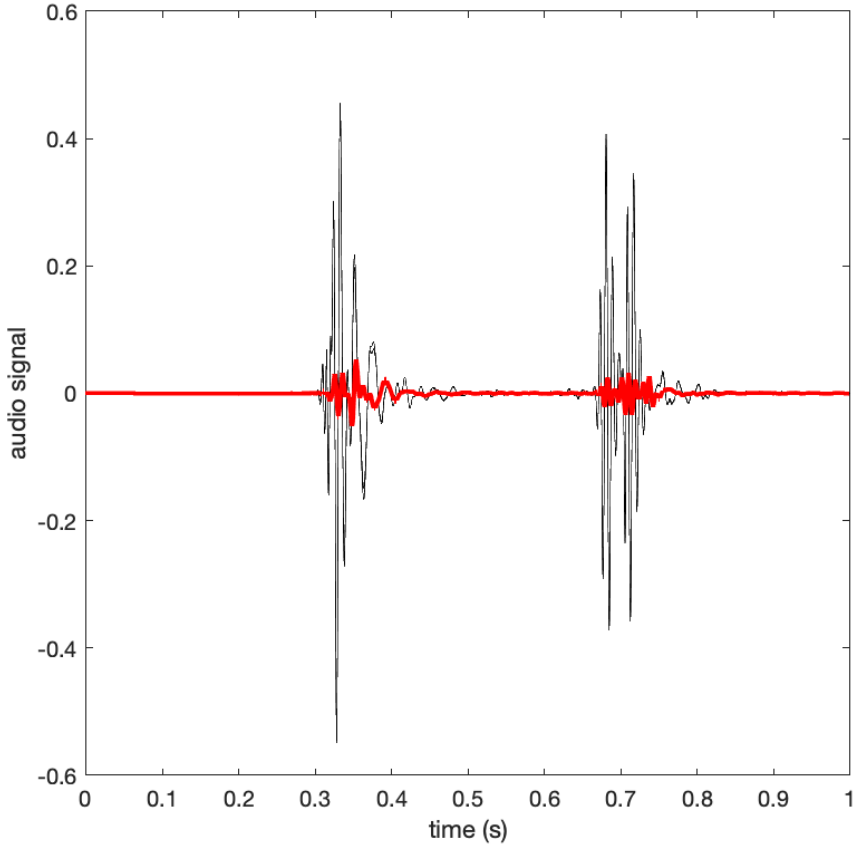
```
close all % Close previous plots

% Create the filtered output
y = filter(b,a,x);

% Plot noisy signal and filtered signal on same plot
g1 = axes;
plot(t,x,'k-')
hold on
plot(t,y,'r-','LineWidth',2)
hold off
axis square
xlabel('time (s)')
ylabel('voltage signal')
legend('Raw data','Filtered signal','Location','northeast')
```



What about filtering the heart beat signal? We could apply the same rolling mean filter in this case.

```
close all % close previous figures

% Create a rolling mean filter
windowSize = 1000;
b = ones(1,windowSize);
a = windowSize;

y_filtered = filter(b,a,y_signal);

% Plot audio signal
plot(time,y_signal,'k-')
hold on
plot(time,y_filtered,'r-','LineWidth',2)
hold off
axis square
xlabel('time (s)')
ylabel('audio signal')
```

```
xlim([0,1])
```



Note: you can incorporate additional mathematical functions to produce different outputs from the filter.

It is up to you, as the engineer, to decide how to apply mathematical and computational tools to best solve a problem. In this case, that means to convert it into something that provides useful information

Creating audio signals

The `sound` function, used above, enables you to play an audio signal on your computer's speakers. Since you can think of an audio signal as a *voltage that dictates how some electromechanical component of your speakers will move*, we can create a **steady oscillation at a defined frequency** to produce a given **pitch**.

```
% Create a time interval that is dependent upon the sampling frequency
Fs = 44100;
t = linspace(1/Fs,1,Fs);
% Produce a note corresponding to "middle C"
y_sound = sin(2*pi*262*t);
```

```
sound(y_sound,Fs)
```

Note: the above vector, y_sound, oscillates at a frequency of 262 times per second. This frequency corresponds to a "middle C". The above vector is sampled 44,100 times per second. Note the difference between these two frequencies.

Note: changing the "musical frequency" will alter the pitch of the sound.

Introduction to image processing

An image **may be read into MATLAB** using the `imread` function.

```
% Read the first image
image1 = imread('L19_image1.jpg');
```

An image may be displayed using `imshow`:

```
imshow(image1)
```

Images are another kind of signal -- in this case, these signals represent **spatial** information.

- **An image is essentially a matrix of integer values**. The dimensions of this matrix correspond to the number of pixels in your image. The dimensions correspond to *rows and columns of pixels*.

```
% Dimensions of an image
size(image1)
```

```
ans = 1×2
```

```
        972         1829
```

```
% Total number of pixels
numel(image1)
```

```
ans = 1777788
```

In this case, the picture is 1.7 MPx (mega-pixels).

In each matrix position, there is an integer number. This is the **gray value** of that pixel.

- In this case, I have gray (not color) image and so each pixel represents the shade of gray of that pixel.
- This integer can take any values within the allowable range of the ***bit depth*** of the image. In this case, the above image is an 8-bit image, and so a value of `0` corresponds to **black** and a value of `255` corresponds to **white**.

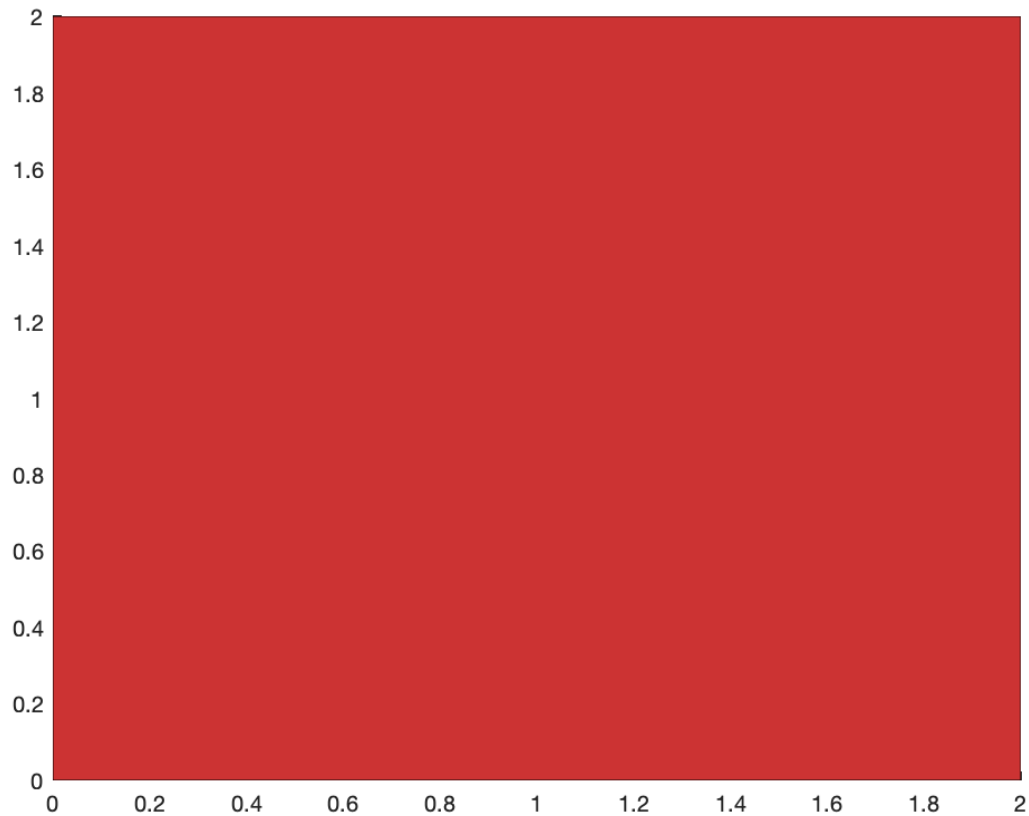
```
% Bit depth of image
class(image1)
```

```
ans = 'uint8'
```

Color images

We have seen representations of color previously in terms of an R-G-B color vector, where the value of each components dictates how red/green/blue the desired color should be. For example, using the rectangle function we discussed last time:

```
close all % close all figures
% Define color
myColor = [0.8,0.2,0.2];
axes
rectangle('Position',[0,0,2,2],'FaceColor',myColor)
```

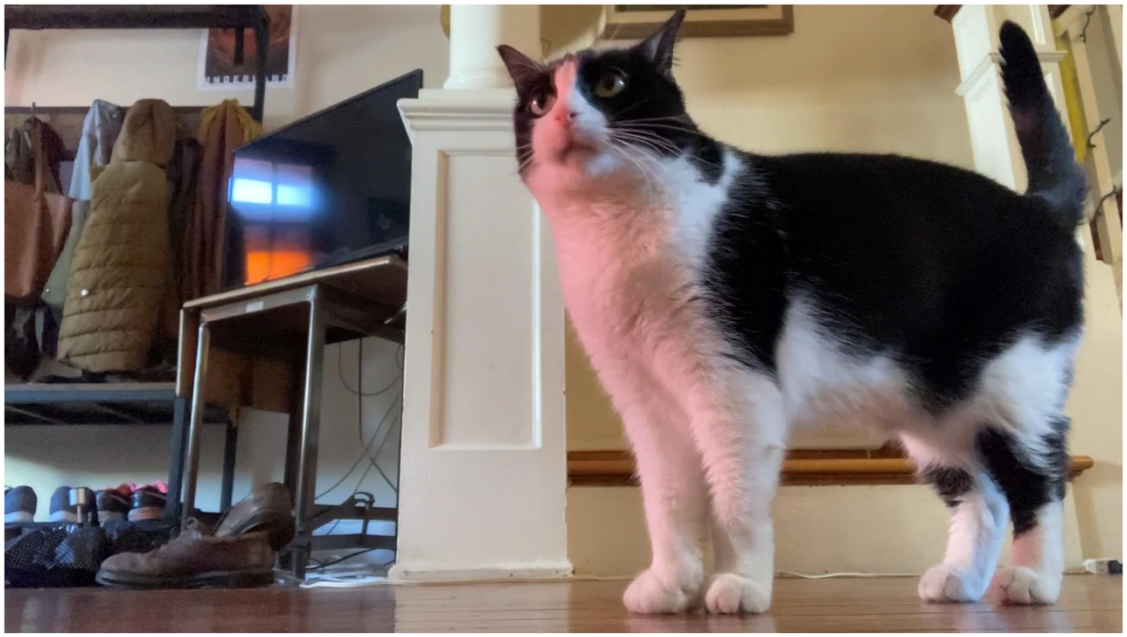


Color images are represented in the same way. In this case, each pixel location has a 3 component vector describing what the color of that pixel should be. To represent the color of the entire image, a color image is represented ***with three channels.***

```
% Load a similar color image
image1_color = imread('L19_image1color.jpg');
% Depict dimensions of color image
size(image1_color)
```

```
ans = 1×3
        1080        1920           3
```

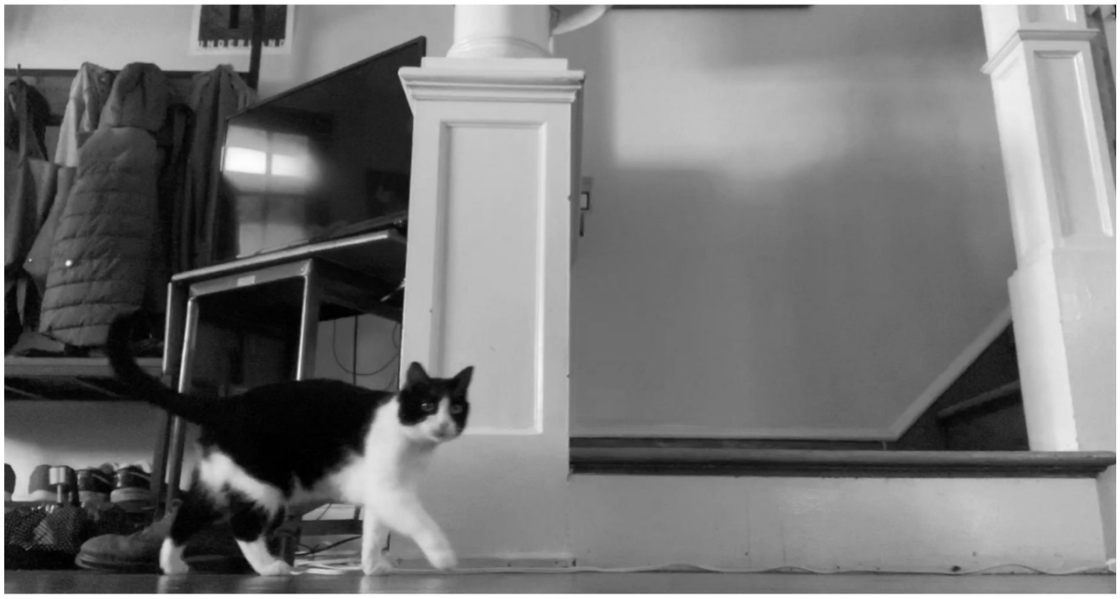
```
% Show color image
imshow(image1_color)
```



Loading multiple signal files

Structure arrays are often a convenient way to store multiple signals or images.

```
% Number of images
numIm = 3;
% Character array that is common to all image files
baseName = 'L19_image';

% Define imData as a structure array
imData = struct;
for k = 1:numIm
    imData(k).pic = imread([baseName,num2str(k),'.jpg']);
end
% Show desired image
imshow(imData(3).pic)
```



Example: removing the cat from the image

Say we are only interested in a picture of the background and that the cat (as cats do) is just being nosy and getting in the way. To produce a picture of the background from our three images, **we can remember that images are *just* matrices with numbers**. As such, you can perform **basic mathematical and array operations** on them.

```
% Goal: create a 3-D matrix, where each image is on one *page* of this
% matrix

% Preallocate matrix
imDataMatrix = zeros([size(imData(3).pic),3]);
% Populate each page of the matrix
for k = 1:3
    imDataMatrix(:,:,k) = imData(k).pic;
end
% Ensure imDataMatrix is the correct data type
imDataMatrix = uint8(imDataMatrix);

% Dimensions of this image "matrix"
size(imDataMatrix)
```

```
ans = 1×3
         972        1829           3
```

In a given pixel location for the three images, that pixel is the "background" in two of them, and the cat in one of them. There are no two pictures where the cat is in the same pixel location. Thus, since we have ***three data points for each pixel***, we can select the median pixel, as this will likely

represent ***just the background.*** *Thus, to create a picture without the cat*, we can simply **find the median gray value in each pixel location**.

Note: remember that when performing mathematical functions on an array, you can specify the dimension to operate over.

- Dim 1: operate over rows
- Dim 2: operate over columns
- Dim 3: operate over pages

```
% Find "median" image
image_median = median(imDataMatrix,3);
imshow(image_median)
```

Takeaway: images are just matrices of integers. From these three images, we can construct an image of just the background pixels using just the `median()` operation, even though **none** of the individual images were without that cat,

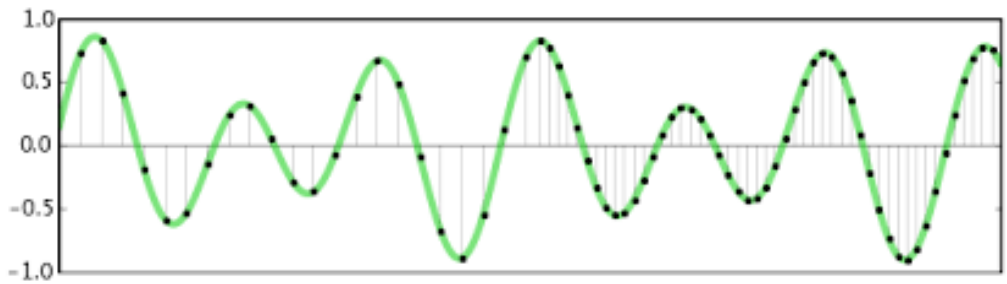
TOPIC 19: Revisiting images and graphics

ENGR105, 11/5/20

Reminder: audio processing

- ***Audio signal***: this is simply an `Nx2` array of type **double**. In MATLAB, each entry in the audio signal must fall in the interval `[-1,1]`. We can import audio using `audioread` and playback using `sound`.

- ***Sampling frequency***: The number of points per second used to discretize a continous signal.



Source: Audacity

- ***Remember***: audio data is just an array of decimal values. This can be manipulated using all of our previous array operations.

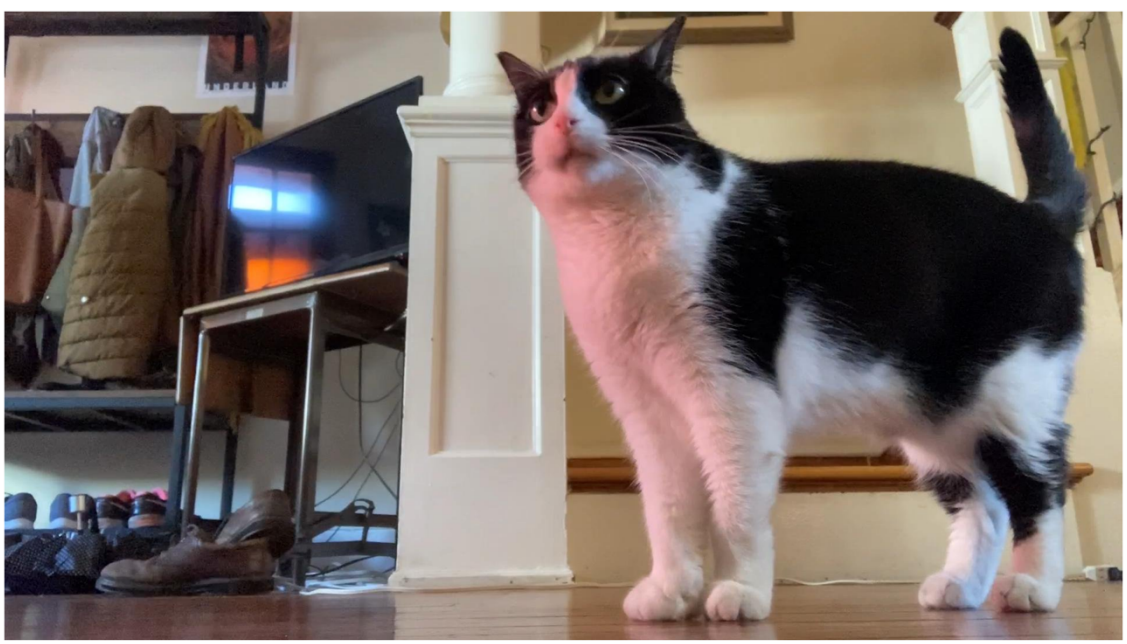
Reminder: image processing

An image **may be read into MATLAB** using the `imread` function.

```
% Read the first image
image1 = imread('L19_image1color.jpg');
```

An image may be displayed using `imshow`:

```
imshow(image1)
```



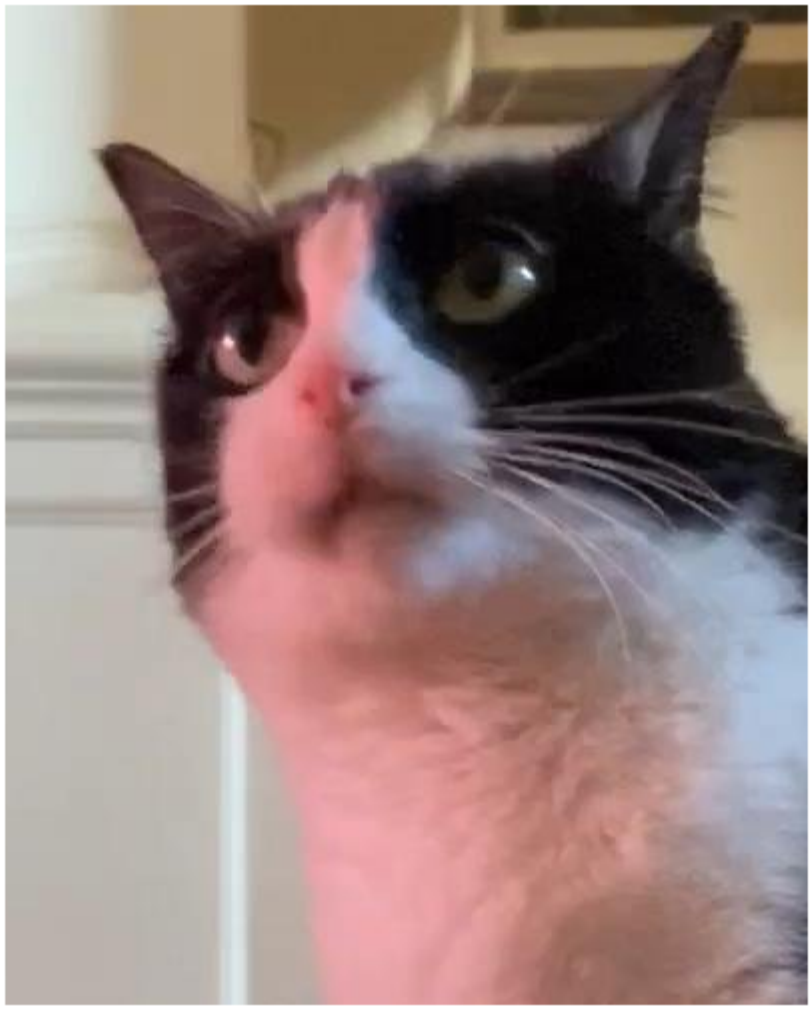
Images are just "integer matrices," in which the each index corresponds to a unique **pixel**, and the value of the corresponding element is the **gray value** of that pixel. The bit depth of an image corresponds to the range of possible integer values each element can take.

```
% Bit depth of image
class(image1)
```

```
ans = 'uint8'
```

Images can be **cropped** using **array indexing**.

```
% Create a cropped image: note the array indices
imX = image1(1:500,800:1200,:);
imshow(imX)
```



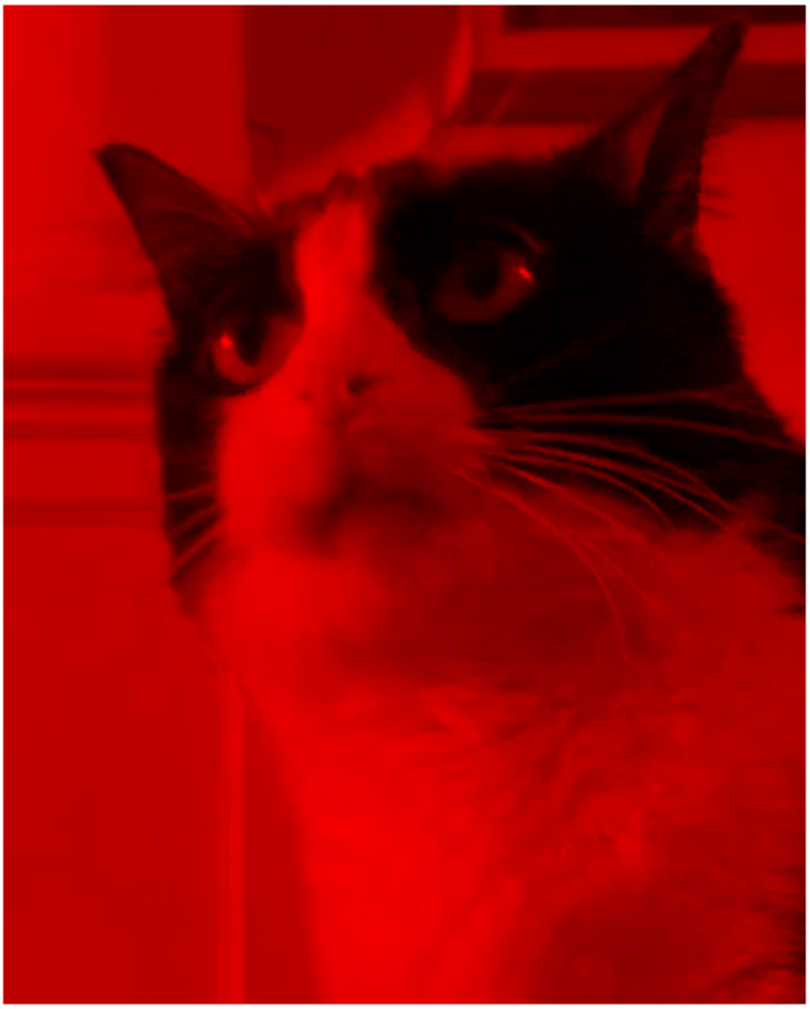
Grayscale images have one value associated with each **X-Y index position**, whereas color images have three values (R-G-B). As a result, color images have **<u>three dimensions</u>**.

```
% Dimensions of image1
size(imX)
```

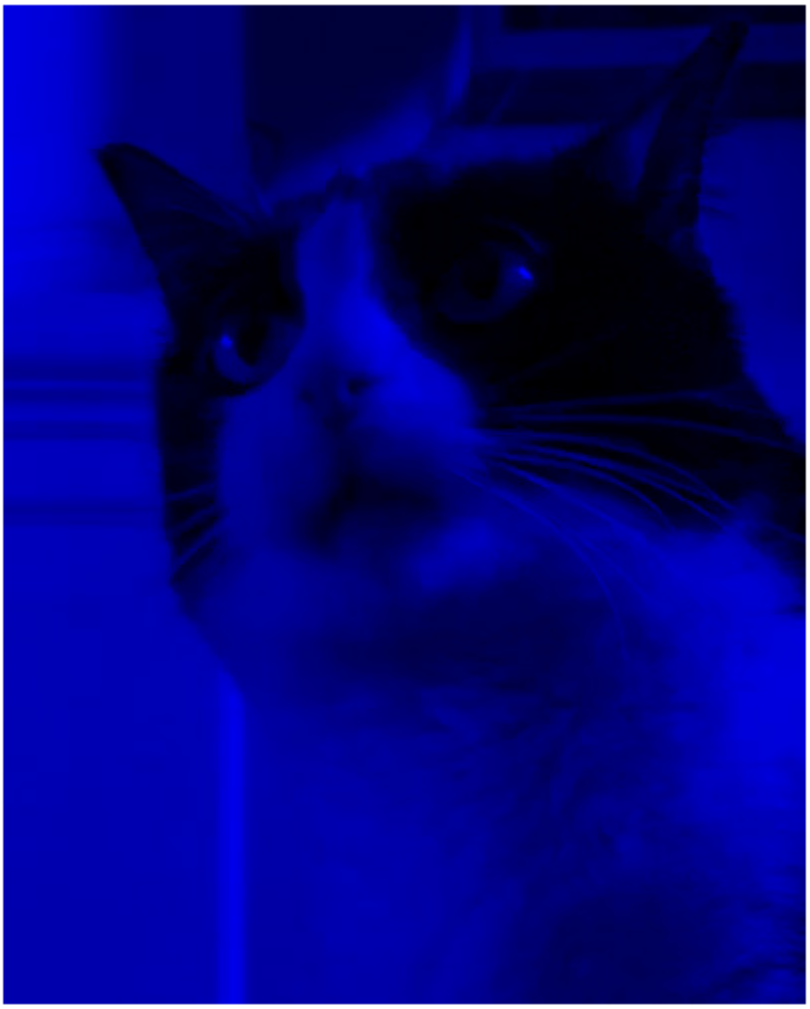
```
ans = 1×3
   500   401     3
```

Since the "third dimension" corresponds to the **color**, each "channel" (*red, green, or blue*) can be displayed by setting the other channels to zero.

```
% Display red channel
imX_red = imX;
imX_red(:,:,2:3) = 0;
imshow(imX_red)
```



```
% Display blue channel
imX_blue = imX;
imX_blue(:,:,1:2) = 0;
imshow(imX_blue)
```



Displaying one color channel as a grayscale image

```
% Create grayscale image from green channel
imX_gray = imX(:, :, 2);
imshow(imX_gray)
```

By changing the pixel value of image elements, you can change the image that will be displayed.

```
% Change the "brightness" of a given pixel
imshow(imX_gray + 100)
```

You can use **logical indexing** on a given image to obtain or modify specific pixel values.

```
% Preallocate array to store information about image to be produced
imX2 = uint8(zeros(size(imX_gray)));

% Loop through some values to convert the image into "buckets"
% of only a few gray values
gvThresh = [0,50,120,200,255];
for k = 1:length(gvThresh)-1
    % All pixel between corresponding gray value limits are set to the
    % lower limit
    imX2(imX_gray > gvThresh(k) & imX_gray <= gvThresh(k+1)) = gvThresh(k);
end
```

```
% Display image
imshow(imX2)
```

<u>Note</u>: the above example uses a `for` loop -- don't forget about vectorization

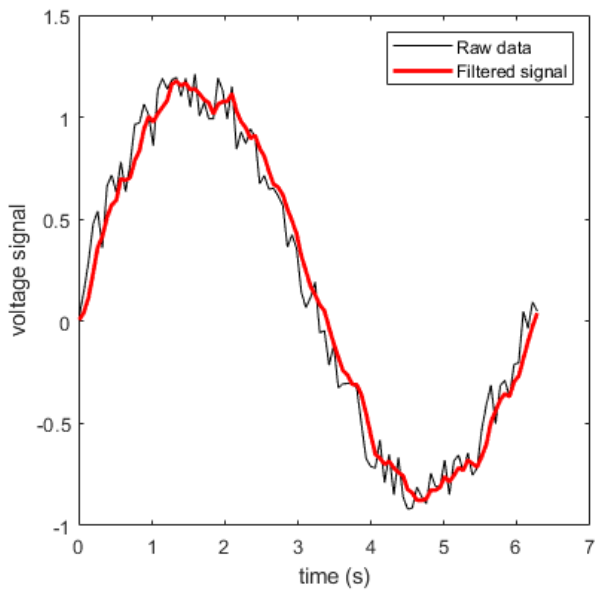
Basic image filtering

Last time, we used `filter` to apply a **rolling mean filter** to a temporal, 1-D signal:

$$y(n) = \frac{1}{\text{windowSize}}(x(n) + x(n-1) + \ldots + x(n - \text{windowSize} + 1))$$

- $y(n)$: the filtered output at point $x = n$
- $x(n)$: the raw, unfilterd signal at $x = n$.

The result was us being able to remove high frequency noise to effectively smoothe the signal.



One can apply a similar concept to images: we can use `filter2` to create a "*2-D mean filter*" to remove "high frequency" details. As a result, the filter image will appear "**blurry**." A few notes:

- This filtering is the result of **2D convolution** of the **image** with a **kernel**.
- This **2-D kernel is equivalent to the 1-D "window"** in the example above. For *every pixel location*, it will take the average of all nearby pixel values.
- Note that the kernal is **typically normalized by the sum of all elements.**

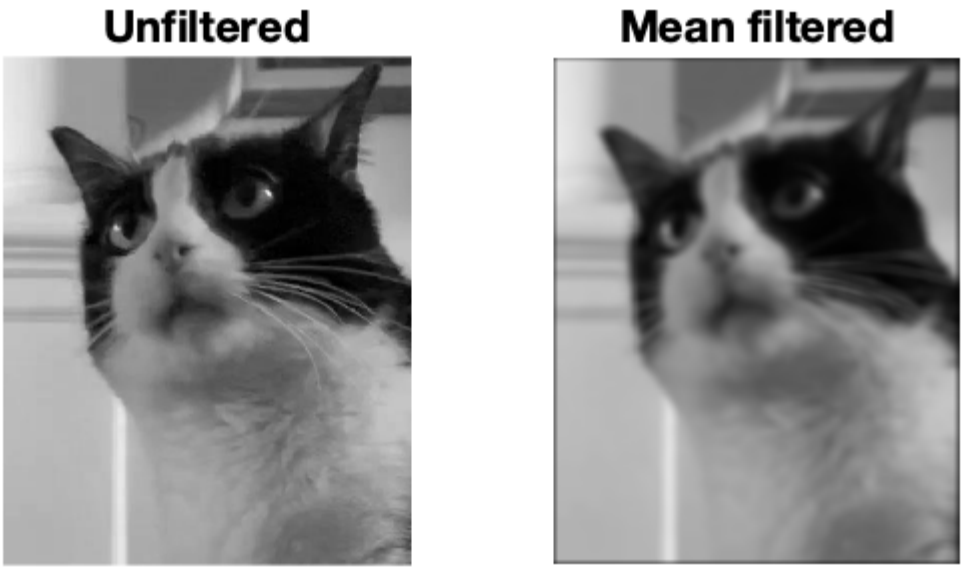
```
% Create the kernel
B = ones(10);
B = B/sum(B,'all');

% Filter image using filter2
imX_filt = filter2(B, imX_gray)
```

```
imX_filt = 500×401
   83.1000   96.8400  110.5800  124.3200  138.0000  137.7000  137.4600
 137.1600  136.9200  136.6200  136.2600  135.9600  135.7200  135.4800
 135.2400  134.9400  134.5800  134.2800  133.9200  133.6200  133.2600
 132.9000  132.4200  131.9400  131.4000  130.8600  130.3800  129.9000
 129.4200  128.9400
   96.9500  112.9800  129.0100  145.0400  161.0000  160.6500  160.3700
 160.0200  159.7400  159.3900  158.9700  158.6200  158.3400  158.0600
 157.7800  157.4300  157.0100  156.6600  156.2400  155.8900  155.4700
 155.0500  154.4900  153.9300  153.3000  152.6700  152.1100  151.5500
 150.9900  150.4300
  110.8000  129.1200  147.4400  165.7600  184.0000  183.6000  183.2800
 182.8800  182.5600  182.1600  181.6800  181.2800  180.9600  180.6400
 180.3200  179.9200  179.4400  179.0400  178.5600  178.1600  177.6800
```

```
177.2000  176.5600  175.9200  175.2000  174.4800  173.8400  173.2000
172.5600  171.9200
  124.6500  145.2600  165.8700  186.4800  207.0000  206.5500  206.1900
205.7400  205.3800  204.9300  204.3900  203.9400  203.5500  203.1600
202.7900  202.3300  201.7900  201.3200  200.7600  200.2800  199.7400
199.1900  198.4900  197.7700  196.9500  196.1300  195.4100  194.7100
194.0200  193.3500
  138.5000  161.4000  184.3000  207.2000  230.0000  229.5000  229.1000
228.6000  228.2000  227.7000  227.1000  226.6000  226.1400  225.6800
225.2600  224.7400  224.1400  223.6000  222.9600  222.4000  221.8000
221.1800  220.4200  219.6200  218.7000  217.7800  216.9800  216.2200
215.4800  214.7800
  138.5000  161.4000  184.3000  207.2000  230.0000  229.5000  229.1000
228.6000  228.2000  227.7000  227.1000  226.6000  226.1100  225.6200
225.1900  224.6600  224.0600  223.5000  222.8400  222.2500  221.6500
221.0200  220.2800  219.4800  218.5500  217.6200  216.8200  216.0800
215.3700  214.7200
  138.5000  161.4000  184.3000  207.2000  230.0000  229.5000  229.1000
228.6000  228.2000  227.7000  227.1000  226.6000  226.0800  225.5600
225.1200  224.5800  223.9800  223.4000  222.7200  222.1000  221.5000
220.8600  220.1400  219.3400  218.4000  217.4600  216.6600  215.9400
215.2600  214.6600
  138.5000  161.4000  184.3000  207.2000  230.0000  229.5000  229.1000
228.6000  228.2000  227.7000  227.1000  226.6000  226.0500  225.5000
225.0500  224.5000  223.9000  223.3000  222.6000  221.9500  221.3500
220.7000  220.0000  219.2000  218.2500  217.3000  216.5000  215.8000
215.1500  214.6000
  138.5000  161.4000  184.3000  207.2000  230.0000  229.5000  229.1000
228.6000  228.2000  227.7000  227.1000  226.6000  226.0200  225.4400
224.9800  224.4200  223.8200  223.2000  222.4800  221.8000  221.2000
220.5400  219.8600  219.0600  218.1000  217.1400  216.3400  215.6600
215.0400  214.5400
  138.5000  161.4000  184.3000  207.2000  230.0000  229.5000  229.1000
228.6000  228.2000  227.7000  227.1000  226.6000  225.9900  225.3800
224.9100  224.3400  223.7400  223.1000  222.3600  221.6500  221.0500
220.3800  219.7200  218.9200  217.9500  216.9800  216.1800  215.5200
214.9300  214.4800
```

```
% Convert filtered image to 8-bit
% NOTE: images must be 8-bit integer values
imX_filt = uint8(imX_filt);

% Create a subplot showing both images
% First image: unfiltered
subplot(1,2,1)
imshow(imX_gray)
title('Unfiltered')
% Second image: mean filtered
subplot(1,2,2)
imshow(imX_filt)
```

```
title('Mean filtered')
```



Note: the above is a simple kernel for mean filtering an image. Other kernels exist for edge detection, image sharpening, and applying a Gaussian blur.

Binary (logical) images

Converting an image to a **logical array** is known as creating a **binary image** (or black/white image or logical image).

```
% Close all previous images
close all
% Create a binary image from imX2
imX_bw = imX_filt<50;
```

```
% Depict properties of the black/white image
whos imX_bw
```

```
  Name          Size              Bytes  Class      Attributes

  imX_bw      500x401            200500  logical
```

```
% Display image
imshow(imX_bw)
```

Binary images are useful for extracting and analyzing "objects" in an image. The following are two useful functions

- `bwlabel` -- labels (isolates) the different connected components of a binary image.

- `imfill` -- perform flood fill operation, convert dark (0) pixels to white (1). The most common setting I use is to fill `'holes'` -- i.e. fill in dark (0) pixels that are internally present within a white (1) object.
- `regionprops` -- gives quantiative information about a binary "object" in an image.

Example. Let's apply all of these functions to the above black/white image

```
% 1) Label all (1) objects in binary image
[L, num] = bwlabel(imX_bw);

% The number of unique binary objects
num
```

```
num = 12
```

```
% Inspect different binary objects
imshow(L == 5)
```

```
% 2) Fill in dark (0) pixels INSIDE object
% Store specific binary image as a variable
im = L == 5;
im2 = imfill(im,'holes')
```

```
im2 = 500×401 logical array
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0																			

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0																			

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0																			

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0

1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

```
0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0
```

```
imshow(im2)
```

```
% Use regionprops to obtain information from the isolated object
pic_info = regionprops(im2, 'all')
```

```
pic_info = struct with fields:
                    Area: 23651
                Centroid: [294.7183 151.5875]
             BoundingBox: [188.5000 9.5000 213 281]
             SubarrayIdx: {[1×281 double]  [1×213 double]}
         MajorAxisLength: 234.9789
         MinorAxisLength: 166.7343
            Eccentricity: 0.7046
```

```
           Orientation: -46.0300
            ConvexHull: [110×2 double]
           ConvexImage: [281×213 logical]
            ConvexArea: 36063
           Circularity: 0.3221
                 Image: [281×213 logical]
           FilledImage: [281×213 logical]
            FilledArea: 23651
           EulerNumber: 1
               Extrema: [8×2 double]
        EquivDiameter: 173.5321
             Solidity: 0.6558
               Extent: 0.3952
         PixelIdxList: [23651×1 double]
            PixelList: [23651×2 double]
            Perimeter: 960.5990
         PerimeterOld: 1.0112e+03
      MaxFeretDiameter: 288.6243
         MaxFeretAngle: -136.6847
   MaxFeretCoordinates: [2×2 double]
      MinFeretDiameter: 194.3400
         MinFeretAngle: 170.7601
   MinFeretCoordinates: [2×2 double]
```

Note: `pic_info` is a **structure array**, where each field can be accessed using dot notation, as before.

```
% Area in pixels of binary object
a = pic_info.Area
```

```
a = 23651
```

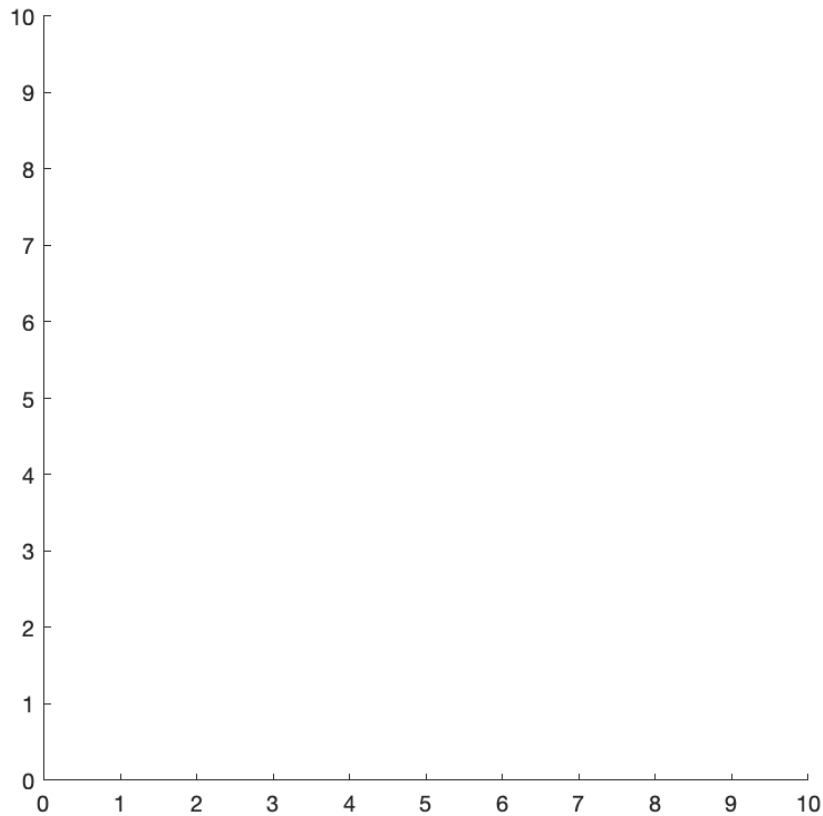
```
% How "circular" the object is
b = pic_info.Circularity
```

```
b = 0.3221
```

Revisiting plotting and graphics

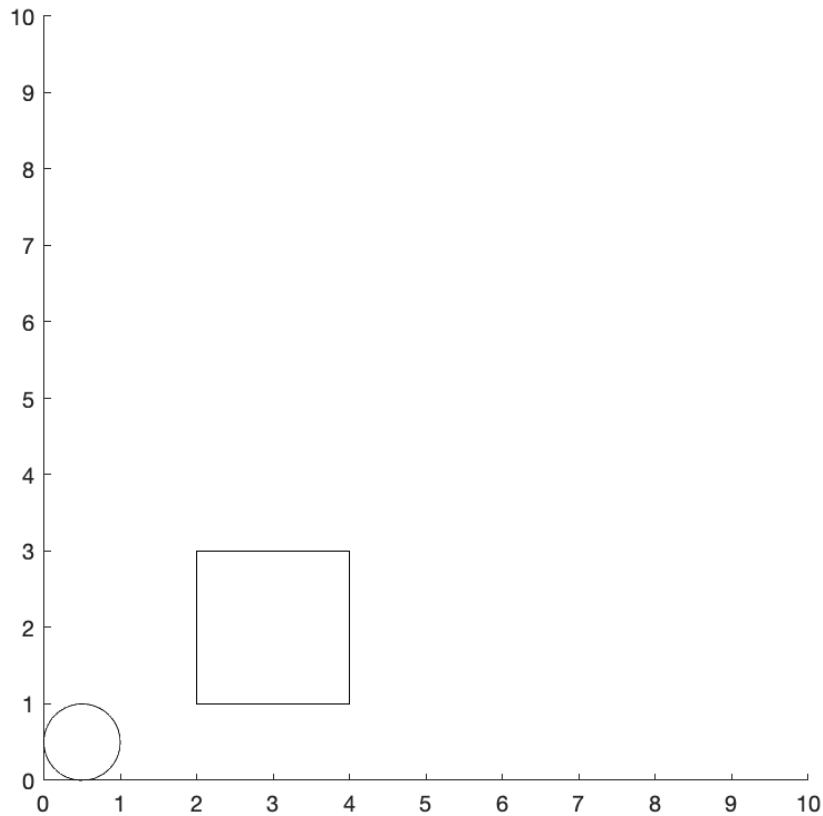
A few **useful functions** for creating and adding graphics:

```
% First, create a blank set of axes to add to
close all
axes;
xlim([0,10]);
ylim([0,10]);
axis square
```



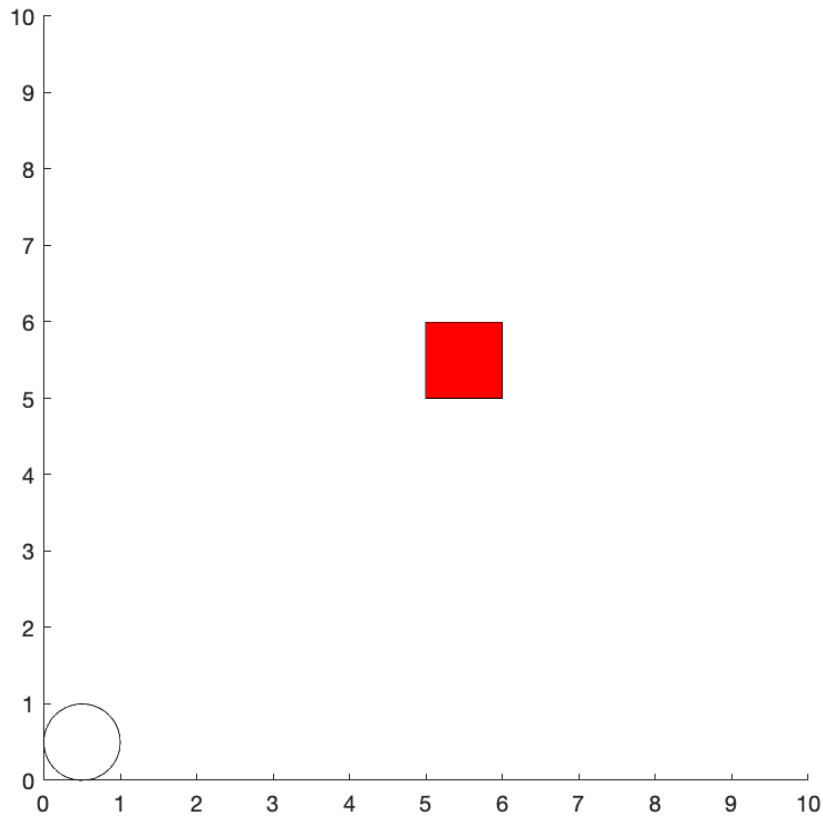
- `rectangle` -- adds a rectangle, rounded-rectangle, or ellipse

```
% Add two rectangles
h1 = rectangle('Position',[2,1,2,2]);
h2 = rectangle('Position',[0,0,1,1],'Curvature',[1,1]);
```



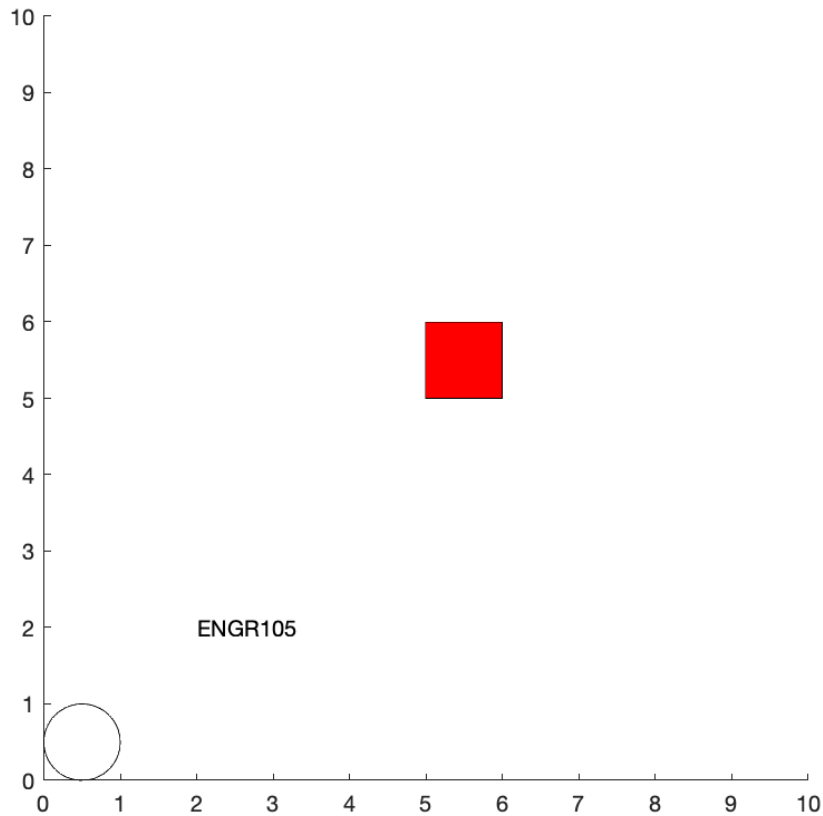
As before, graphics object parameters can be changed and accessed by referencing the **graphics object handle**.

```
h1.FaceColor = 'r';
h1.Position = [5,5,1,1];
```



- `text` -- adds text to a plot at the defined position

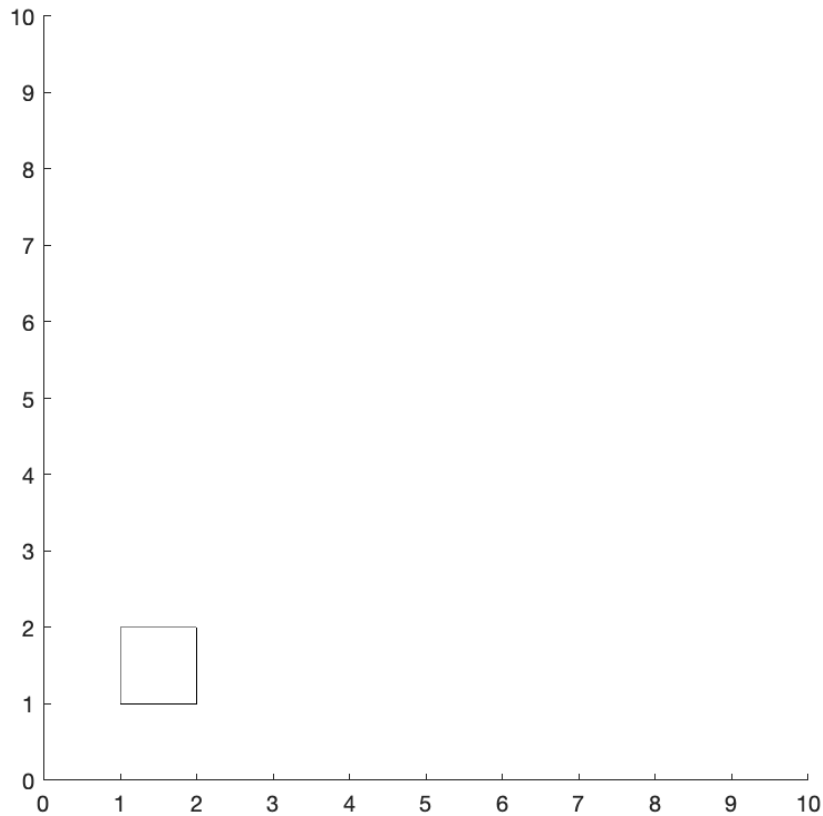
```
text(2,2,'ENGR105')
```



Animating objects

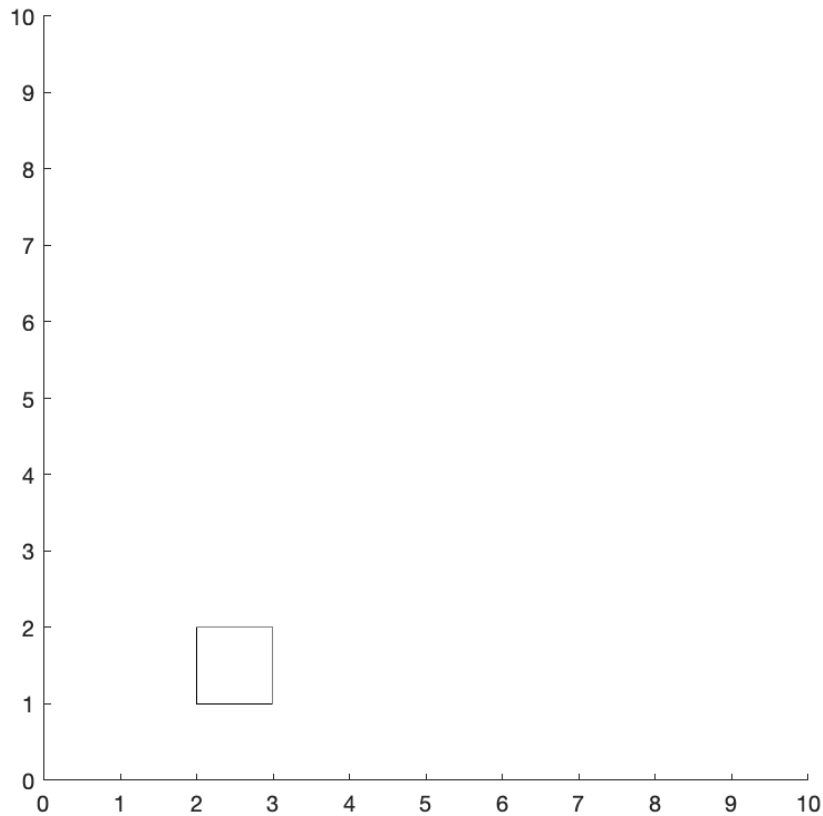
In lecture 18, we **imported positional data** for creating a visualization of particle motion.

```
figure
axis square
xlim([0,10])
ylim([0,10])
h3 = rectangle('Position',[1,1,1,1]);
```



One could also specify positions based on a **mathematical expression.**

```
% Create time and positional vectors
t = 1:10;
x = cos(t/10*2*pi)+1;
y = sin(t/10*2*pi)+1;

% For each time point specified above, move the rectangle to a new spot
for k = t
    % Pause MATLAB for 0.5 seconds
    pause(0.5)
    % Update to new position
    h3.Position = [x(k),y(k),1,1];
end
```

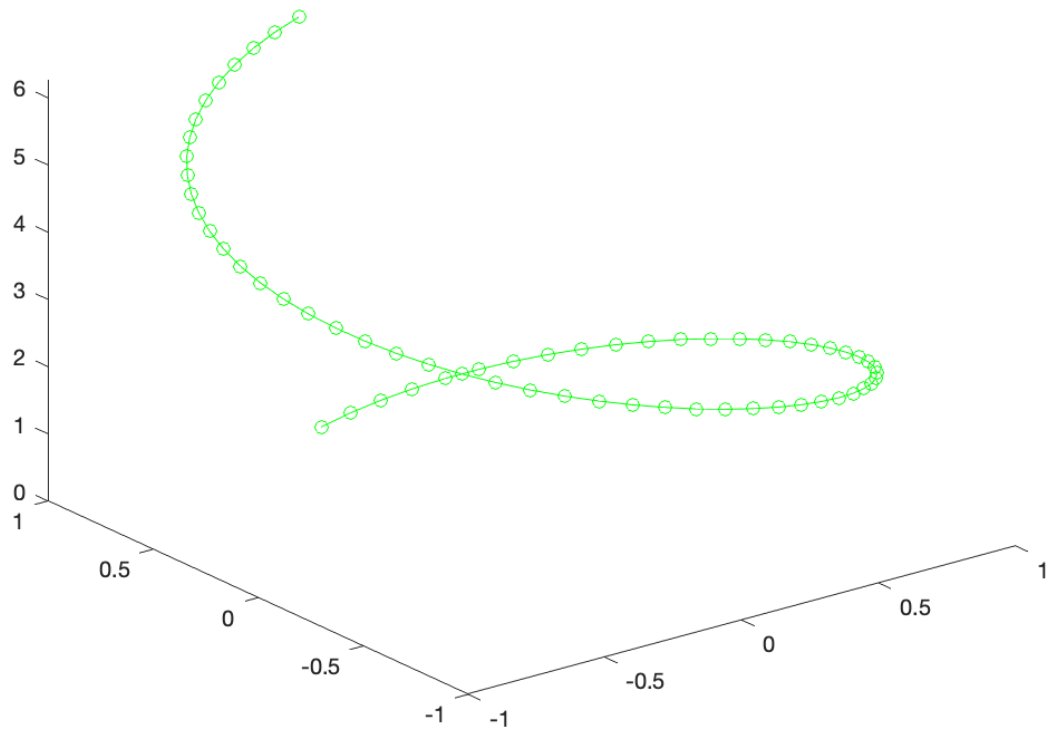


<u>Note</u>: `pause(t)` pauses for `t` seconds

Example. Plot (in 3D) $x(t) = \sin(t)$, $y(t) = \cos(t)$, $z(t) = \cos(t)$ for $t = [0, 2\pi]$

```
% New plot
figure;

% Step 1: create vectors for time and corresponding positions
t = 0:.1:2*pi;
x = sin(t);
y = cos(t);
z = t;

% Step 2: Plot nothing and establish axis bounds.
p = plot3(NaN,NaN,NaN,'go-');
xlim([-1,1])
ylim([-1,1])
zlim([0,2*pi])

% Step 3: the plot over time
```

```
for jj = 1:length(x)
    p.XData = x(1:jj);
    p.YData = y(1:jj);
    p.ZData = z(1:jj);
    pause(.1)   % Pause for 100 ms
end
```



Keyboard callbacks

Keyboard interactions with a script or interface often make for a more useful and interactive experience. Such interfaces may include movement of on-screen objects by keyboard direction, selection points or regions on a pair of axes, or selecting options or menu items. Interaction between the keyboard and your user interface/program are achieved using keyboard callbacks.

Additional resources:

- MathWorks documentation
- Callback definitions

The general format for adding a **keyboard callback** to a figure is as follows. The `'KeyPressFcn'` field is just a parameter associated with a figure graphics handle.

```
% Figure to be associated with callback
h1 = figure;
% Specify the function that will be executed upon a key press event
h1.KeyPressFcn = @FctnToCall
```

The corresponding callback function stored at the end of the document:

```
% Callback function.
function FctnToCall(hObj,eventdata)
    if strcmp(eventdata.Key,'w')
      disp('Pressed a w!')
    end
end
```

Notes:

- **Keyboard callbacks are often associated with a `figure`.**
- `hObj` refers to the calling object but is unnecessary in most cases, *i.e.* `hObj` could be listed as ~ in most situations.
- `eventdata` contains several fields, one of which is the keyboard key pressed

Useful: to view the character codes or names associated with a keystroke, type the following into the command line:

```
run(fullfile(docroot,'techdoc','ref','examples','ex_KeyPressFcn'));
```

Example 1. Printing to command line by pressing the `'w'` key.

```
close all % Close all previous figures
% Figure to be associated with callback
h1 = figure;
set(h1,'Visible','on')
% Specify the function that will be executed upon a key press event
```

Note: since the call back is associated with a figure, when using a LiveScript file, the visibility of the figure needs to be set to `'on'`

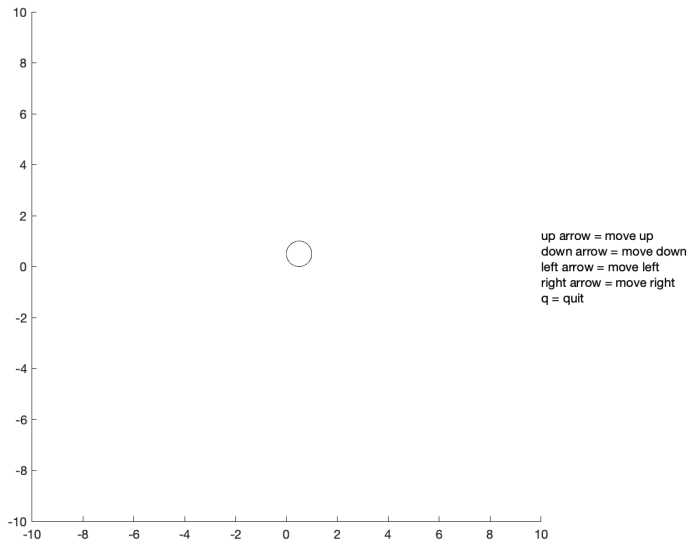
Example 2. Moving a ball

In the following, example, we can use the `arrow keys` to move a ball across the screen, while `'q'` closes the figure window. The following callback function is located at the end of the LiveScript document.

```
function DoWhenKeyIsPressed(~,eventdata,hBall)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.

    % Movement increment
    inc = 0.3;

    % Move up
```

```
    if strcmp(eventdata.Key,'uparrow')
        hBall.Position(2) = hBall.Position(2)+inc;
    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hBall.Position(2) = hBall.Position(2)-inc;
    % Move left
    elseif strcmp(eventdata.Key,'leftarrow')
        hBall.Position(1) = hBall.Position(1)-inc;
    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        hBall.Position(1) = hBall.Position(1)+inc;
    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end
end
```

```
% Initialize the figure and assign the keyboard callback
% function.
hFig = figure('Position',[200,200,1000,500]);

% Specify plot formatting
text(10,0,{'up arrow = move up','down arrow = move down', ...
    'left arrow = move left','right arrow = move right','q = quit'})
axis square
xlim([-10,10])
ylim([-10,10])

% Create a "ball" and place it at [0,0] initially
hBall = rectangle('Position',[0,0,1,1],'Curvature',[1,1]);

% Assign a keyboard call back
hFig.KeyPressFcn = {@DoWhenKeyIsPressed,hBall};

% Set figure to visible -> have it pop out so we can input the keyboard
% call backs
set(hFig,'Visible','on')
```



TOPIC 20: Simulations and Callbacks (cont'd)

ENGR105, 11/10/20

Revisiting simulations

We have already created and ran simulations in this class for a few different applications. Namely, we have simulated:

- The *number of turns* to obtain certain face values on a set of dice.
- The *number of turns* to obtain a number coin heads/tails in a row.
- The *outcome* from many interactions between **two individuals**, where an individual can act in cooperative, hostile, or more sophisticated manner.

As in these cases, **simulations** enable us to **study how a system evolves over time** and to **provide insight into possible outcomes**. Simulations are often simplified representations of something complex, and it is up to the scientist and engineer to determine appropriate rules, initial conditions, and boundary conditions.

Practice problem: simulating an infectious disease

Many simulations have been performed over the last year to describe viral transmission within a population. One recent simulation by the Washington Post modeled the transmission of SARS-CoV-2 amongst a population of *moving, interacting particles*:

https://www.washingtonpost.com/graphics/2020/world/corona-simulator/

Even in this *highly-simplified system*, the beneficial effects of **quarantining and social distancing** -- introducing "barriers" to social interactions resulted in fewer "super-spreader" events and the *flattening of the curve* of infections over time.

Create a program that models the spread of an infectious disease between moving, interacting particles. Your simulation should have the following rules:

1. It should consider **two particles**: one *"healthy"* particle and one *"infected"* particle.
2. Each particle can be represented *visually* as a **circle with a diameter of 1.**
3. Each particle should move in **steps**, where in a given step each particle may change x position by a value randomly distributed on the interval `[-0.5,0.5]`.
4. Each particle **may only move in the `+/-` x direction**. That is, the particle motion is 1-D.
5. The particle motion is confined to the domain `[0,7]`: that is, any "part" of each particle may not move to an x value less than `0` or an x value greater than `7`.
6. **The infected particle remains infected for the entire simulation**. The healthy particle remains healthy until the particles have "interacted," *i.e.* when the distance between particles is less than one particle diameter.

Question. By simulating ***many*** "sets" of particles, what are the **effects of social distancing**?

PRO, Step 1: Plan, **P**repare, and **P**seudocode

Helpful tasks and thoughts:

- What do we want to create? A simulation of moving particles, where we record when the particles come within one particle diameter of each other. This is when the "infection" occurs.
- What is our ultimate goal? Run many simulations with and without "social distancing." How can we implement social distancing? One idea would be to have **one particle remain stationary** while the other can move.
- How can we break this up into manageable chunks?

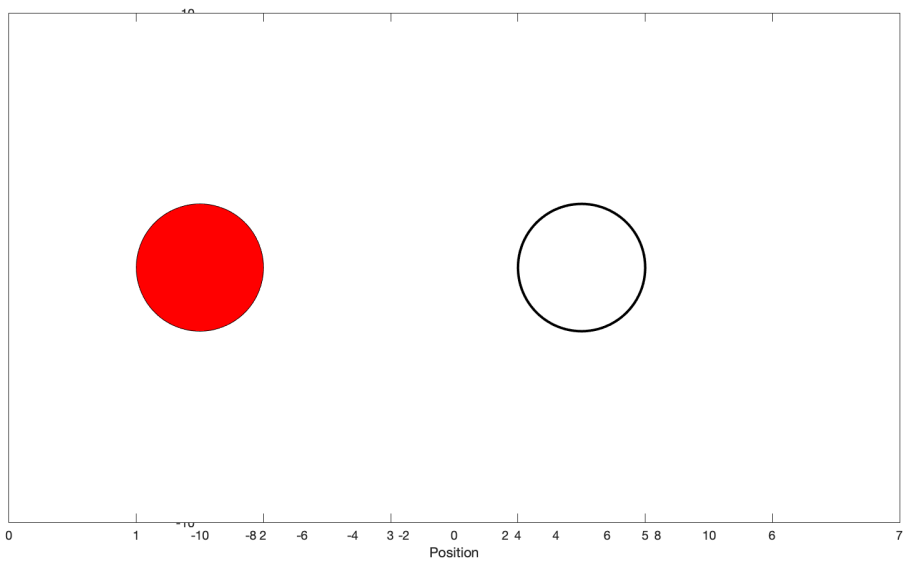
Remember: when faced with a large task like this, ***start small and gradually build up your solution***. Frequently check your output along the way. In this case, we will stop at defined checkpoints.

PRO, Step 2: w**R**ite **R**ough (increments of code) and **R**efine

Checkpoint goal 1: Produce a visualization of both particles.

```
clear
%1) Vector representing the positions of the particles

pos = [1,4];

% particle one would be infected
```

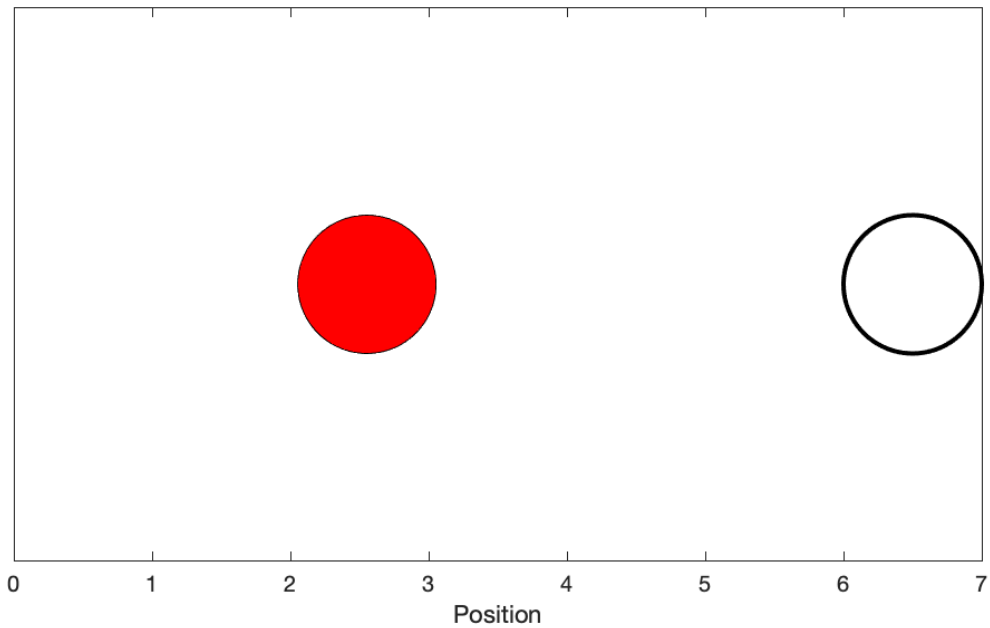
```
%particle 2 is healthy

%3)produce visualization of the particles
g1 = axes;
axis image
xlim([0,7])
ylim([-2,2])
set(g1, 'ytick', [])
set(g1, 'Box', 'on')
xlabel('Position')

% Plot both particles initial positions
p1 = rectangle('Position',[pos(1),-0.5,1,1],'Curvature',[1,1],'FaceColor','r');
p2 = rectangle('Position',[pos(2),-
0.5,1,1],'Curvature',[1,1],'FaceColor','w','LineWidth',2);
```



Checkpoint goal 2: Produce a visualization of particles moving.

```
close all
%1) Vector representing the positions of the particles

pos = [1,4];

% particle one would be infected
%particle 2 is healthy
```

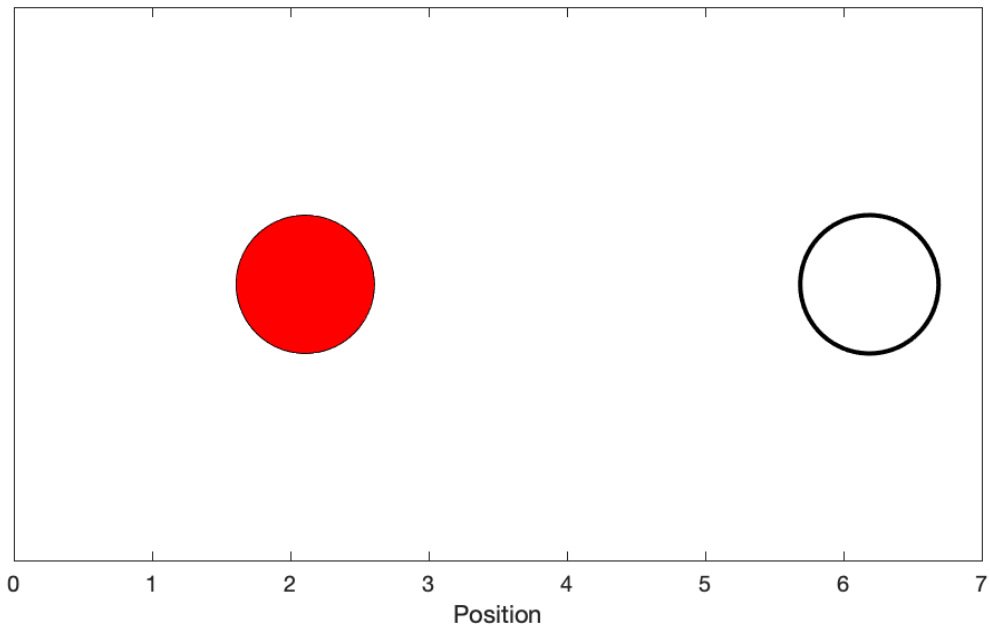
```
%3)produce visualization of the particles
g1 = axes;
axis image
xlim([0,7])
ylim([-2,2])
set(g1, 'ytick', [])
set(g1, 'Box', 'on')
xlabel('Position')

% Plot both particles initial positions
p1 = rectangle('Position',[pos(1),-0.5,1,1],'Curvature',[1,1],'FaceColor','r');
p2 = rectangle('Position',[pos(2),-
0.5,1,1],'Curvature',[1,1],'FaceColor','w','LineWidth',2);

% Step throught he simulation. In each step, update the particles positions
for k = 1:50

    % A) Create vector determining particle step
    d_k = (rand(1,2)-0.5);

    % B) Update particle position
    pos = pos + d_k;

    % C) Ensure particle doesn't move out of bounds
    pos = max(min(pos,6),0);

    % D) Update visualization
    p1.Position(1) = pos(1);
    p2.Position(1) = pos(2);

    % E) Pause Matlab
    pause(0.2)
end
```



Checkpoint goal 3: Incorporate interactions. Change healthy particle color when infection occurs

```
close all
%1) Vector representing the positions of the particles

pos = [1,4];

% particle one would be infected
%particle 2 is healthy

%3)produce visualization of the particles
g1 = axes;
axis image
xlim([0,7])
ylim([-2,2])
set(g1, 'ytick', [])
set(g1, 'Box', 'on')
```

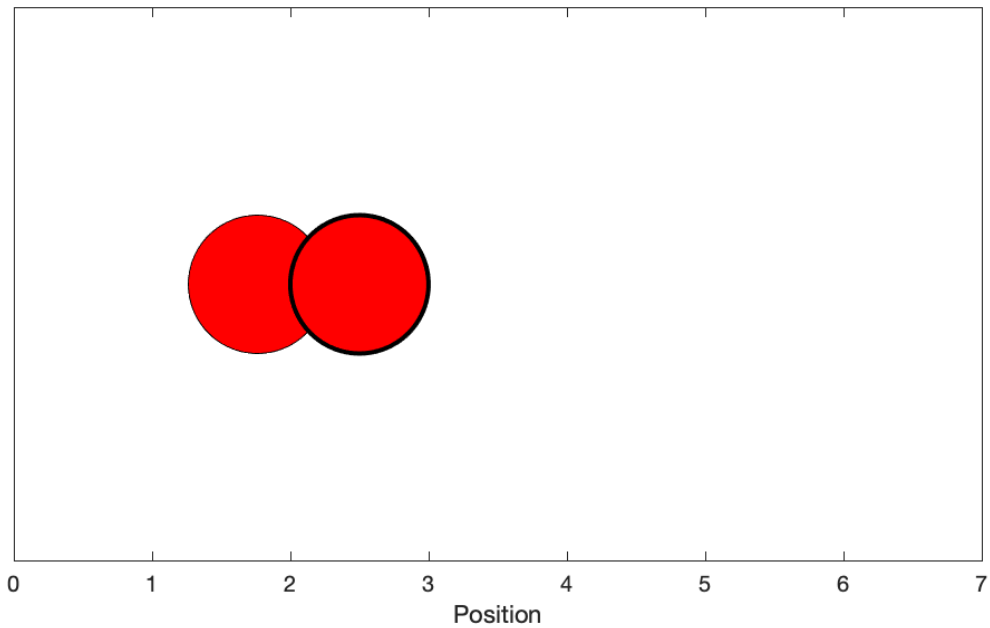
```
xlabel('Position')

% Plot both particles initial positions
p1 = rectangle('Position',[pos(1),-0.5,1,1],'Curvature',[1,1],'FaceColor','r');
p2 = rectangle('Position',[pos(2),-
0.5,1,1],'Curvature',[1,1],'FaceColor','w','LineWidth',2);

% Step throught he simulation. In each step, update the particles positions
for k = 1:50

    % A) Create vector determining particle step
    d_k = (rand(1,2)-0.5);

    % B) Update particle position
    pos = pos + d_k;

    % C) Ensure particle doesn't move out of bounds
    pos = max(min(pos,6),0);

    % Check if the particles interact
    if abs(pos(1) - pos(2)) < 1
        p2.FaceColor = 'r';
    end

    % D) Update visualization
    p1.Position(1) = pos(1);
    p2.Position(1) = pos(2);

    % E) Pause Matlab
    pause(0.2)
end
```



Checkpoint goal 4: Make the simulation run *until* the particles interact.

```
close all

% 1) vector representing positions of two particles
pos = [1,4];

% 2) Define particle identities.
% Particle 1 will be infected
% Particle 2 will intiially be healthy

% 3) Produce visualization of each particle
g1 = axes;
axis image
xlim([0,7])
ylim([-2,2])
set(g1, 'ytick', [])
set(g1, 'Box', 'on')
```

```
xlabel('Position')


% Plot both particles initial positions
p1 = rectangle('Position',[pos(1),-0.5,1,1],'Curvature',[1,1],'FaceColor','r');
p2 = rectangle('Position',[pos(2),-
0.5,1,1],'Curvature',[1,1],'FaceColor','w','LineWidth',2);


% Step through the simulation. In each step, update the particles positions
while abs(pos(1) - pos(2)) >=1
    % A) Create vector determining particle step
    d_k = (rand(1,2)-0.5);

    % B) Update particle position
    pos = pos + d_k;

    % C) Ensure particle doesn't move out of bounds
    pos = max(min(pos,6),0);

    % Check if the particles interact
    if abs(pos(1) - pos(2)) < 1
        p2.FaceColor = 'r';
    end
    % D) Update visualization
    p1.Position(1) = pos(1);
    p2.Position(1) = pos(2);

    % E) Pause Matlab
    pause(0.2)
end
```



Checkpoint goal 5: We've produced a visualization of particle simulation. Ultimately, this visualization is not important. Since we will eventually compare normal particles to "socially distancing particles," we really just want to know the time required until the infection occurs.

```
close all

% 1) vector representing positions of two particles
pos = [1,4];

% 2) Define particle identities.
% Particle 1 will be infected
% Particle 2 will intiially be healthy

% Number of time steps until infection occurs
numSteps = 0;

% Step through the simulation. In each step, update the particles positions
```

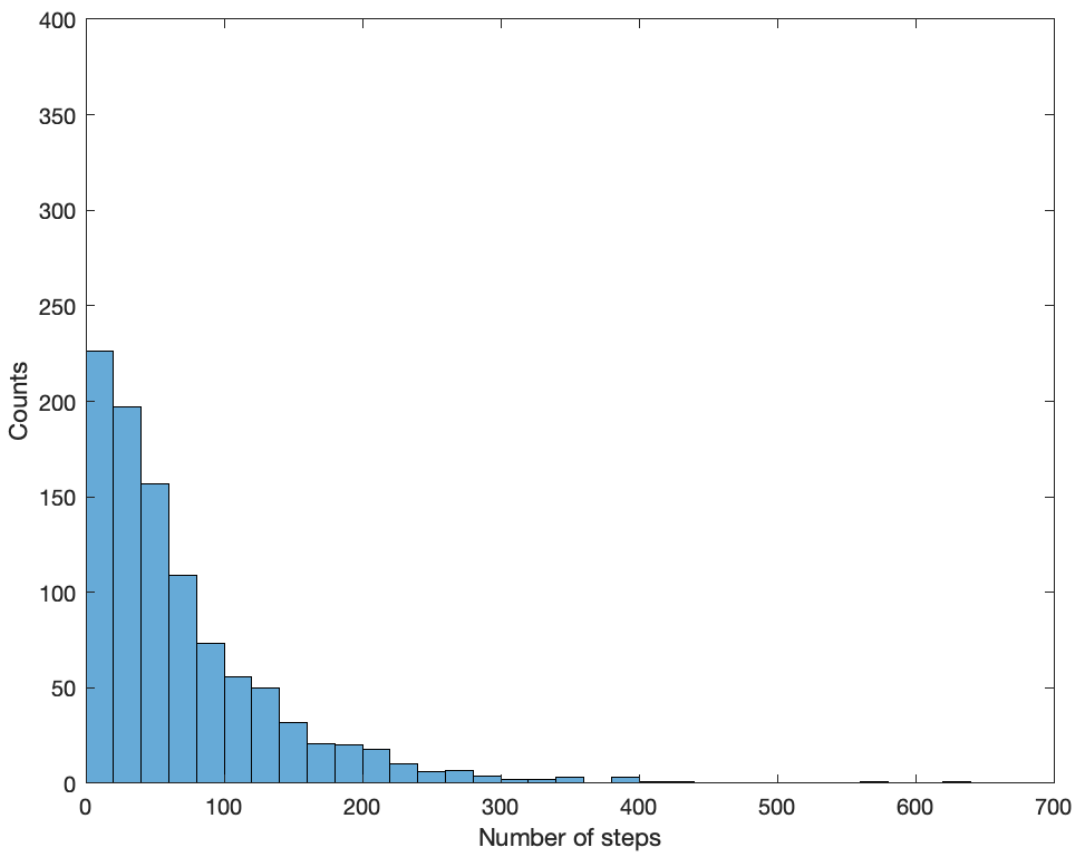
```
% Step through the simulation. In each step, update the particles positions
while abs(pos(1) - pos(2)) >=1
    % A) Create vector determining particle step
    d_k = (rand(1,2)-0.5);

    % B) Update particle position
    pos = pos + d_k;

    % C) Ensure particle doesn't move out of bounds
    pos = max(min(pos,6),0);

    % D) Update Counter
    numSteps = numSteps + 1;
end

disp(numSteps)
```

```
108
```

Checkpoint goal 6: Adapt the code so that we can observe the distribution of times until an infection occurs when 100 sets of simulations are performed.

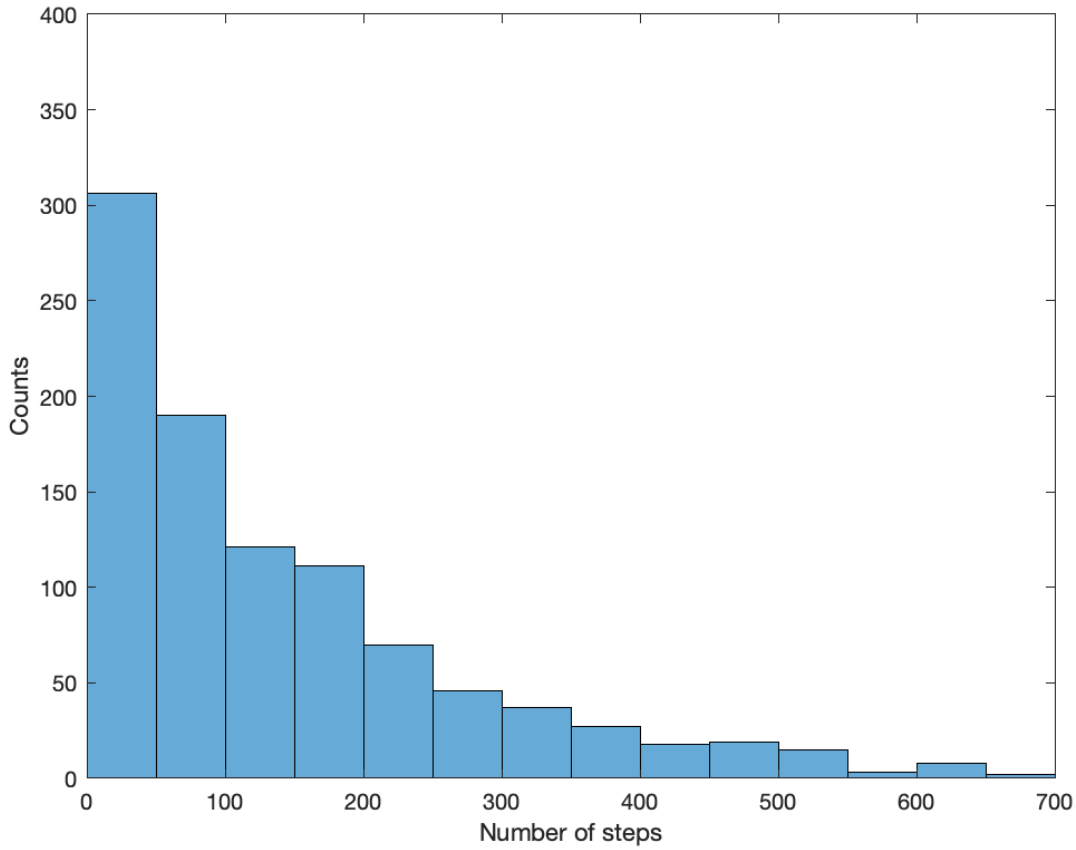
```
% Number of simulations
numSim = 1000;

% Preallocate storage vector
numSteps = zeros(1,numSim);

% Repeat simulation numSim times
for k = 1:numSim
    %1) Vector representing positions of two particles
    pos = [1,4];

    % 2) Define particle identities.
    % Particle 1 will be infected
    % Particle 2 will intiially be healthy

    % Step through the simulation. In each step, update the particles positions
    while abs(pos(1) - pos(2)) >=1
        % A) Create vector determining particle step
        d_k = (rand(1,2)-0.5);

        % B) Update particle position
```

```
        pos = pos + d_k;

        % C) Ensure particle doesn't move out of bounds
        pos = max(min(pos,6),0);

        % D) Update Counter
        numSteps(k) = numSteps(k) + 1;
    end

end

% Create visualization
histogram(numSteps)
xlabel('Number of steps')
ylabel('Counts')
xlim([0,700])
ylim([0,400])
```



Checkpoint goal 7: Adapt the code so that we can observe the distribution for if one particle is **social distancing**. That is, assume the infected particle does not move in this simulation

```
% Number of simulations
numSim = 1000;

% Preallocate storage vector
numSteps = zeros(1,numSim);

% Repeat simulation numSim times
for k = 1:numSim
    %1) Vector representing positions of two particles
    pos = [1,4];

    % 2) Define particle identities.
    % Particle 1 will be infected
    % Particle 2 will intiially be healthy

    % Step through the simulation. In each step, update the particles positions
    while abs(pos(1) - pos(2)) >=1
        % A) Create vector determining particle step
        d_k = (rand(1,2)-0.5).*[0,1];

        % B) Update particle position
        pos = pos + d_k;

        % C) Ensure particle doesn't move out of bounds
        pos = max(min(pos,6),0);

        % D) Update Counter
        numSteps(k) = numSteps(k) + 1;
    end

end

% Create visualization
histogram(numSteps)
xlabel('Number of steps')
ylabel('Counts')
xlim([0,700])
ylim([0,400])
```

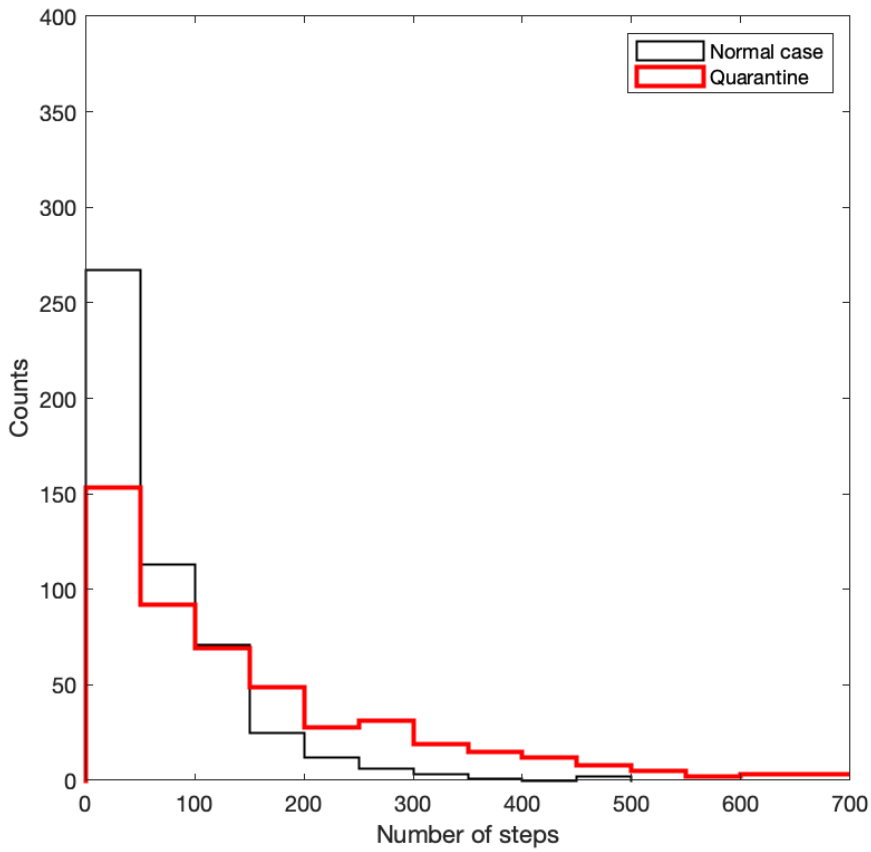


Checkpoint goal 8: Consolidate the above into one piece of code for comparing social distancing.

Create a relevant function:

```
% Number of simulations
numSim = 500;

% Preallocate storage vector
numSteps_norm = zeros(1,numSim);
numSteps_quar = zeros(1,numSim);

% Repeat simulation numSim times
```

```
for k = 1:numSim
    % Simulate normal
    numSteps_norm(k) = simulateDisease([1,1]);
    % Simulate quarantine
    numSteps_quar(k) = simulateDisease([0,1]);
end

% create visualization
h1 = histogram(numSteps_norm);
hold on
h2 = histogram(numSteps_quar);
hold off


% Visualization
xlabel('Number of steps')
ylabel('Counts')
h1.BinWidth = 50;
h2.BinWidth = 50;
h1.DisplayStyle = 'stairs';
h2.DisplayStyle = 'stairs';
h2.LineWidth = 2;
h1.LineWidth = 1;
h2.EdgeColor = 'r';
h1.EdgeColor = 'k';
xlim([0,700])
ylim([0,400])
axis square
legend('Normal case', 'Quarantine', 'Location', 'Northeast')
```



<u>Takeaways</u>:

- We have thus simulated how group dynamics can influence the spread of an infectious disease!
- How does this relate to what we might expect?
- When approaching problems, **start small and work up to a solution**.

TOPIC 21: Callbacks and simple interfaces

ENGR105, 11/12/20

Miscellaneous information: changing the directory and accessing all files in the current folder

- The **current directory** can be retrieved using `cd`. This will save the directory as a character array.

```
currDir = cd;
disp(currDir)
```

```
/Users/raribidesi/Desktop/ENGR 105/Class Notes/Final Participation
```

- You can use `cd` with a character input to navigate to a **different directory**

```
cd(currDir)
```

- `dir` retrieves the list of files in a given directory. If assigned to a variable, it will store the information as a structure array, where each structure corresponds to one file.

```
dirFiles = dir;
```

- You can **retrieve a list of file names** from the name field. This can be converted to a cell array using `{}`

```
files = {dirFiles.name};
size(files)
```

```
ans = 1×2
     1    21
```

```
files{5}
```

```
ans = 'L18_TrackingData.txt'
```

Note: this enables automated processing of a large number of files in a given directory.

Reminder: keyboard callbacks (key press)

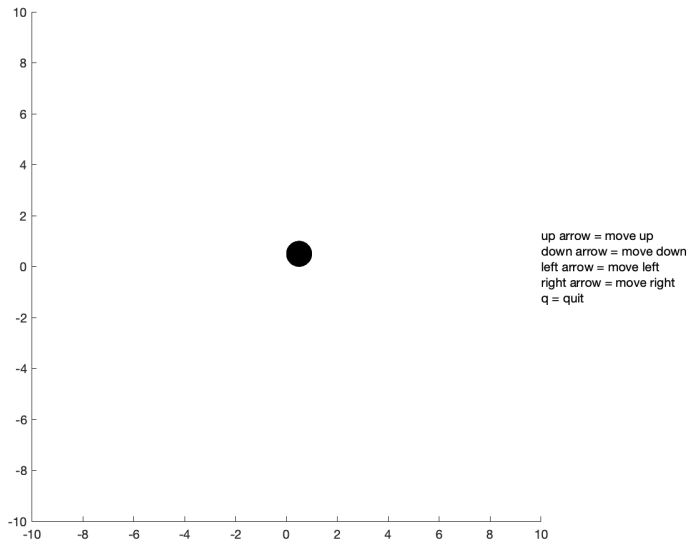
In Lecture 20/Topic 19, we produced an example where we can use the `arrow keys` to move a ball across the screen, while `'q'` closes the figure window. The following callback function is located at the end of the LiveScript document.

Example 1: Moving ball

```
function DoWhenKeyIsPressed(~,eventdata,hBall)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.

    % Movement increment
    inc = 0.3;

    % Move up
    if strcmp(eventdata.Key,'uparrow')
        hBall.Position(2) = hBall.Position(2)+inc;
    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hBall.Position(2) = hBall.Position(2)-inc;
    % Move left
    elseif strcmp(eventdata.Key,'leftarrow')
```

```
        hBall.Position(1) = hBall.Position(1)-inc;
    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        hBall.Position(1) = hBall.Position(1)+inc;
    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end
end
```

```
% Initialize the figure and assign the keyboard callback
% function.
hFig = figure('Position',[200,200,1000,500]);

% Specify plot formatting
text(10,0,{'up arrow = move up','down arrow = move down', ...
    'left arrow = move left','right arrow = move right','q = quit'})
axis square
xlim([-10,10])
ylim([-10,10])

% Create a "ball" and place it at [0,0] initially
hBall = rectangle('Position',[0,0,1,1],'Curvature',[1,1]);
hBall.FaceColor = 'k';

% Assign a keyboard call back
hFig.KeyPressFcn = {@Topic21DoWhenKeyIsPressed,hBall};

% Set figure to visible -> have it pop out so we can input the keyboard
% call backs
set(hFig,'Visible','on')
```



<u>Note</u>: callback functions do not support outputs.

Keyboard callbacks (key release)

In the above example, something happens when we *press a button*. We can have something happen when we release a button. This can be specified using the `'KeyReleaseFcn'` field of the graphics object for the current figure.

Example 2. Produce a game where you must keep moving the ball. If the player doesn't move for ~ 1 second, display "`Game Over`." Furthermore, make the "ball" become "whiter" as the game approaches the end game scenario.

To create this solution, we must create:

- A function to be assigned to the *key press*

```
function ex3_keyPress(~,eventdata,hBall)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.

    % Movement increment
    inc = 0.3;

    % Reset FaceColor to be black
    hBall.FaceColor = [0,0,0];

    % Move up
    if strcmp(eventdata.Key,'uparrow')
        hBall.Position(2) = min(hBall.Position(2)+inc,9);
    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hBall.Position(2) = max(hBall.Position(2)-inc,-10);
    % Move left
```

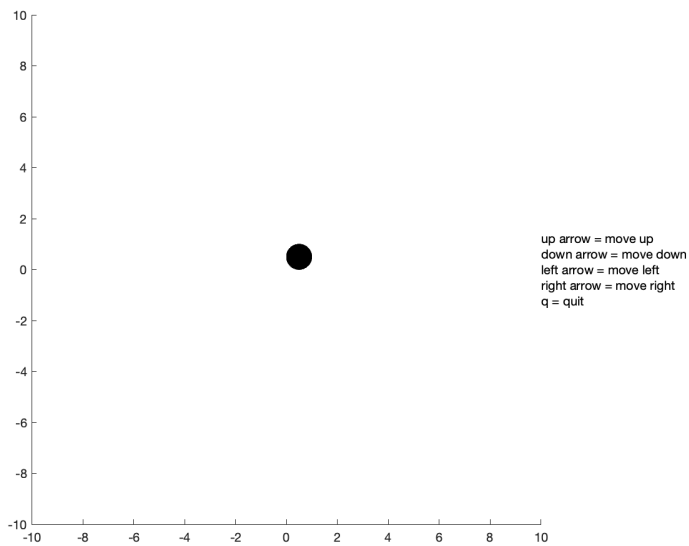
```
    elseif strcmp(eventdata.Key,'leftarrow')
        hBall.Position(1) = max(hBall.Position(1)-inc,-10);
    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        hBall.Position(1) = min(hBall.Position(1)+inc,9);
    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end
end
```

- A function to be assigned to the *key release*

```
function ex3_keyRelease(~,~,hBall)
    % Function to be executed when key is released. Takes a graphics
object
    % corresponding to the ball as an input

    % Increase the whiteness of the ball until it becomes fully white
    while hBall.FaceColor(1) < 0.9

        % Make FaceColor whiter
        hBall.FaceColor = min(hBall.FaceColor + 0.1,1);

        % Pause for 100 ms
        pause(0.1)
    end

    % If ~1 second passes without moving, game over
    t1 = text(-10,0,'GAME OVER');
    t1.FontSize = 50;
    t1.Color = 'r';

end
```

```
close all % Close previous windows

% Initialize the figure and assign the keyboard callback
% function.
hFig = figure('Position',[200,200,1000,500]);

% Specify plot formatting
text(10,0,{'up arrow = move up','down arrow = move down', ...
    'left arrow = move left','right arrow = move right','q = quit'})
axis square
xlim([-10,10])
ylim([-10,10])
```

```
% Create a "ball" and place it at [0,0] initially
hBall = rectangle('Position',[0,0,1,1],'Curvature',[1,1]);
hBall.FaceColor = 'k';  % Default color is black

% Assign a keyboard call back for when a buttom is pressed
hFig.KeyPressFcn = {@ex3_keyPress,hBall};
% Assign a keyboard call back for when a button is release
hFig.KeyReleaseFcn = {@ex3_keyRealease,hBall};

% Set figure to visible -> have it pop out so we can input the keyboard
% call backs
set(hFig,'Visible','on')
```



Challenge. Can you create modifications to the code so that:

- The radius of the ball resets on a key press but increases as the color becomes whiter?
- You want the ball to avoid regions of the playable area?
- At the end ot the game, the game window closes and the new window appears that says "Game Over?" Or the user cannot further move the ball.

<u>Takeaway note:</u> *in these cases (and below), callback functions are not compatible with outputs.*

Mouse callbacks

Similarly to assigning keyboard callbacks, one can assign callback functions to **mouse operations** as function handles to the appropriate field of a **graphics handle object**. In particular, we will discuss the following fields associated with mouse operations:

- `WindowButtonMotionFcn` - the mouse pointer is moved
- `WindowButtonDownFcn` - the mouse button is pressed
- `WindowButtonUpFcn` - the mouse button is released

Example 3. Create a script that tracks the mouse cursor across a figure windows and reports the x and y position at the tip of the cursor.

Callback function:

```
function moveText(~,~,g1)
    % The mouse button motion callback.

    % Get the current mouse location.
    pos = get (gca, 'CurrentPoint');
    x = pos(1,1); y = pos(1,2);

    % Update the text object readout to the current location
    % and place its location at the cursor tip.
    g1.String = ['x = ',num2str(x),', y = ',num2str(y)];
    g1.Position = [x, y, 0]; % position 0 in 3rd dimension
end
```

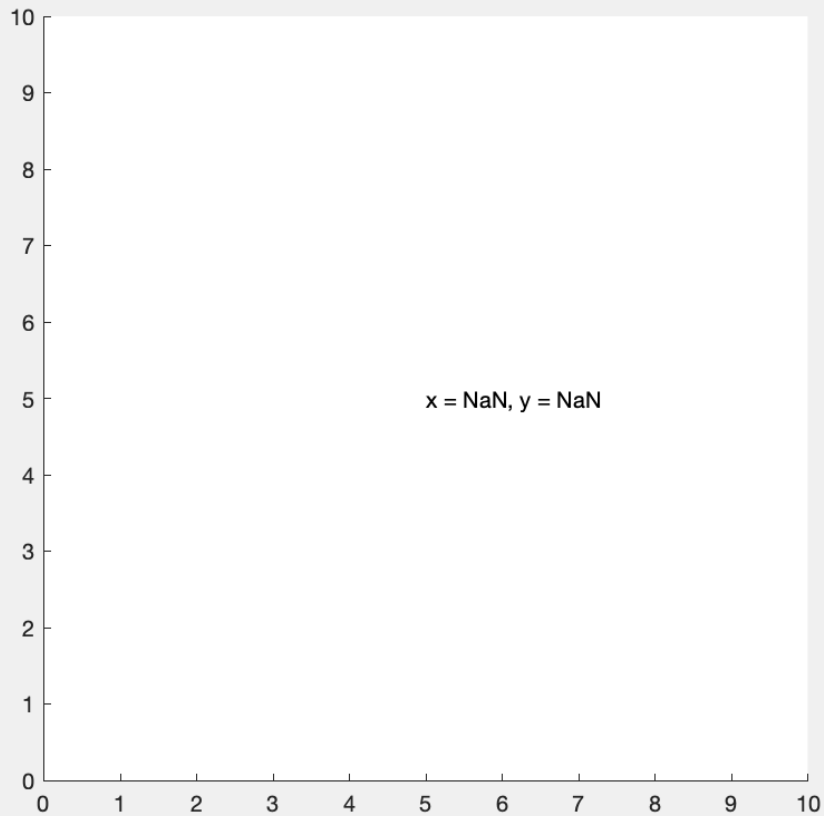
This will utilize a callback associated with the `'WindowButtonMotionFcn'` field, which **executes whenever motion on the figure is detected**.

<u>Note</u>: this utilizes `get(gca,'CurrentPoint')` in the callback function, which reports the location of the mouse button as a `2x3` matrix:

- The 1st row, 1st column of the matrix is the x position of the click
- The 1st row, 2nd column of the matrix is the y position of the click

Incorporating this into a script:

```
close all % close previous figures

% Create a figure to be associated with the mouse callback
hFig1 = figure('Visible','on');
axis equal
xlim([0,10])
ylim([0,10])

% Initialize a text object for displaying coordinates
g = text(5,5,'x = NaN, y = NaN');
```




```
% Assign mouse callback function
hFig1.WindowButtonMotionFcn = {@moveText,g};
```

Mouse button clicks can be registered in a figure window using the `WindowButtonDownFcn` callback.

Example 4. In the following, you can observe the `SelectionType` for different mouse clicks.

```
function click(h,~)
    % The callback for a mouse button press. Print the mouse
    % button identifier to the command window.
    h.SelectionType
end
```

```
close all

% Create a figure and establish the callback associated with a
% mouse button press.
figure('WindowButtonDownFcn',@click,'Visible','on')
```

- `'normal'` - left mouse button click
- `'alt'` - right mouse button click
- `'extend'` - middle mouse button click
- `'open'` - double click

Example 5. Create an **interactive plot** where points can be added to the plot based on the location of the mouse cursor when pressing any mouse button.

Callback function:

```
function plotPoints1(~,~,h)
    % Call back function for updating a plot

    % Get the current mouse location
    pos = get(gca,'CurrentPoint');

    % Update the plot with the new position
    h.XData(end+1) = pos(1,1);
    h.YData(end+1) = pos(1,2);
    drawnow
end
```

Note: the above uses `drawnow`. `drawnow` is equivalent to `pause(`*<shortest time based on your computer's processor>*`)` and updates figures and processes any pending callbacks. Use this command if you modify graphics objects and want to see the updates on the screen immediately.

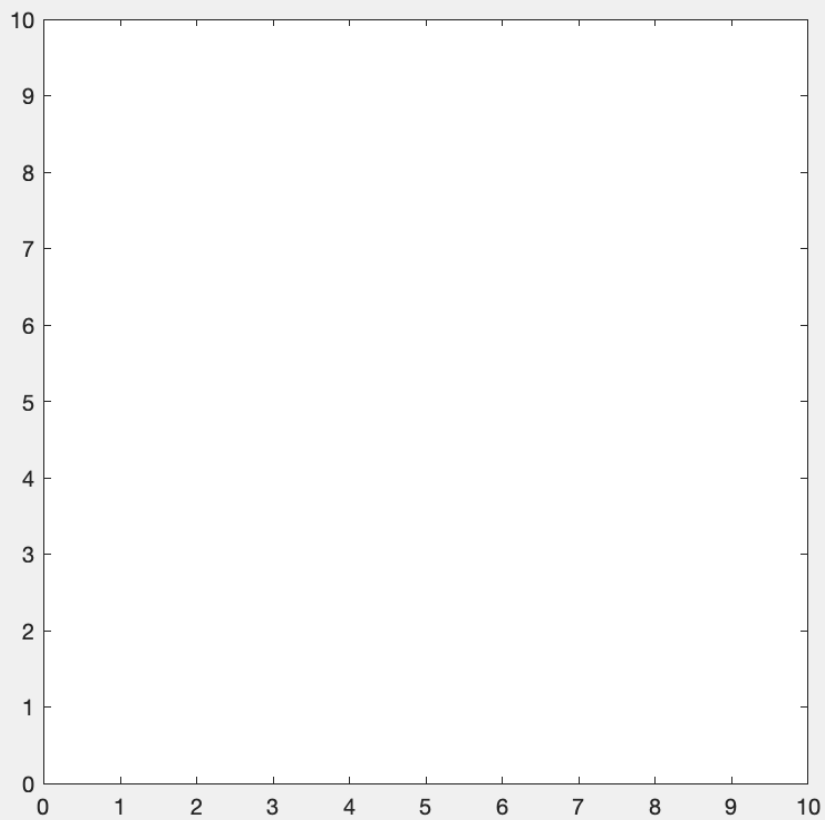
Incorporating this into a script:

```
close all

% Create a figure with mouse button press callback and plot
% NaNs (nothingness) on the figure.
hFig2 = figure('Visible','on');
h = plot(NaN,NaN,'ro-','MarkerFaceColor','r','MarkerSize',5);

% Assign callback function
hFig2.WindowButtonDownFcn = {@plotPoints1,h}
```

```
hFig2 =
  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [561 529 560 420]
       Units: 'pixels'

  Show all properties
```

```
% Graphical parameters
```

```
axis equal
xlim([0,10])
ylim([0,10])
```



Example 6. Create an similarly **interactive plot that** populates **lines of different colors** depending on if the left or right mouse button is clicked.

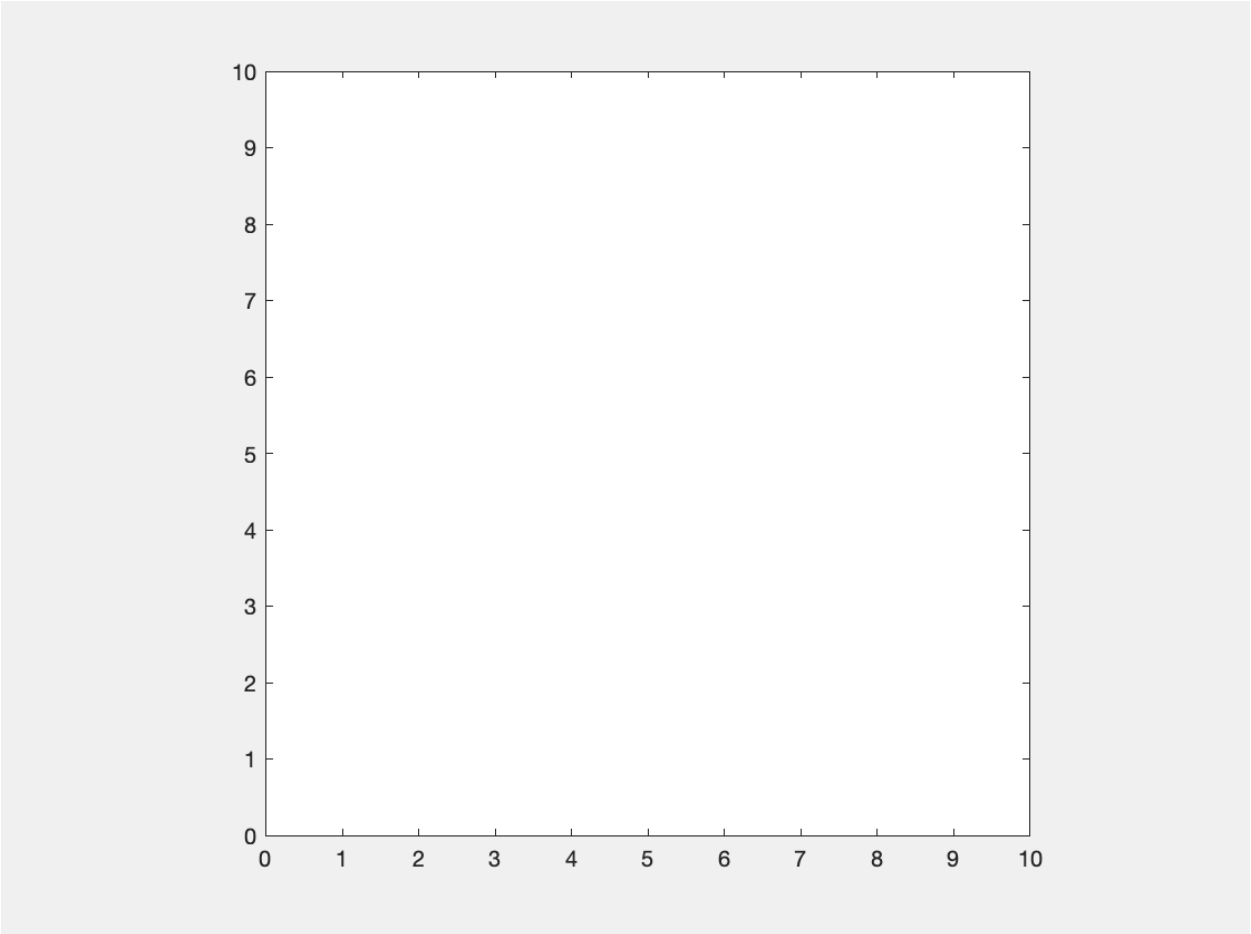
Callback function:

```
function plotPoints2(h,~,hRed,hGreen)
    % Callback function for updating a plot with multiple colored points

    % Get the current mouse location.
    pos = get(gca, 'CurrentPoint');

    % If a left click: add to the red plot
    if strcmp(h.SelectionType,'normal')

        % Add to the data in the red plot
        hRed.XData(end+1) = pos(1,1);
        hRed.YData(end+1) = pos(1,2);
        drawnow
```

```
    % If a right click: add to the green plot
    elseif strcmp(h.SelectionType,'alt')

        % Add to the data in the red plot
        hGreen.XData(end+1) = pos(1,1);
        hGreen.YData(end+1) = pos(1,2);
        drawnow

    end
end
```

Incorporating into a script:

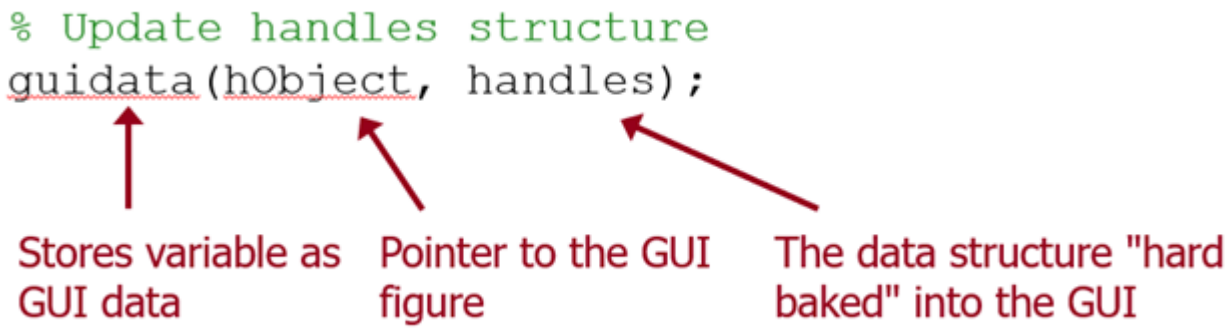
```
close all % Delete previous

% Create a figure with mouse button press callback
hFig3 = figure('Visible','on');

% Plot each color line, initially as NaN. The first plot (handle = hRed)
% hold the red points and the second plot (handle = hGreen) hold
% the green points.
hRed = plot(NaN,NaN,'ro-','MarkerFaceColor','r','MarkerSize',5);
hold on
hGreen = plot(NaN,NaN,'go-','MarkerFaceColor','g','MarkerSize',5);
hold off

% Assign callback function
hFig3.WindowButtonDownFcn = {@plotPoints2, hRed, hGreen};

axis square
xlim([0,10])
ylim([0,10])
```



Storing data in an interface with `guidata`

In the above example, we **add points** to one of two data sets. What if we want to ***remove points*** in reverse order? This is difficult because callback functions **<u>do not</u>** return outputs. That is, we cannot store the order of the points added.

- To address this issue, `guidata()` can be used to pass variables among callbacks in figures and interfaces.
- `guidata` is used to update a **structure** variable that is associated with a given figure or interface.
- The following depicts the syntax for `guidata()`. `handles` is a common representation for a data structure that can store information. The name comes from the application of guidata in creating graphical user interfaces using the built-in `guide` designer.



Note: guidata is used to both **store** and **retrieve** the data structure.

Example 7. For an interactive plot created in Example 6, add a functionality such that

Updated callback function:

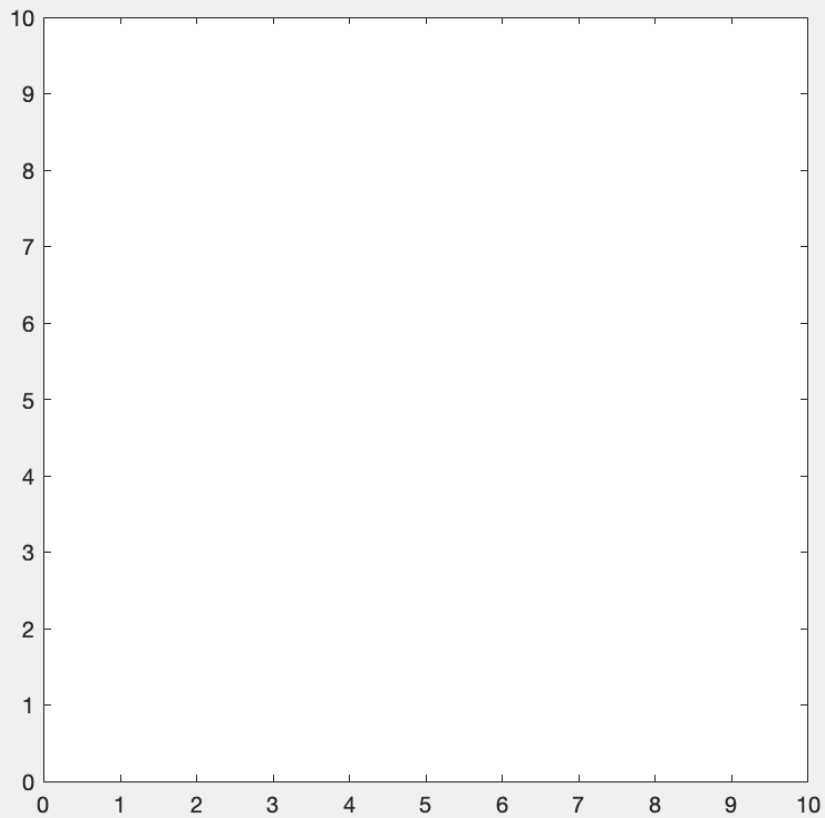
```
function plotPoints3(h,~,hRed,hGreen)
    % Callback function for updating a plot with multiple colored points

    % Get the current mouse location.
    pos = get(gca, 'CurrentPoint');

    % Load GUI information
    handles = guidata(h);

    % If a left click: add to the red plot
    if strcmp(h.SelectionType,'normal')

        % Add to the data in the red plot
        hRed.XData(end+1) = pos(1,1);
        hRed.YData(end+1) = pos(1,2);
        drawnow

        % Specify that the most recent point was red
        handles.order{end+1} = 'r';

    % If a right click: add to the green plot
    elseif strcmp(h.SelectionType,'alt')

        % Add to the data in the red plot
        hGreen.XData(end+1) = pos(1,1);
        hGreen.YData(end+1) = pos(1,2);
        drawnow

        % Specify that the most recent point was red
        handles.order{end+1} = 'g';

    % If a middle click (and note empty: delete the most recent point
    elseif strcmp(h.SelectionType,'extend') && ~isempty(handles.order)
```

```
        % If last point was red
        if strcmp(handles.order{end},'r')
            % Remove last data poitn
            hRed.XData(end) = [];
            hRed.YData(end) = [];
            drawnow

            % Delete last entry
            handles.order(end) = [];

        % If last point was green
        else
            % Remove last data poitn
            hGreen.XData(end) = [];
            hGreen.YData(end) = [];
            drawnow

            % Delete last entry
            handles.order(end) = [];

        end
    end

    % Update the guidata with the data structure
    guidata(h,handles);
end
```

Incorporating into a script:

```
close all % Delete previous

% Create a figure with mouse button press callback
hFig3 = figure('Visible','on');

% Create a data structure to store the order of moves
handles.order = cell(0);
% Update the guidata with that data structure
guidata(hFig3,handles);


% Plot each color line, initially as NaN. The first plot (handle = hRed)
% hold the red points and the second plot (handle = hGreen) hold
% the green points.
hRed = plot(NaN,NaN,'ro-','MarkerFaceColor','r','MarkerSize',5);
hold on
hGreen = plot(NaN,NaN,'go-','MarkerFaceColor','g','MarkerSize',5);
hold off

% Assign callback function
```

```
hFig3.WindowButtonDownFcn = {@lotPoints3, hRed, hGreen};

axis square
xlim([0,10])
ylim([0,10])
```



Example: Revisiting Tic Tac Toe

In lectures 9 - 11 and in Homework #6, we created an environment for playing Tic Tac Toe.

```
function ticTacToe(compStrategy)
    % Define initial variables and arrays
    gameboard = zeros(3,3); % gameboard that we will populate

    % Repeat: alternate for players
    player = 1; % player one

    % Logical variable to determine if game should stil be playing
    gameStillGoing = 1;

    % Iterate until someone has won (or the game is a draw)
    while gameStillGoing
```

```
        % Player 1: human player. If it is player 1's turn, specify a
        % position to be played
        if player == 1

            % Logical variable corresponding if the move is valid
            invalidMove = 1;

            % Make player pick a valid position
            while invalidMove

                % Position to play, in single index notation
                playPos = input('Select a position to play...');

                if ~isempty(playPos)
                    % Determine if move is allowable
                    invalidMove = all((gameboard(:) == 0)'.*(1:9) ~=
playPos);
                end
            end

        % Player 2: computer player. If it is player 2's turn, determine
the
        % position played based on the algorithm contained within
compStrategy().
        else
            % Call a function that will output a position based upon the
            % strategy defined for the computer
            playPos = compStrategy(gameboard);
        end

        % Update the gameboard
        gameboard(playPos) = player;

        % Step 3: display game board
        disp(gameboard)

        % Step 4: evaluate stopping condition -> did someone win?
        gameStillGoing = checkEndGame2(gameboard, player);

        % Step 5: change player
        player = mod(player,2)+1;
    end
end
```

```
% Play tic tac toe against AI player who plays randomly
ticTacToe(@playRandom)
```

```
     1     0     0
     0     0     0
     0     0     0

     1     0     0
     0     0     0
     0     0     2
```

```
     1     1     0
     0     0     0
     0     0     2

     1     1     0
     0     0     2
     0     0     2

     1     1     1
     0     0     2
     0     0     2

1 wins!
```

Goal: we want to create a user interface for playing Tic Tac Toe.

Our previous methodology:

1. Initialize settings and gameboard
2. Sequentially loop through turns as long as a end condition is not satisfied
3. If it is the player's turn, input an index number.
4. If it is the computer's turn, pick the next position according to some strategy.
5. Update gameboard and evaluate end game conditions.
6. Iterate as long as necessary

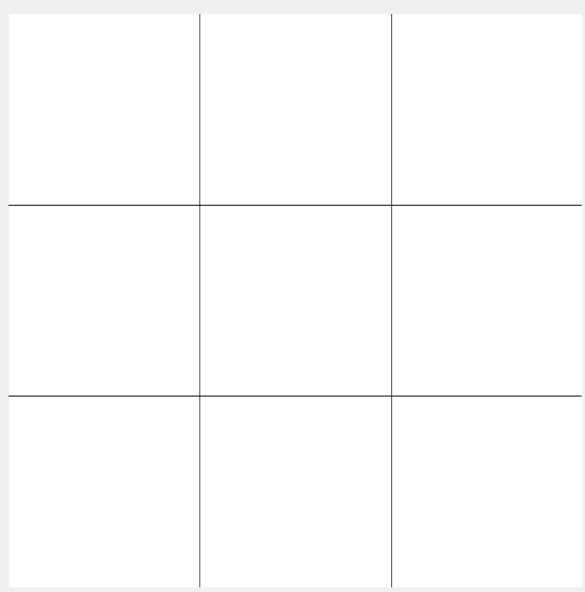
One possible new methodology:

1. Create a graphics object (a figure)
2. On mouse click: the player makes a move, the gameboard is updated, and end game conditions are evaluated
3. On mouse release: the computer will make a move, the gameboard is updated, and end game conditions are evaluated
4. Once end game conditions are satisfied, the game ends.

Question: what variable will need to be "updated" throughout all of this? The gameboard. For this reason, we should anticipate using guidata to store and retrieve the current gameboard.

```
% Create figure objects
h1 = figure('Visible','on');
g1 = axes;

% Plot grid lines
plot([-1,-1],[-4,4],'k-')
hold on
plot([1,1],[-4,4],'k-')
plot([-4,4],[1,1],'k-')
plot([-4,4],[-1,-1],'k-')
hold off

% Modify graphical parameters
```

```
 xlim([-3,3])
 ylim([-3,3])
 g1.XColor = 'none';
 g1.YColor = 'none';
 g1.Position(3:4) = 0.75*g1.Position(3:4);
 g1.OuterPosition(1) = 0.125;
 axis square

 % Create two empty text containers. On every turn, we will update the the
 % position of either the next X or next O marker from NaN to the
 % appropriate coordinates
 x1 =
text(NaN(5,1),NaN(5,1),'X','FontName','Arial','FontSize',50,'FontWeight','bold')
;
 x2 =
text(NaN(4,1),NaN(4,1),'O','FontName','Arial','FontSize',50,'FontWeight','bold')
;
```



```
 % Create data structure to store gameboard
```

```
handles.gameboard = zeros(3);
% Store data structure
guidata(h1,handles);

% Assign callback function for mouse click
h1.WindowButtonDownFcn = {@placeX,x1};
% Assign callback function for mouse release
h1.WindowButtonDownFcn = {@place0,x2,@playRandom}
```

```
h1 =
  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [561 529 560 420]
       Units: 'pixels'

  Show all properties
```

Creating figure UI objects using `uicontrol`

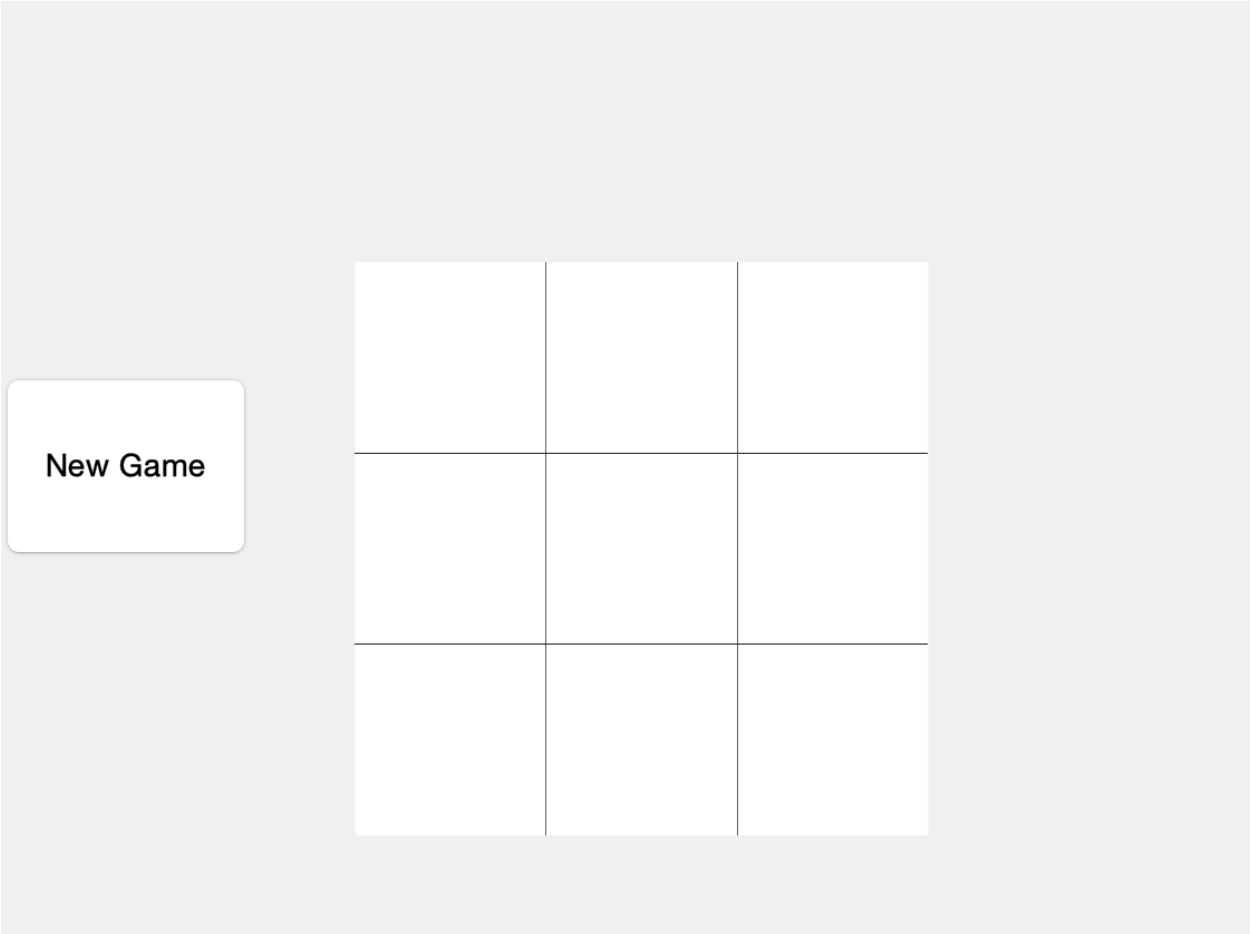
Interactive elements may be added directly to a figure using `uicontrol`. For a list of interactive objects, please see the MATLAB documentation. Briefly, the types of objects to be incorporated include:

Style Property Value	Example
'pushbutton'	Push Button
'togglebutton'	Toggle Button Toggle Button
'checkbox'	Check Box Check Box
'radiobutton'	Radio Button Radio Button
'edit'	Enter search term.
'text'	Select an item below:
'slider'	
'listbox'	Item 1 Item 2 Item 3
'popupmenu'	Item 1 Item 1 Item 2 Item 3

Example. Update the Tic Tac Toe code to include a button that resets the game.

```
% Create figure objects
h1 = figure('Visible','on');
g1 = axes;

% Plot grid lines
plot([-1,-1],[-4,4],'k-')
hold on
plot([1,1],[-4,4],'k-')
plot([-4,4],[1,1],'k-')
plot([-4,4],[-1,-1],'k-')
hold off

% Modify graphical parameters
xlim([-3,3])
ylim([-3,3])
g1.XColor = 'none';
g1.YColor = 'none';
g1.Position(3:4) = 0.75*g1.Position(3:4);
g1.OuterPosition(1) = 0.125;
axis square
```

```
 % Create two empty text containers. On every turn, we will update the the
 % position of either the next X or next O marker from NaN to the
 % appropriate coordinates
 x1 =
text(NaN(5,1),NaN(5,1),'X','FontName','Arial','FontSize',50,'FontWeight','bold')
;
 x2 =
text(NaN(4,1),NaN(4,1),'O','FontName','Arial','FontSize',50,'FontWeight','bold')
;

 % Create data structure to store gameboard
 handles.gameboard = zeros(3);
 % Store data structure
 guidata(h1,handles);

 % Assign callback function for mouse click
 h1.WindowButtonDownFcn = {@placeX,x1};
 % Assign callback function for mouse release
 h1.WindowButtonUpFcn = {@placeO,x2,@playRandom};

 % Pushbutton to subtract 1 from the current value of the counter
 u1 = uicontrol('Style','pushbutton','Units','Normalized','Position',...
     [0,0.4,0.2,0.2],'String','New Game');
 u1.Callback = {@resetGame,h1,x1,x2,@placeX,@placeO,@playRandom};
 u1.FontSize = 14;
```



TOPIC 22: Calculus in MATLAB

ENGR105, 11/12/20

Practice problem: diagnosing sickle cell disease

Sickle cell disease is an inherited red blood cell disorder that affects millions of people worldwide and approximately 1 in 3,000 Americans. Normally, red bloods cells are round and flexible: these can easily move through one's blood vessels. In sickle cell disease, however, red bloods are shaped like "sickles" and are rigid: these diseased cells can slow or block blood flow and oxygen to parts of the body, leading to pain, anemia, a compromised immune system, and other complications.

Sickle cell disease is caused by a *point mutation* in the gene for hemoglobin, the iron-rich compound that gives blood its red color and enables *red blood cells* to carry oxygen from your lungs to the rest of your body. A point mutation is when *one letter* of a nucleotide sequence ***differs from what is found in a "normal" or healthy sequence***. In this problem, we will consider the *mRNA sequence* of the hemoglobin gene. Recall from lecture that an **mRNA sequence is *just* a sequence of**

letters (text), where each position could be one of four potential letters: **A, U, C, or G**. Each letter corresponds to a different nucleotide, which in turn corresponds to a chemical structure in the mRNA molecule. This sequence of letters represents information about *how* the corresponding protein will be constructed.

Consider the **first 24 nucleotides of the mRNA sequence for the hemoglobin gene**, shown below. *Normally*, the 20th nucleotide is an "**A**," corresponding to the adenine nucleotide:

AUGGUGCACCUGACUCCUG**A**GGAG

In a person carrying the sickle cell trait (and therefore at risk for either having the disease or passing it on to their children), the 20th nucleotide is often instead a "**U**,", corresponding to the uracil nucleotide:

AUGGUGCACCUGACUCCUG**U**GGAG

Question. Produce a script that loads the data and evaluates how well the test works.

```
% Step 1: Read data from text file. We will use readcell
patientData = readcell('patientData.txt');

% Step 2: Prepare and preallocate necessary vectors
trueResult = zeros(size(patientData,1),1);  % Blood test result
diagResult = zeros(size(patientData,1),1);  % Diagnostic

% Step 3: Loop through each row (data point / patient) of the collected data
for k = 1:size(patientData,1)

    % Evaluate if patient has sickle cell disease based on sequence. Will
    % return 1 if they have sickle cell, 0 if they do not
    trueResult(k) = patientData{k,1}(20) == 'U';

    % Evaluate what the diagnostic predicts. Will return 1 if diagnostic
    % indicates sickle cell disease, 0 if it does not.
    diagResult(k) = strcmp(patientData{k,2},'sicklecell');

end

% Determine "accuracy" of diagnostic result
% accuracy1: simply finds proportion of predicted true relative to absolute
% true
accuracy1 = sum(diagResult(trueResult == 1))/length(trueResult(trueResult ==
1))
```

```
accuracy1 = 0.7302
```

```
% accuracy2 (better): determines the proportion of test results that the
% diagnostic accurately classified as having sickle cell or not
accuracy2 = sum(trueResult == diagResult) / length(trueResult)
```

```
accuracy2 = 0.8759
```

A simpler solution. A 3 line solution can be created using `cellfun`, which applies any specified function to each entry in a specified cell.

- *First input*: function handle
- *Second input*: cell array

```
% Determine patients that have the disease
trueResult = cellfun(@(y) y(20) == 'U', patientData(:,1));

% Result of diagnostic
diagResult = cellfun(@(x) strcmp(x,'sicklecell'),patientData(:,2));

% Determine accuracy, as above
accuracy1 = sum(diagResult(trueResult == 1))/length(trueResult(trueResult ==
1))
```

```
accuracy1 = 0.7302
```

```
accuracy2 = sum(trueResult == diagResult) / length(trueResult)
```

```
accuracy2 = 0.8759
```

Takeaway: always remember to consider vectorization. We've shown that this typically produces more efficient solutions.

Calculus in MATLAB

MATLAB can be used to tackle calculus problems (integration, differentiation, numerical evaluation of ordinary differential equations)

1-D Differentation

Many problems in engineering revolve around answering questions related to changes:

- What is the tendency of a function at some point?
- What is the slope of a surface or the normal to a surface?
- What is the incremental value of a cost function?

These inquiries may be answered using differentiation – numeric/discrete differentiation may be achieved using `diff()`

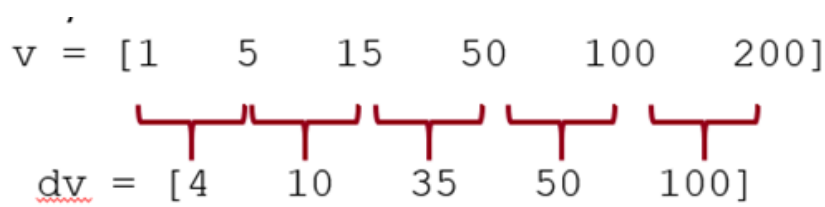
```
% Vector
v = [1,5,15,50,100,200]
```

```
v = 1×6
     1     5    15    50   100   200
```

```
% Numeric derivative
dv = diff(v)
```

```
dv = 1×5
     4    10    35    50   100
```

Note: dv contains 1 fewer element than v because it calculates the *difference* between entries.



Example: Usain Bolt and the 100 m

In the 2009 World Championships in Athletics, Usain Bolt ran a record 9.58s 100 meter sprint. The positional information for this run is contained in the file `L23_UB_data.xlsx`. Compute Usain Bolt's **velocity** and **acceleration** during this sprint.

Side note: note the file format. Microsoft Excel files can be loaded using `importdata()`. However, a simpler function for this specific file type is `xlsread`.

```
% Load the data from the excel file
[data,headers] = xlsread('L23_UB_data.xlsx');
```

The time and position data for this event can be read into MATLAB from a excel file and differentiated to obtain velocity and acceleration. Specifically we need to calculate:

- velocity

$$v = \frac{\Delta x}{\Delta t}$$

- acceleration

$$a = \frac{\Delta v}{\Delta t}$$

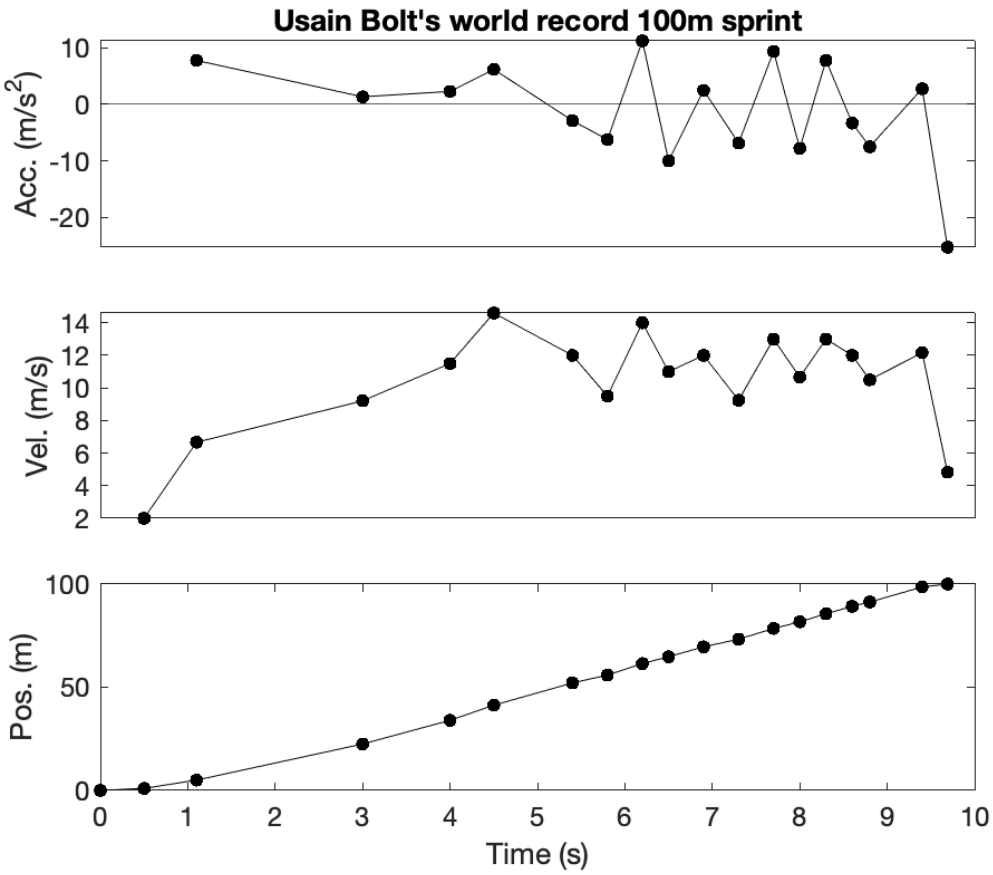
```
% 1) Place the data in time and position vectors
t = data(:,1);
```

```
pos = data(:,2);
% 2) Calculate the velocity and acceleration as a
% function of time
v = diff(pos)./diff(t);
a = diff(v)./diff(t(2:end));
```

Having calculated each parameter, we can plot. Note the following shows how you can create multiple axes in a single figure **without** using `subplot`.

```
close all
% Produce a plot depicting all results
fs = 12;    % Font size
ms = 16;    % Marker size

% First plot: position vs. time
g1 = axes;
g1.Position = [0.15,0.15,0.7,0.22];
plot(t,pos,'k.-','MarkerSize',ms);
xLim = xlim;
xlabel('Time (s)')
ylabel('Pos. (m)')
g1.FontSize = fs;

% Second plot: velocity vs. time
g2 = axes;
g2.Position = [0.15,0.44,0.7,0.22];
plot(t(2:end),v,'k.-','MarkerSize',ms);
ylabel('Vel. (m/s)')
axis tight
xlim(xLim)
g2.XTick = [];
g2.FontSize = fs;

% Third plot: acceleration vs. time
g3 = axes;
g3.Position = [0.15,0.73,0.7,0.22];
plot(t(3:end),a,'k.-','MarkerSize',ms);
hold on
plot(xLim,[0 0],'k-')
hold off
set(gca,'FontSize',fs)
ylabel('Acc. (m/s^2)')
axis tight
xlim(xLim)
g3.XTick = [];
g3.FontSize = fs;
```

```
title('Usain Bolt''s world record 100m sprint')
```



Note: if you use `diff` on a matrix, be sure to specify a dimension. You can also specify an "order" of the difference.

```
% Declare 2-D array
A = [4,5,6;3,2,1]
```

```
A = 2×3
     4     5     6
     3     2     1
```

```
% Difference over rows
diff(A,1,1)
```

```
ans = 1×3
    -1    -3    -5
```

```
% Difference over columns
diff(A,1,2)
```

```
ans = 2×2
     1     1
    -1    -1
```

Visualizing and creating multi-dimensional derivatives

"Vector fields" (and 3D data) may be represented using `meshgrid`. `meshgrid` creates matrices of numbers that have as many columns as `length(x)` and as many rows as `length(y)` with `x` repeated along rows and `y` repeated along columns `X` and `Y` represent the `x` and `y` values at each index.

```
% X values
x = -3:0.5:3;
% Y values
y = -2:0.5:2;
% Create meshgrid
[X,Y] = meshgrid(x,y)
```

```
X = 9×13
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0
0.5000    1.0000    1.5000    2.0000    2.5000    3.0000
Y = 9×13
   -2.0000   -2.0000   -2.0000   -2.0000   -2.0000   -2.0000   -2.0000
-2.0000   -2.0000   -2.0000   -2.0000   -2.0000   -2.0000
   -1.5000   -1.5000   -1.5000   -1.5000   -1.5000   -1.5000   -1.5000
-1.5000   -1.5000   -1.5000   -1.5000   -1.5000   -1.5000
   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000
-1.0000   -1.0000   -1.0000   -1.0000   -1.0000   -1.0000
   -0.5000   -0.5000   -0.5000   -0.5000   -0.5000   -0.5000   -0.5000
-0.5000   -0.5000   -0.5000   -0.5000   -0.5000   -0.5000
```

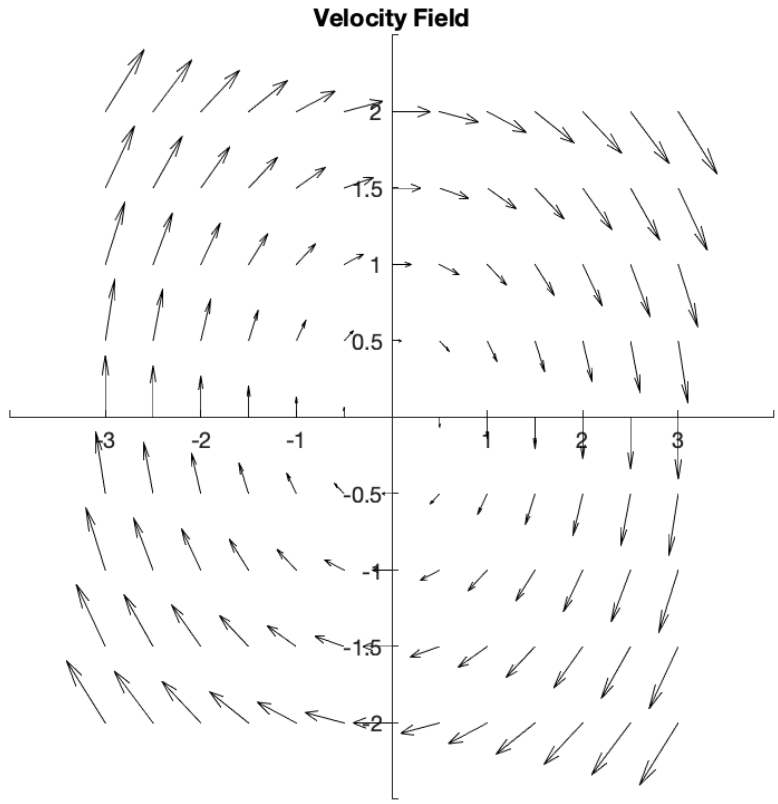
```
         0         0         0         0         0         0         0
 0         0         0         0         0         0
    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000    0.5000
 0.5000    0.5000    0.5000    0.5000    0.5000    0.5000
    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
 1.0000    1.0000    1.0000    1.0000    1.0000    1.0000
    1.5000    1.5000    1.5000    1.5000    1.5000    1.5000    1.5000
 1.5000    1.5000    1.5000    1.5000    1.5000    1.5000
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
 2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
```

Note the definitions of X and Y, above. X corresponds to rows, Y corresponds to columns.

Visualizing velocity fields: `quiver` plots

Using data in the above format, you can assign **x and y components of velocity**. Note that the velocity component in a given index location corresponds to the X and Y positional entries in the corresponding indices of X and Y.

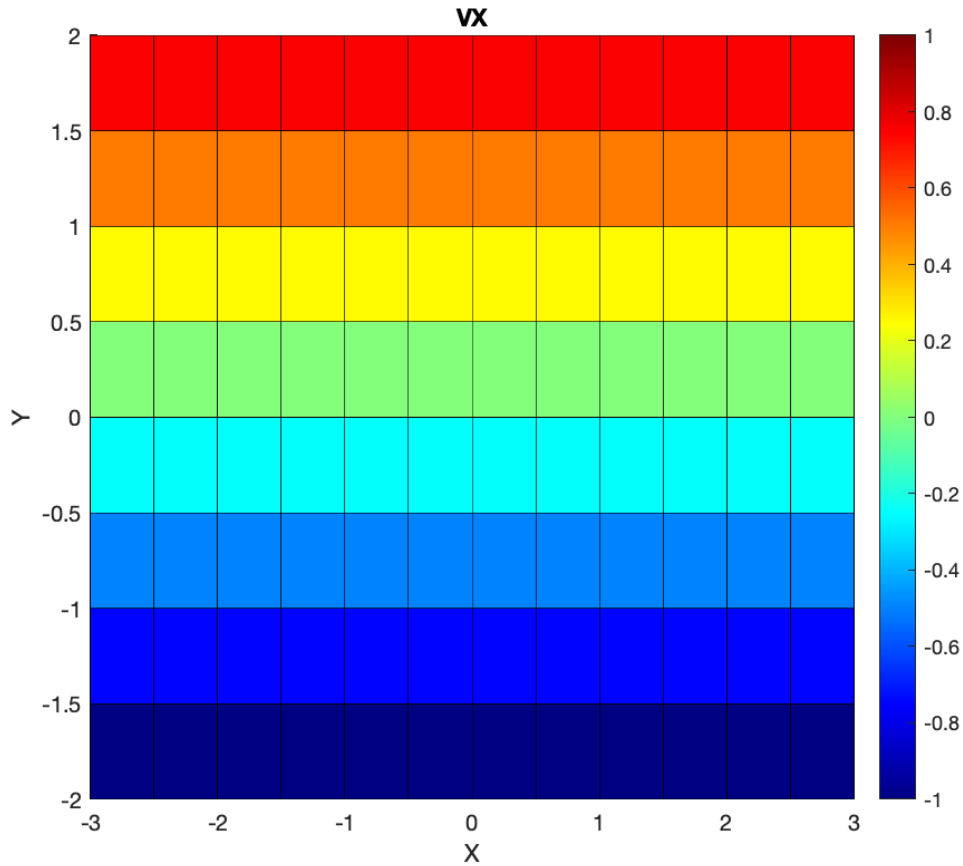
```
% Specify X component of velocity
VX = Y/max(Y,[],'all');
% Specify Y component of velocity
VY = -X/max(X,[],'all');
close all
% Visualize the velocity profile
q = quiver(X,Y,VX,VY);
q.Color = 'k';
set(gca,'YAxisLocation','Origin')
set(gca,'XAxisLocation','Origin')
set(gca,'Box','off')
axis square
title('Velocity Field')
```



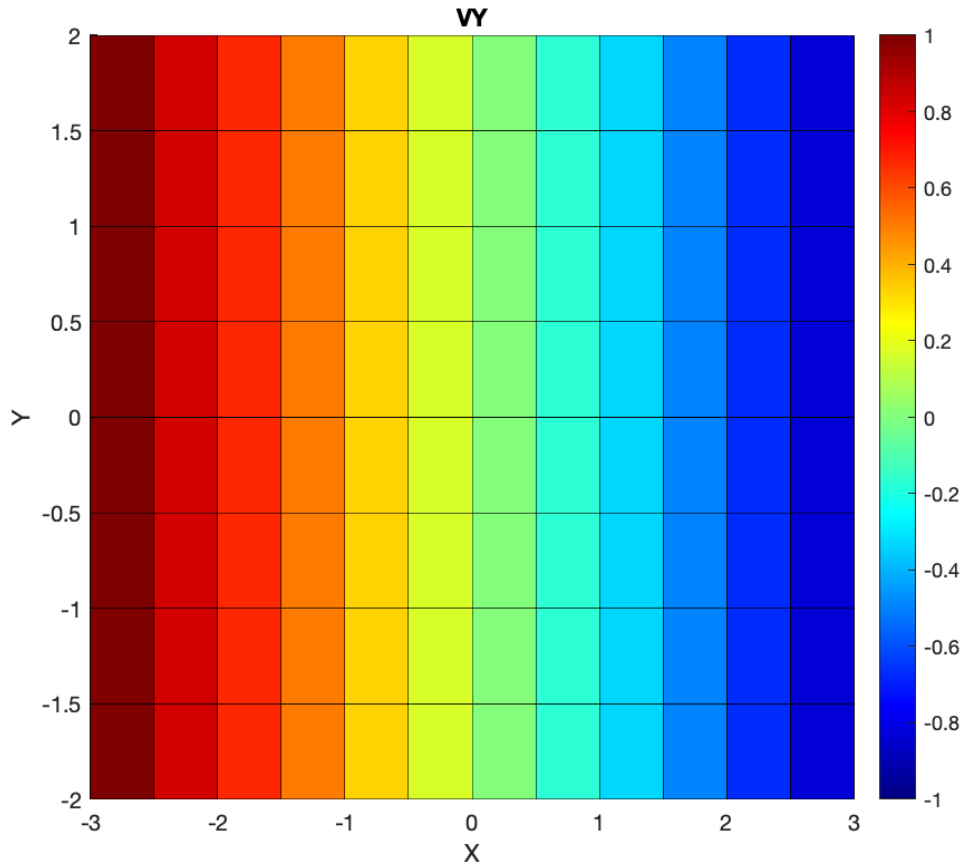
Visualizing velocity fields: `surface` plots

An alternative, `surface` produces a 3D shaded surface of a matrix or surface with a top down view.

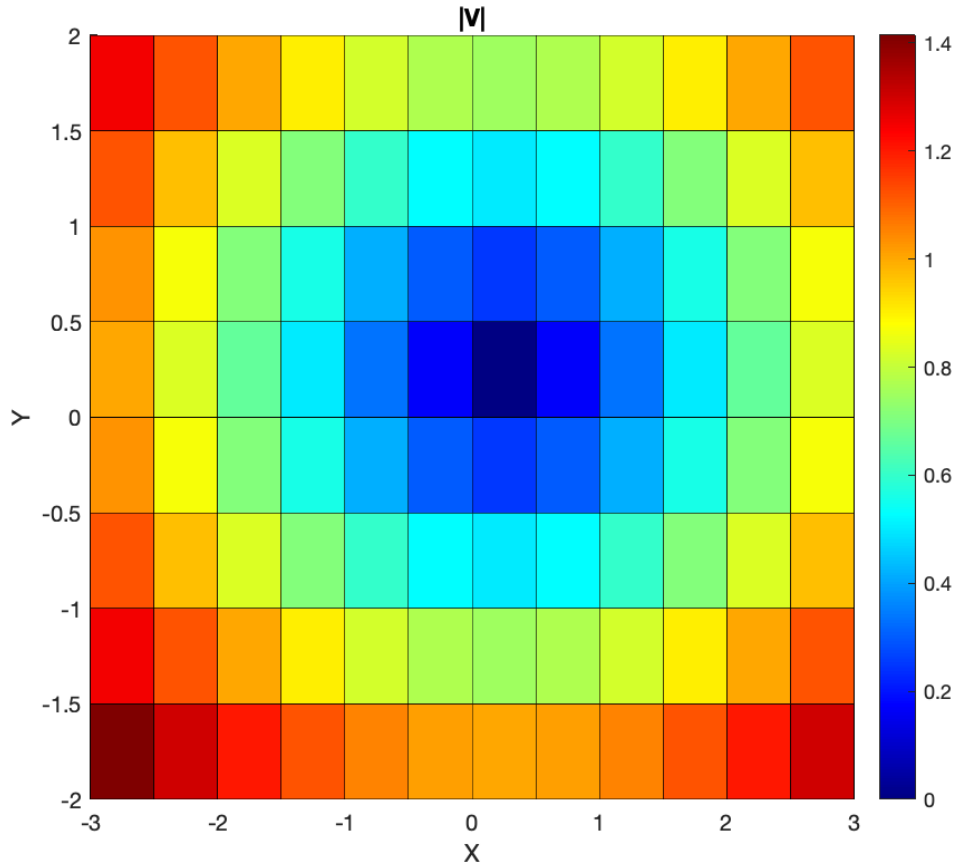
```
close all

% Plot the X component of the velocity
figure
surface(X,Y,VX)
axis square
title('VX')
xlabel('X')
ylabel('Y')
colorbar
colormap('jet')
```




```
% Plot the Y component of the velocity
figure
surface(X,Y,VY)
axis square
title('VY')
xlabel('X')
ylabel('Y')
colorbar
colormap('jet')
```




```
% Plot the magnitude of the velocity
figure
surface(X,Y,sqrt(VY.^2+VX.^2))
axis square
title('|V|')
xlabel('X')
ylabel('Y')
colorbar;
colormap('jet')
```



<u>Note</u>: The above code adds colorbars using the `colorbar` function. The purpose of this is serve as a "color scale."

Multi-dimensional differentiatial calculus

In addition to 1-D differentiation, MATLAB can handle multivariable and vector calculus derivatives.

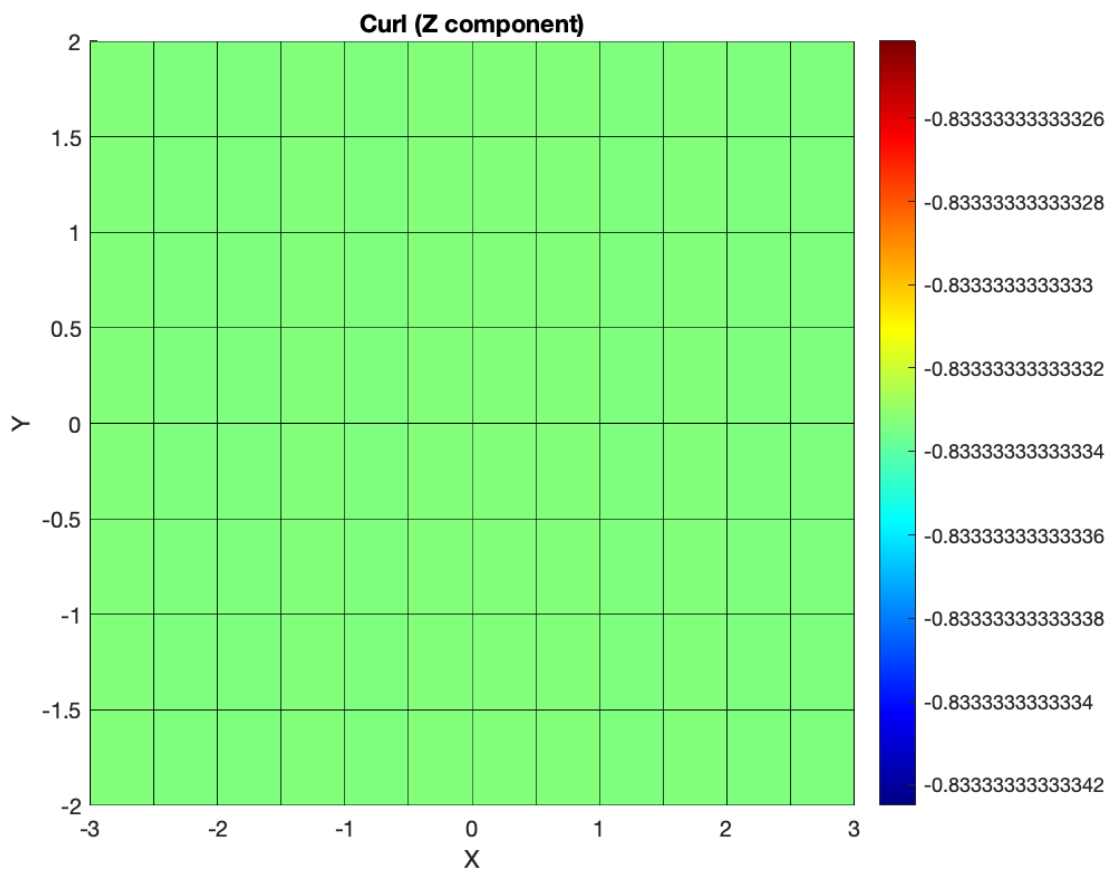
- curl, $\nabla \times F$
- gradient, ∇F
- divergence, $\nabla \cdot F$

Where ∇ = nabla = del operator = $\frac{\partial}{\partial x}\hat{i} + \frac{\partial}{\partial y}\hat{j} + \frac{\partial}{\partial z}\hat{k}$

Curl. The curl of a vector is the cross product of the del operator and the vector and can be invoked using the `curl` function in MATLAB. **Note that the curl of a vector returns a vector.**

$$\nabla \times F = \begin{vmatrix} \hat{i} & \hat{j} & \hat{k} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ F_x & F_y & F_z \end{vmatrix} = \left(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}\right)\hat{i} - \left(\frac{\partial F_z}{\partial x} - \frac{\partial F_x}{\partial z}\right)\hat{j} + \left(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}\right)\hat{k}$$

```
close all
% Compute the curl and angular velocity of the above vector field
[CURLZ, CAV]= curl(X,Y,VX,VY);

% Plot the X component of the velocity
surface(X,Y,CURLZ)
axis square
title('Curl (Z component)')
xlabel('X')
ylabel('Y')
colorbar
colormap('jet')
```



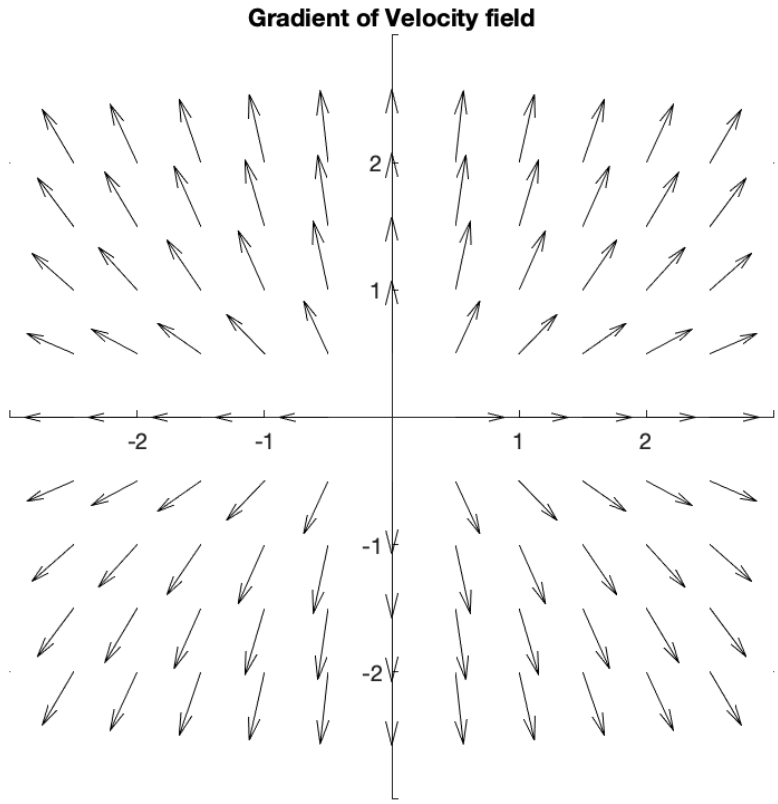
<u>In physical systems</u>, the curl can be thought of as the circulation around an infinitesimal point - the direction of the vector describes the direction of rotation via the right hand rule.

<u>In this example</u>: the entire field is rotating around one point. Thus the curl (and angular velocity) is the same everywhere.

Gradient. The gradient of a function is the result of passing a function through the del operator and may invoked using the `gradient` function in MATLAB. **Note that the gradient of a vector returns a vector.**

$$\nabla F = \frac{\partial F}{\partial x}\hat{i} + \frac{\partial F}{\partial y}\hat{j} + \frac{\partial F}{\partial z}\hat{k}$$

```
close all
% Compute the curl and angular velocity of the above vector field
[GX,GY] = gradient(sqrt(VX.^2+VY.^2),0.5,0.5);

% Plot the X component of the velocity
q = quiver(X,Y,GX,GY);
axis square
xlim([-3,3])
ylim([-3,3])
q.Color = 'k';
set(gca,'YAxisLocation','Origin')
set(gca,'XAxisLocation','Origin')
set(gca,'Box','off')
axis square
title('Gradient of Velocity field')
```



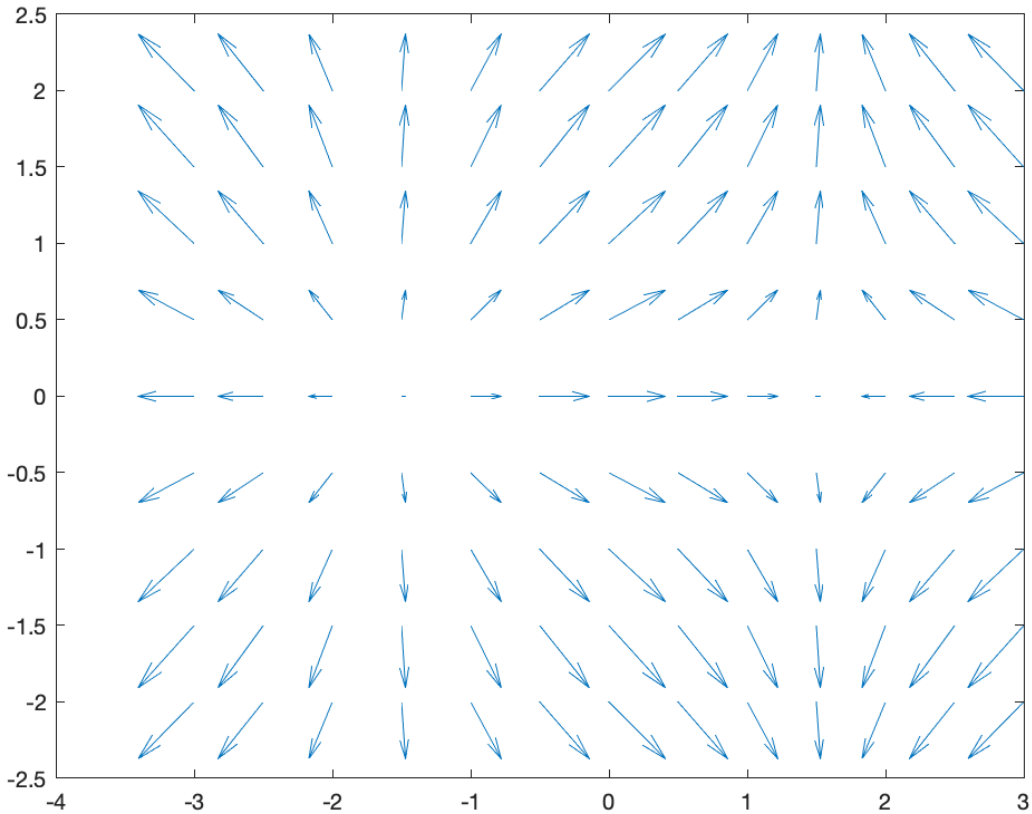
<u>In physical systems</u>, the gradient be thought of as a vector in the direction of the **steepest slope** of the function at the point of evaluation

<u>In this case</u>, the gradient of the velocity is a function of **position**.

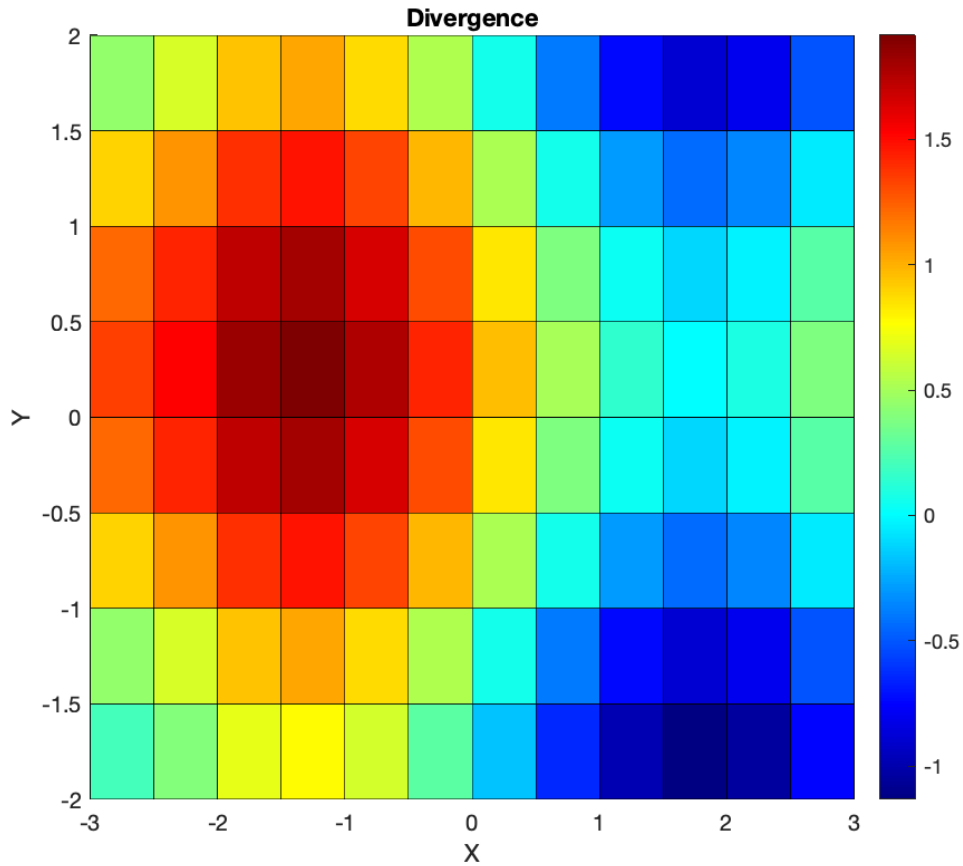
Divergence. The divergence of a vector is the result of the dot product of the del operator and the vector and may invoked using the `divergence` function in MATLAB. **Note that the divergence of a vector returns a scalar**.

$$\nabla \cdot F = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial y} + \frac{\partial F}{\partial z}$$

```
close all
% New variable C
CX = cos(X);
CY = sin(Y);
quiver(X,Y,CX,CY)
```




```
figure
% Calculate the divergence
DIV = divergence(X,Y,CX,CY);

% Plot the X component of the velocity
surface(X,Y,DIV)
axis square
title('Divergence')
xlabel('X')
ylabel('Y')
colorbar
colormap('jet')
```



<u>In physical systems</u>, the divergence may be thought of as the flux rate out of an infinitesimal region (+ for flux outward)

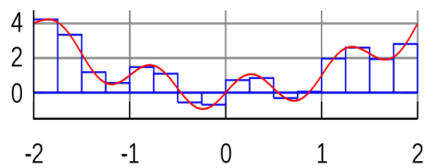
Integration

MATLAB offers the ability to perform **numerical integration** using the following functions (in order of increasing accuracy).

<u>Note</u>: these are **numerical** methods -- the integral can approximated from **evaluations of the original function**.

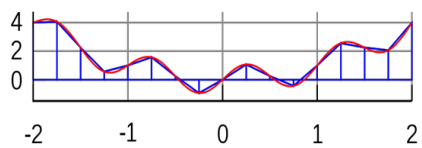
- <u>Rectangle method</u>:no built-in implementation in MATLAB

$$A = \int_a^b f(x)\,dx \approx y_1\Delta x + y_2\Delta x + y_3\Delta x + \cdots + y_{n-1}\Delta x$$

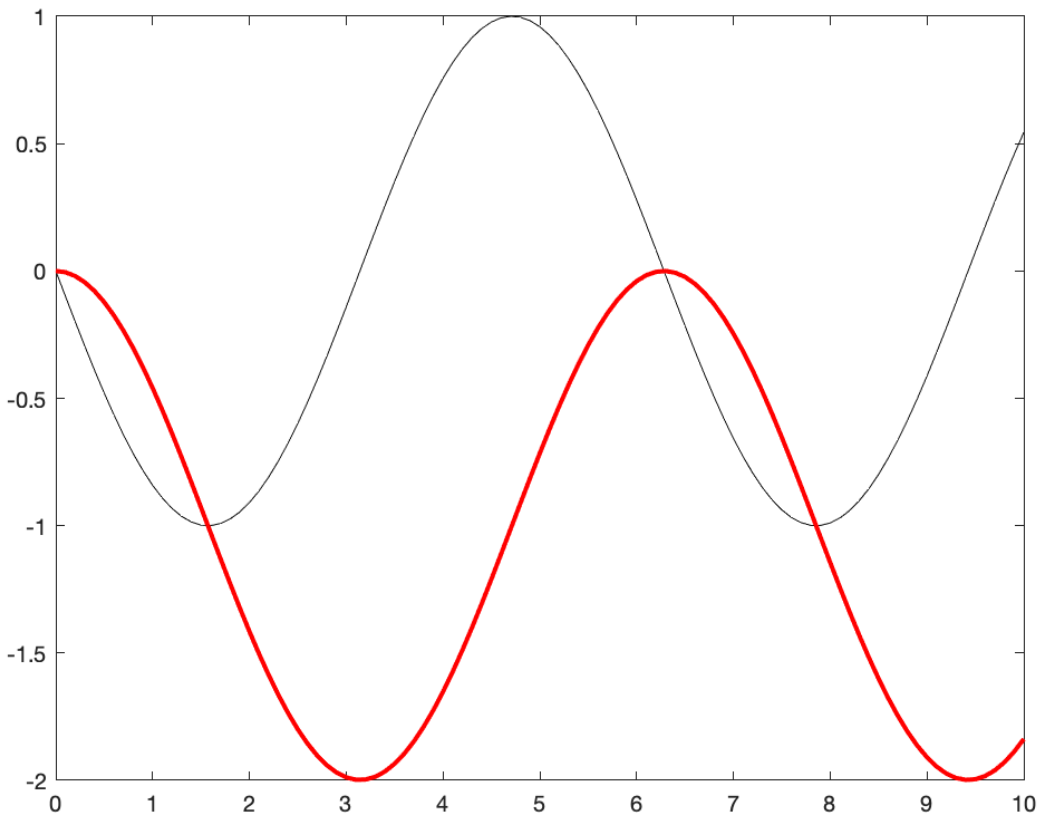


- Trapezoidal integration: accessed using the `trapz` function

$$A = \int_a^b f(x)\,dx \approx (y_1 + y_2)\frac{\Delta x}{2} + (y_2 + y_3)\frac{\Delta x}{2} \cdots + (y_{n-1} + y_n)\frac{\Delta x}{2}$$

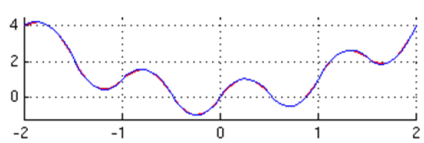



```
close all
% Define X and Y variables
x = 0:0.1:10;
y = -sin(x);
% Calculate definite integral over time
Y = zeros(size(y));
for k = 2:length(y)
    Y(k) = trapz(x(1:k),y(1:k));
end

% Plot the function and its *definite* integral
plot(x,y,'k-')
hold on
plot(x,Y,'r-','LineWidth',2)
hold off
```



- Quadrature integration: accessed using the `integral` function

$$A = \int_a^b f(x)\,dx \approx \frac{h}{3}\left(y_1 + y_n + 4\sum_{i=1}^{n-1} y_{2i} + 2\sum_{i=1}^{n} y_{2i-1} \right), \quad h = \frac{b-a}{2n}$$



Numerical integration is most often approached in MATLAB using the `integral` function, which requires input as an anonymous function.

```
% Create function handle for function of interest
F = @(x) -sin(x)
```

```
F = function_handle with value:
    @(x)-sin(x)
```

```
% Calculate integral
q = integral(F,0,pi)
```

```
q = -2.0000
```

Engineering example: controllers

When describing (biological, chemical, mechanical, electrical) engineering systems, the engineer is often interested in how a given **output** results from a given **input**.

Examples of systems:

- A medical device for regulating blood sugar levels
- A chemical reactor for producing acetylene
- A prosthetic hand

Examples of inputs:

- A voltage (related to amount of insulin to be injected into the patient).
- Calcium carbide reagent entry into a reactor
- An electrical signal applied to a mechanical actuator

Examples of outputs:

- A voltage from a sensor that is proportional to blood glucose levels
- Material and energy produced from chemical reaction.
- Change in position of prosthetic fingers

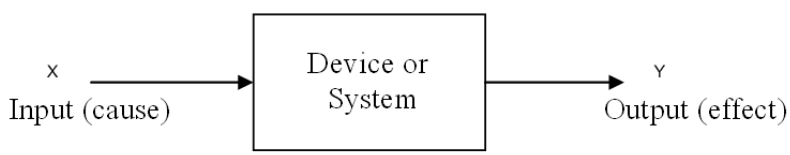


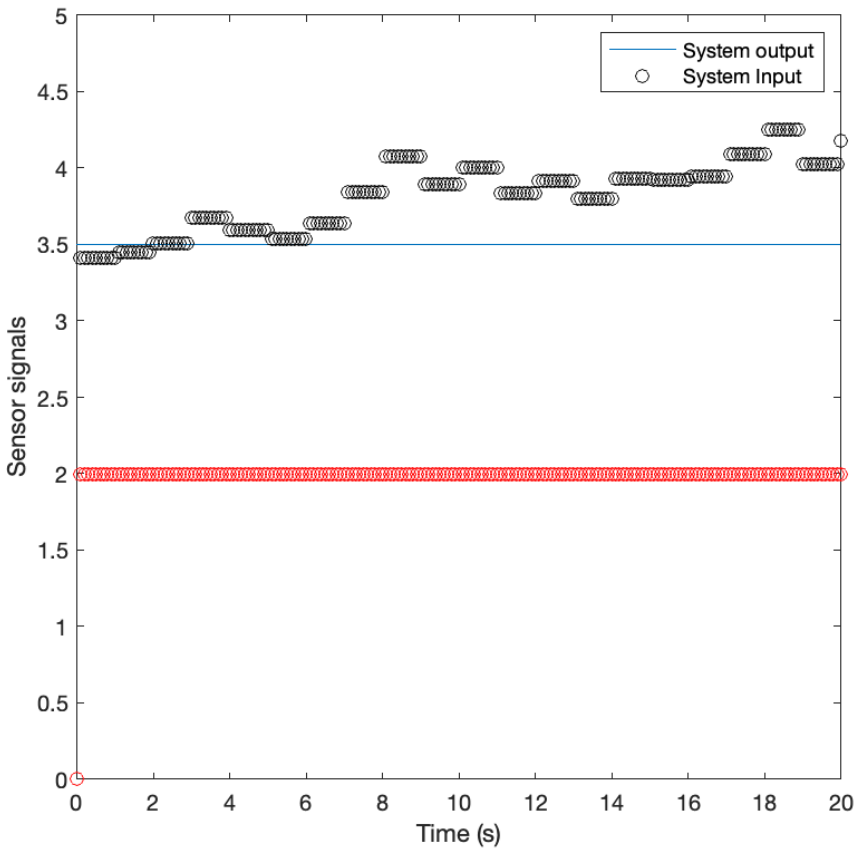
Fig – A: Block Diagram of a System with input and output

For example, consider the following code that represents some system. You can assume the following represents a device, in which it can be read (in real time) how an **output voltage** changes in response to the **input voltage.**

```
close all
```

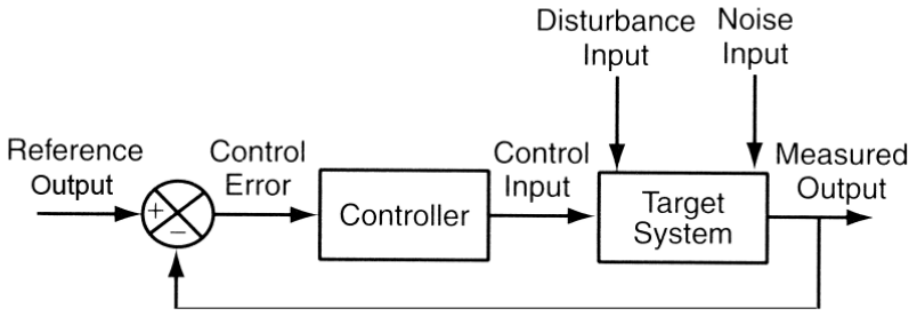
```
% Create figure
h1 = figure;

plot([0,20],3.5*[1,1])
hold on
% Output "sensor" data
g_out = plot(0,0,'ko');
hold on
% Input "sensor" data
g_in = plot(0,0,'ro');
hold off

% Graphical parameters
xlim([0,20]);
ylim([0,5]);
axis square
ylabel('Sensor signals')
xlabel('Time (s)')
legend('System output','System Input','Location','NorthEast')

% Counter to store time
time = 0;
outError = 0;

% Plot in real time for 20 seconds
while time < 20

    % Pause for 100 ms
    pause(0.1);

    % Update time
    time = time + 0.1;

    % The following describes how the system responds to a given input
    % Assume the system drifts in some time dependent fashion
    outError = outError + (abs(mod(time,1)) < 0.1)*(rand - 0.5);
    % Function describing how output responds to input
    y = @(x) sqrt(x) + 2 + 0.5*outError;

    % Update vector to contain input information
    g_in.XData = [g_in.XData,time];
    g_in.YData = [g_in.YData,2];

    % Update vector to contain input information
    g_out.XData = [g_out.XData,time];
    g_out.YData = [g_out.YData,y(g_in.YData(end))];
```

```
end
```



Instead of simply observing how an output changes in reponse to a given input, engineers seek to **control** how the output varies in time. For example, a chemical process engineer wants to ensure that a chemical system is safely operating within the constraints providing by the equipment, process of interest, etc.

One way to implement this process control is to integrate a **feedback controller**. As shown in the following diagram, feedback control enables one to **adjust an input depending on how the output is changing** over time.



Common types of controllers:

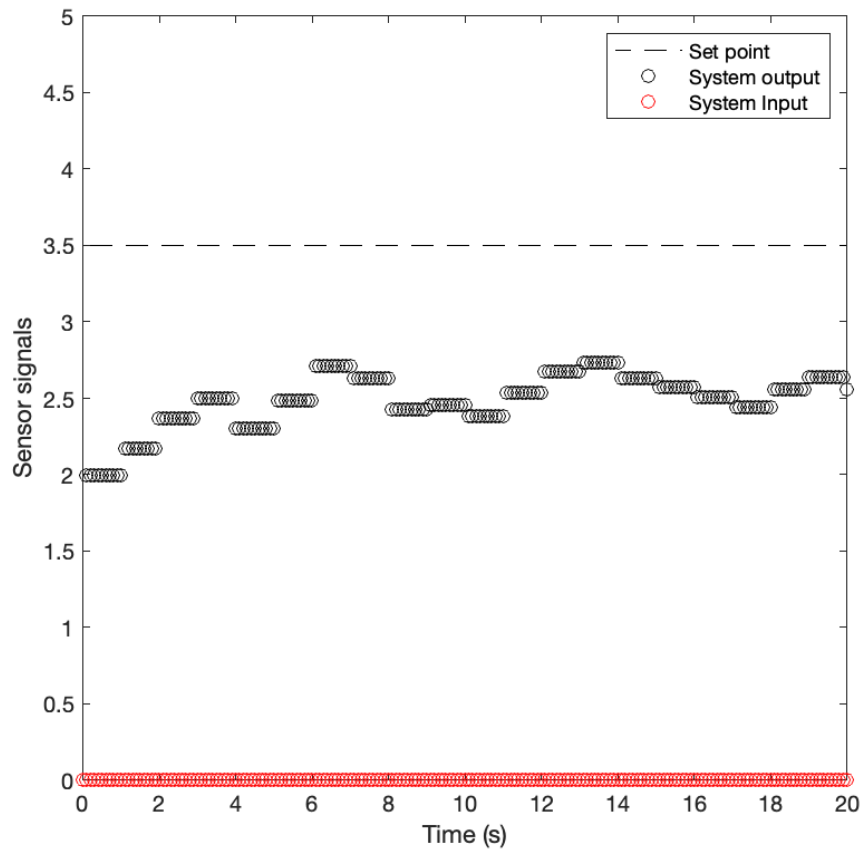
- **Proportional**: the control input is proportional to the difference between the actual ouput and the set point.
- **Integral**: the conrol input is proporitional to the integrated difference between the actual output and the set point
- **Differential**: the control input is proportional to the differentiated difference between the actual output and the set point

Controllers can integrate multiple control strategies: **PI** controllers, **PID** controllers, etc.

Goal. Create a **Proportional** and **PI** controller and observe the signal in real time.

```
close all

% Create figure
h1 = figure;

% Determine our goal set point
ysp = 3.5;

plot([0,20],ysp*[1,1],'k--')
hold on
% Output "sensor" data
g_out = plot(0,0,'ko');
% Input "sensor" data
g_in = plot(0,0,'ro');
plot(xlim,ysp,'k-')
hold off

% Graphical parameters
xlim([0,20]);
ylim([0,5]);
axis square
ylabel('Sensor signals')
xlabel('Time (s)')
legend('Set point','System output','System Input','Location','NorthEast')
```

```
% Counter to store time
time = 0;
outError = 0;

% Declare a variable to represent the error from the set point
x_err = 0;

% Plot in real time for 20 seconds
while time < 20

    % Pause for 100 ms
    pause(0.1);

    % Update time
    time = time + 0.1;

    % The following describes how the system responds to a given input
    % Assume the system drifts in some time dependent fashion
    outError = outError + (abs(mod(time,1)) < 0.1)*(rand - 0.5);
    % Function describing how output responds to input
    y = @(x) sqrt(x) + 2 + 0.5*outError;

    % DEFINE CONTROLLER HERE
    if time > 0.1



    end


    % Update vector to contain input information
    g_in.XData = [g_in.XData,time];
    g_in.YData = [g_in.YData,max(0,x_err)];

    % Update vector to contain input information
    g_out.XData = [g_out.XData,time];
    g_out.YData = [g_out.YData,y(g_in.YData(end))];

end
```



TOPIC 23: Differential equation solvers and other numerical methods

ENGR105, 11/19/20

Solving ordinary differential equations (ODEs)

ODEs desribe a relationship between derivatives with respect to a single variable. For example, a common **first order ODE** relates the time derivative of a function, $\frac{d}{\mathrm{d}t}x(t)$, to the function itself, $x(t)$, and the independent variable:

$$\frac{d}{\mathrm{d}t}x(t) = f(t, x(t))$$

The **<u>order</u>** of an ODE is the highest order derivative. **Higher-order ODEs** relate higher-order derivatives to lower order derivatives, the dependent variable / function, and the independent variable.

$$\frac{d^2}{dt^2}x(t) = f\left(t, x(t), \frac{d}{dt}x(t)\right)$$

<u>Remember</u>: to solve an ODE, the number of initial conditions must be same as the order.

Systems of ordinary differential equations appear in many engineering and science disciplines:

- Robotics and systems control
- Weather systems
- Population growth, logistics, and radioactive decay
- Dynamics of physical and electronic systems
- Many more ...

For **simple cases**, ODEs can be solved analytically by direct integration.

- Example 1

$$\frac{d}{dt}x(t) = 3t^2 + 4t + \sin(t)$$

$$x(t) = t^3 + 2t^2 - \cos(t) + c_1$$

- Example 2

$$\frac{d}{dt}x(t) = k\,x(t)$$

$$x(t) = c_2\,e^{kt}$$

However, many (if not most) ODEs can be solved more easily using a **numerical algorithm**.

Taylor series approximations

An underrated and extremely useful approximation for engineers involves the Taylor series representation of a function. A **one-dimensional** Taylor series is an expansion of a real function about a point **a** given by:

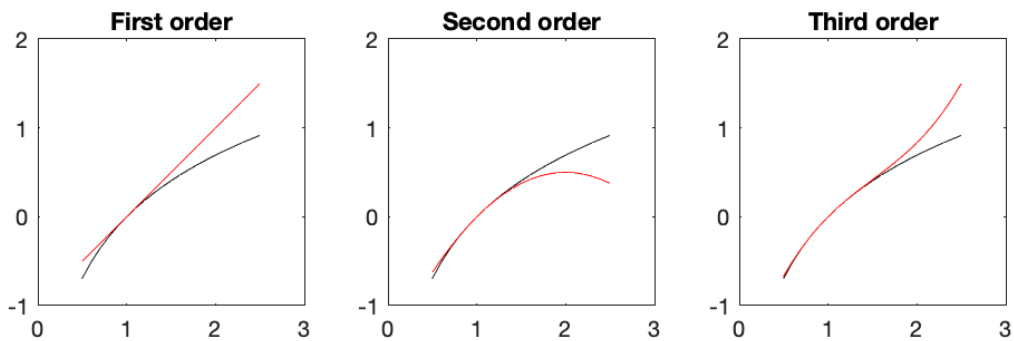
$$f(x) = f(x)|_a + \left.\frac{df}{dx}\right|_a (x-a) + \frac{1}{2!}\left.\frac{d^2f}{dx^2}\right|_a (x-a)^2 + \frac{1}{3!}\left.\frac{d^3f}{dx^3}\right|_a (x-a)^3 + O(x^4)$$

This is **extremely useful** because it allows us to represent an arbitrary function in terms of an infinite polynomial. In many cases (especially in engineering), we **approximate** a function by only considering the **first order term** in the Taylor series expansion:

$$f(x) \approx f(x)|_a + \left.\frac{df}{dx}\right|_a (x-a)$$

Example. Compare the first, second, and third order approximations of $f(x) = \ln(x)$ at x = 1

```
% x range of interest
x = 0.5:0.1:2.5;
% Actual function
f = log(x);

% Taylor series approximations
f_ts1 = x-1;
f_ts2 = x-1 - 1/2*(x-1).^2;
f_ts3 = x-1 - 1/2*(x-1).^2 + 1/3*(x-1).^3;


subplot(1,3,1)
plot(x,f,'k-',x,f_ts1,'r-')
axis square
ylim([-1,2])
title('First order')
subplot(1,3,2)
plot(x,log(x),'k-',x,f_ts2,'r-')
axis square
ylim([-1,2])
title('Second order')
subplot(1,3,3)
plot(x,log(x),'k-',x,f_ts3,'r-')
axis square
ylim([-1,2])
title('Third order')
```



Takeaway: the **more terms** that are included, the *better* the approximation will be.

Euler method: simple approximation of an ODE solution

Consider first order differential equation:

$$\frac{d}{\mathrm{d}t}x(t) = f(t, x(t))$$

The solution to this ODE is $x(t)$. By performing a Taylor series expansion about t, we can approximate the value of the solution after some time interval:

$$x(t + \Delta t) \approx x(t) + \left.\frac{dx}{dt}\right|_t ((t + \Delta t) - t)$$

$$x(t + \Delta t) \approx x(t) + f\Delta t$$

This representation is known as **Euler's method**. It provides a first-order approximation for the next state of the system and **an approximate solution** can be created by **numerically** calculating the "next" point. As can be seen above, <u>the **accuracy** of this approach is strongly determined by the **size of the time step**, Δt.</u>

Example. Consider the following differential equation. Use the **Euler method** to find a numerical solution.

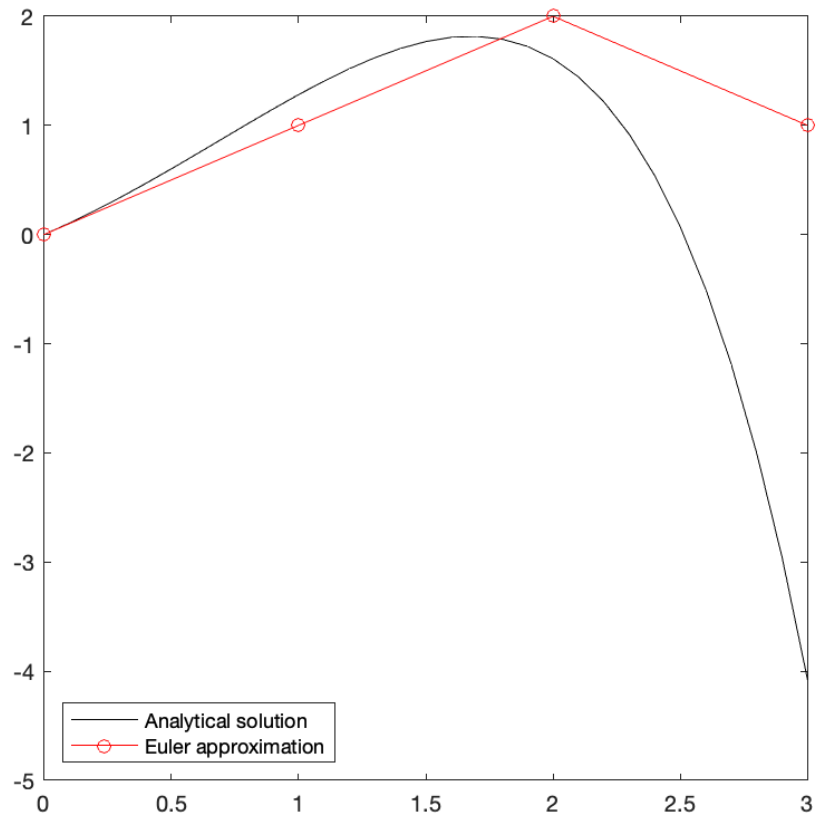
$$\frac{d}{dt}x(t) = x - t^2 + 1, \text{ where } x(0) = 0$$

This can be solved using an *analytical technique* (*e.g.* with a Laplace Transform) to produce the solution:

$$x(t) = 1 - e^t + t^2 + 2t$$

Applying the Euler method:

```
close all
% Time vector
t = 0:0.1:3;

% The function representing the RHS of the differential equation. This is
% defined by the problem
f = @(T,X) X-T^2+1;

% Analytical solution
x_an = (1-exp(t))+t.^2+2*t;

% Use Euler method
% Define the time step
dt = 1;

% Preallocate vector to store
t_approx = 0:dt:3;
x_euler = zeros(size(t_approx));

% Calculate each entry in the Euler approximation
for k = 1:length(x_euler)-1
    % Apply the Euler method
    % Calculate the next value of the solution
    x_euler(k+1) = x_euler(k) + f(t_approx(k),x_euler(k))*dt;
end

% Plot the solution
plot(t,x_an,'k-',t_approx,x_euler,'ro-')
axis square
legend('Analytical solution','Euler approximation','location','southwest')
```



<u>Note</u>: there is a tradeoff between accuracy and computational cost.

Predictor-corrector method: better approximation of a solution

A better approximation than Euler can be achieved using the predictor-corrector method. Here, consider again:

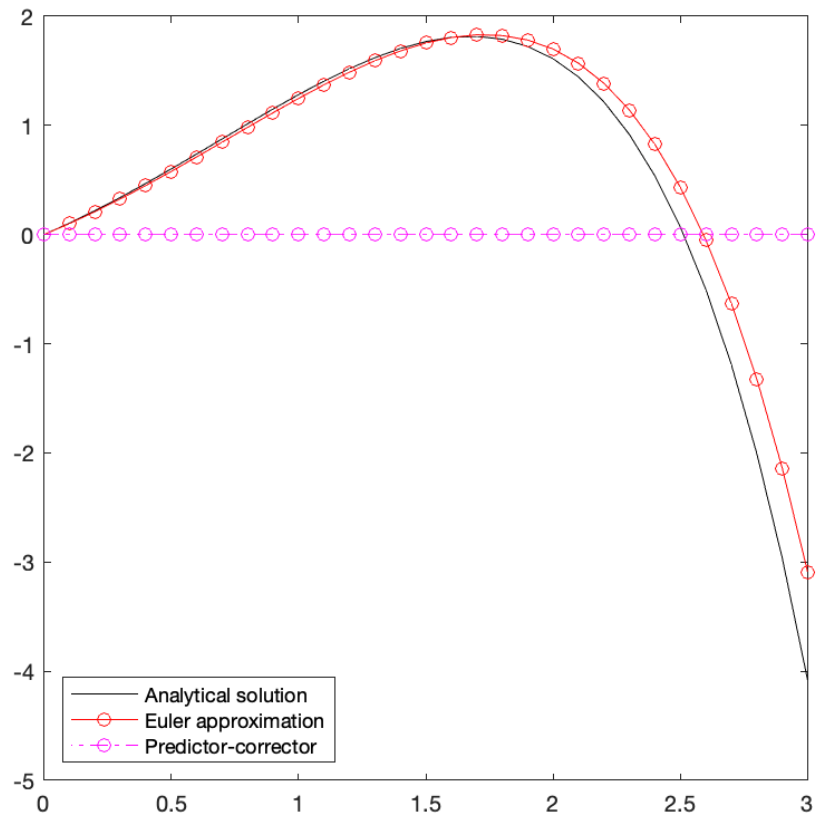
$$\frac{d}{\mathrm{d}t}x(t) = f(t, x(t))$$

If we include some more *higher order terms in the Taylor Series approximation*, we would expect to achieve a better approximation. The predictor-corrector method does that, effectively estimating the derivative at the beginning (k_1) and end (k_2) of an interval

$$\begin{aligned} x(t+\Delta t) &\approx x(t) + \frac{1}{2}(f(t, x(t)) + f(t+\Delta t, x(t) + f(t, x(t))\Delta t))) \\ &\approx x(t) + \frac{1}{2}(k_1 + k_2)\Delta t \\ k_1 &= f(t, x(t)) \\ k_2 &= f(t+\Delta t, x(t) + k_1 \Delta t) \end{aligned}$$

Applying predictor-corrector and comparing to the above:

```
close all
% Define the time step
dt = 0.1;

% Preallocate vectors to store
t_approx = 0:dt:3;                % Time vector for all
x_euler = zeros(size(t_approx)); % Euler
x_pc = zeros(size(t_approx));    % Predictor corrector

% Calculate each entry in the Euler approximation
for m = 1:length(x_euler)-1
    % Use Euler method
    % Calculate the next value of the solution
    x_euler(m+1) = x_euler(m) + f(t_approx(m),x_euler(m))*dt;

    % Use predictor-corrector method
    k1 = f(t_approx(m),x_pc(m));
    k2 = f(t_approx(m)+dt,x_pc(m)+k1*dt);

end

% Plot the solutions
plot(t,x_an,'k-', ...
    t_approx,x_euler,'ro-',...
    t_approx,x_pc,'mo-.')
axis square
legend('Analytical solution','Euler approximation','Predictor-corrector',...
    'location','southwest')
```



Runge-Kutta solutions to differential equations

Including even more terms, the **Runge-Kutta** method provides a better approximation than predictor-corrector. Runge-Kutta can also be thought of as "following the *logic of the predictor-corrector method,*" where the accuracy of the estimate can be improved by evaluating the derivative at *multiple points* along an interval.

$$x(t+\Delta t) \approx x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t$$

$$k_1 = f(t, x(t))$$

$$k_2 = f\left(t + \frac{\Delta t}{2}, x(t) + k_1 \frac{\Delta t}{2}\right)$$

$$k_3 = f\left(t + \frac{\Delta t}{2}, x(t) + k_2 \frac{\Delta t}{2}\right)$$

$$k_4 = f(t + \Delta t, x(t) + k_3 \Delta t)$$

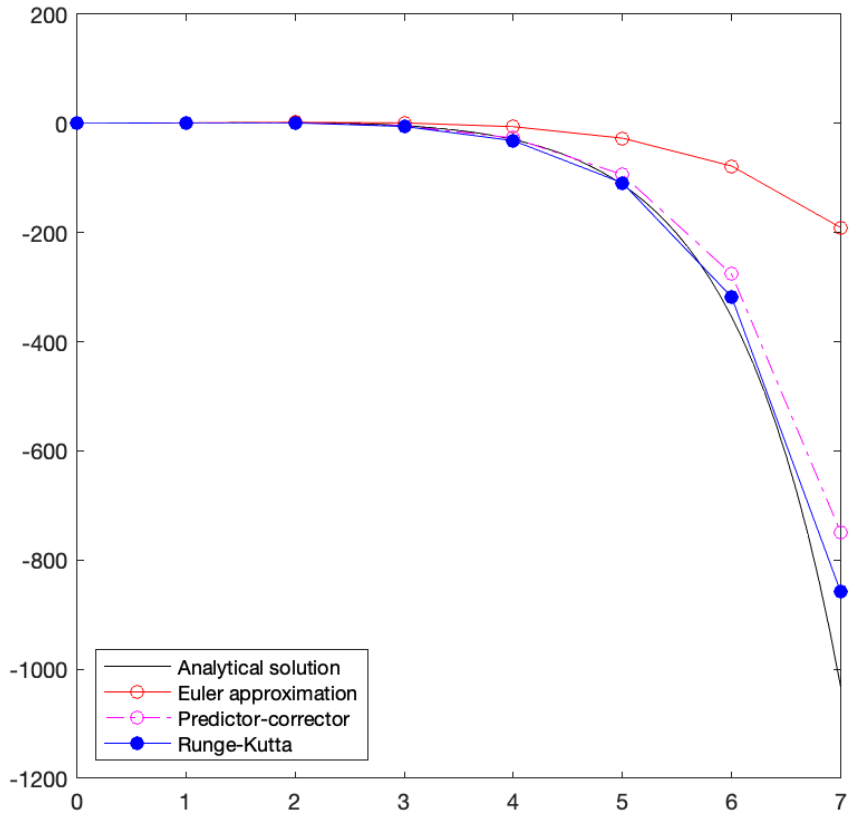
Here, k_1 corresponds to the derivative at the beginning of the interval, k_2 and k_3 are two estimates for the derivative at the midpoint of the interval, and k_4 is an estimate of the derivative at the end of the interval.

The above is a **Runge-Kutta approximation of rank 4** (RK4).

Applying Runge-Kutta and comparing to the above:

```
close all

% Time vector
t = 0:0.1:7;
% Analytical solution
x_an = (1-exp(t))+t.^2+2*t;

% Define the time step
dt = 1;

% Preallocate vectors to store
t_approx = 0:dt:7;                % Time vector for all
x_euler = zeros(size(t_approx)); % Euler
x_pc = zeros(size(t_approx));    % Predictor corrector
x_rk = zeros(size(t_approx));   % Runge-Kutta

% Calculate each entry in the Euler approximation
for m = 1:length(x_euler)-1
    % Use Euler method
    % Calculate the next value of the solution
    x_euler(m+1) = x_euler(m) + f(t_approx(m),x_euler(m))*dt;

    % Use predictor-corrector method
    k1 = f(t_approx(m),x_pc(m));
    k2 = f(t_approx(m)+dt,x_pc(m)+k1*dt);
    x_pc(m+1) = x_pc(m) + 1/2*(k1+k2)*dt;

    % Use Runge-Kutta method
    K1 = f(t_approx(m),x_pc(m));
    K2 = f(t_approx(m)+dt/2,x_pc(m)+K1*dt/2);
    K3 = f(t_approx(m)+dt/2,x_pc(m)+K2*dt/2);
    K4 = f(t_approx(m)+dt,x_pc(m)+K3*dt);
    x_rk(m+1) = x_rk(m) + 1/6*(K1 + 2*K2 + 2*K3 +K4)*dt;

end

% Plot the solutions
plot(t,x_an,'k-', ...
    t_approx,x_euler,'ro-',...
    t_approx,x_pc,'mo-.')
hold on
plot(t_approx,x_rk,'bo-','MarkerFaceColor','b')
```

```
hold off
axis square
legend('Analytical solution','Euler approximation','Predictor-corrector',...
    'Runge-Kutta','location','southwest')
```



Runge-Kutta ODE solvers of RK23 and RK45 are most often used in MATLAB and can be accessed with the commands `ode23()` and `ode45()`

<u>Note</u>: How do we ever know when we have the 'right' numerical solution? We don't - the best we can do is increase the accuracy of the numerical method until no further wild changes occur over the interval of interest.

Like in the previous examples, we would decrease the time step and/or increase the solution accuracy (`'AbsTol'` and `'RelTol'`) to achieve to see when the solution settles

Differential equation solvers in MATLAB

MATLAB offers a wide array of differential equation solvers - each has their own application set or unique "personality." The two most popular, `ode23()` and `ode45()` are based on the **Runge-Kutta** algorithm (with *additional* numerical details as well).

<u>Note</u>: <u>`ode45()` should be your default ODE solver for most applications.</u>

Like our previous numerical methods, these built-in functions require an input function containing the differential equation to be solved for.

Revisiting Newton-Raphson. Consider our numerical root-finding algorithm that we created in Topic 12. This function takes an input function (`F`), an initial guess (`xn`), and a user defined tolerance (`tol`) that determines how long the iterative method should be repeated. *The input function is repeatedly evaluated during the iterative process.*

```
function xn = newtonSolve(F,xn,tol)
    % Function for using the Newton-Raphson method to solve for f(x)

    % Step 1: specify initial guess
    xprev = inf; % Variable to store previous xn value -> ensure it is
    % not xn to enter while loop

    % Step 2: iterate until xn -> xn+1
    while abs(xn - xprev) > tol

        % Step 3: evaluate function and derivative at xn
        f = F(xn);

        % Instead of calculating the "next" x value, store the previous
xn
        % value
        xprev = xn;

        % Step 3b: update xn to next value for the iteration
        xn = xn - f(1)/f(2);
    end
end
```

An example input function:

```
function F = originalEqn(x)
    % Function of interest
    F(1) = x^4 + 2*(x-2)^2 - 20;
    % Derivative of function of interest
    F(2) = 4*x^3 + 4*(x-2);
end
```

Our function would be called as:

```
% Determine root of original f(x)
rootF = newtonSolve(@originalEqn,-5,10^-5)
```

In the above case, the solver function repeatedly interacts with the equation function during the "solving process." `ode45` requires similar formatting - namely representation of the differential equations in <u>state space</u> and the use of <u>function handles</u>

```
[t,out] = ode45(function, domain, initial conditions, options, input
parameters/constants...)
```

- `outputs ([t, out]):` time and corresponding position(s), velocity, etc.
- `function`: handle to function containing the description of the ODE
- `domain`: the analysis time, a 1x2 row vector `[t_min,t_max]`
- `initial conditions`: initial position(s), velocity, etc. - a 1 x n row vector depending on how many ODEs are being solved.
- `options`: user specified accuracy, resolution, etc.
- `parameters/constraints`: constants used during the evaluation of the ODE

Example. Consider the example above

$$\frac{d}{dt}x(t) = x - t^2 + 1, \text{ where } x(0) = 0$$

<u>This ODE can be stored in the following function</u> to be used with `ode45`

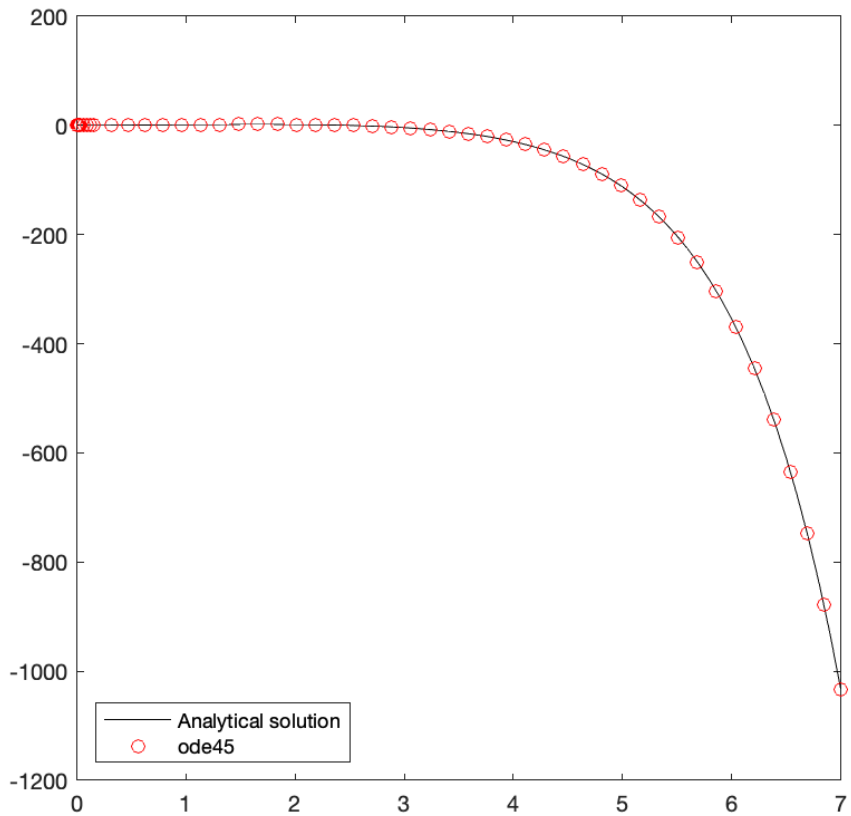
```
function dxdt = myODE(t,x)
    dxdt = x(1) - t^2 + 1;
end
```

<u>Notes:</u>

- Analagously to Newton-Raphson, the numerically-solved differential function is stored as its **own MATLAB function**
- It is important that a variable representing **time appears as the first input** to the ODE function even if not *explicitly* referenced in the ODE function

Solving the ODE:

```
close all
% Solve ODE using ode45
[ts,xs] = ode45(@myODE,[0,7],[0]);

% Compare to analytical solution
plot(t,x_an,'k-',ts,xs,'ro')
axis square
legend('Analytical solution','ode45','location','southwest')
```



Notes:

- ode45 requires inputs of time and initial conditions, and returns a vector or matrix of solutions to the differential equation
- Instead of input start and end points of a time domain, you can also **enter a vector of time points to evaluate the function at**.

Example: chemical reaction kinetics

Consider an arbitrary set of chemical reactions.

$$A \underset{k_{1,r}}{\overset{k_{1,f}}{\Longleftrightarrow}} B$$

$$B \underset{k_{2,r}}{\overset{k_{2,f}}{\Longleftrightarrow}} C$$

Queston. If 1 mol of A is initially present, how does the composition of the system evolve over time? In this case, assume the reaction has following rate constants: $k_{1,f} = 1\,s^{-1}$, $k_{1,r} = 2\,s^{-1}$, $k_{2,f} = 2\,s^{-1}$, $k_{2,r} = 0.5\,s^{-1}$

First observation: we are dealing with a **set of differential equations**. Applying our knowledge of chemistry and substituting in the rate constants produces the following system of differential equations:

$$\frac{dA}{dt} = -A + 2B$$

$$\frac{dB}{dt} = A - 2B + 0.5C - 2B = A - 4B + 0.5C$$

$$\frac{dC}{dt} = 2B - 0.5C$$

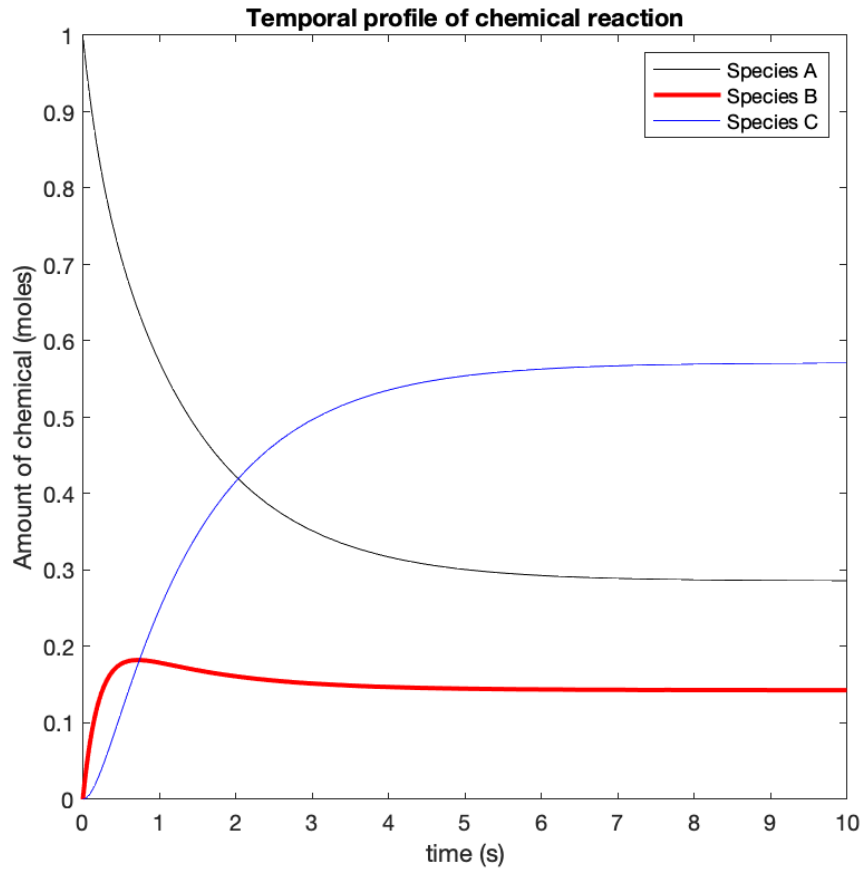
This set of differential equations can be encapsulated into a function. In this case, there a multiple inputs and multiple outputs. Each input corresponds to the amount of one chemical, and each output coresponds to the differential equation relating the change of that variable.

<u>Note</u>: the derivatives have to be returned as a **column vector**

```
function dxdt = rxnEqns(t,x)
    % x(1) : A
    % x(2): B
    % x(3): C
    % Preallocate output column vector
    dxdt = zeros(3,1);
    dxdt(1) = -x(1) + 2*x(2);
    dxdt(2) = x(1) - 4*x(2) + 0.5*x(3);
    dxdt(3) = 2*x(2) - 0.5*x(3);
end
```

```
close all
% Solve ODE using ode45
[t,moles] = ode45(@rxnEqns,[0,10],[1,0,0]);

% Plot time-dependent profiles
plot(t,moles(:,1),'k-')
hold on
plot(t,moles(:,2),'r-','LineWidth',2)
plot(t,moles(:,3),'b-')
hold off

% Graphical parameters
axis square
legend('Species A','Species B','Species C','location','northeast')
ylim([0,1])
ylabel('Amount of chemical (moles)')
xlabel('time (s)')
```

```
title('Temporal profile of chemical reaction')
```



ODEs of higher orders

The previous example was a system of **first order** ODEs. To solve a higher order ODE using `ode45`, we need to convert it to a **system of first order ODEs**.

Example: formulating a system of first order ODEs from a third-order linear ODE. Say we have an equation of the form:

$$\dddot{y} + p\ddot{y} + q\dot{y} + ry = 0$$

This would be input into MATLAB as a series of differential equations of first order using the following method.

1. Define a new variable
2. Rearrange the ODE so that the highest order derivative is on the left hand side of the equation and all other terms are on the right hand side of the equation
3. Rewrite the ODE from (2) substituting in the generalized variables
4. Relate the lower order derivatives to the higher order derivatives

1) We start by assigning generalized variables (say *x*) to derivatives of *y* of increasing order as such

$$x_1 = y$$
$$x_2 = \dot{y}$$
$$x_3 = \ddot{y}$$

2) Next, we **rearrange the ODE** to get the derivative of highest order on the left side of the equation

$$\dddot{y} = -p\ddot{y} - q\dot{y} - ry$$

3) Now, we **substitute the new variables** into the ODE

$$\dot{x}_3 = -px_3 - qx_2 - rx_1$$

4) Finally, we establish relationships between lower order derivatives and non-differentiated terms

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = x_3$$

Summary. The original third-order ODE was:

$$\dddot{y} = -p\ddot{y} - q\dot{y} - ry$$

And has been converted to the following set of linear first-order ODEs for input into a MATLAB ODE solver

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = x_3$$
$$\dot{x}_3 = -px_3 - qx_2 - rx_1$$

MATLAB implementation.

When it is in the correct format (system of differential equations), the set of equations can then be input into a function definition of the ODE for use in MATLAB.

```
function dxdt = thirdOrderODE(t,x)
    % Parameters
    p = 2;
    q = 1.5;
    r = 0.1;
```

```
    % Note:
    % x(1) corresponds to y
    % x(2) corresponds to y'
    % x(3) corresponds to y''

    % Preallocating the output as a column vector
    dxdt = zeros(3,1);

    dxdt(1) = x(2);
    dxdt(2) = x(3);
    dxdt(3) = -p*x(3)-q*x(2)-r*x(1);
end
```

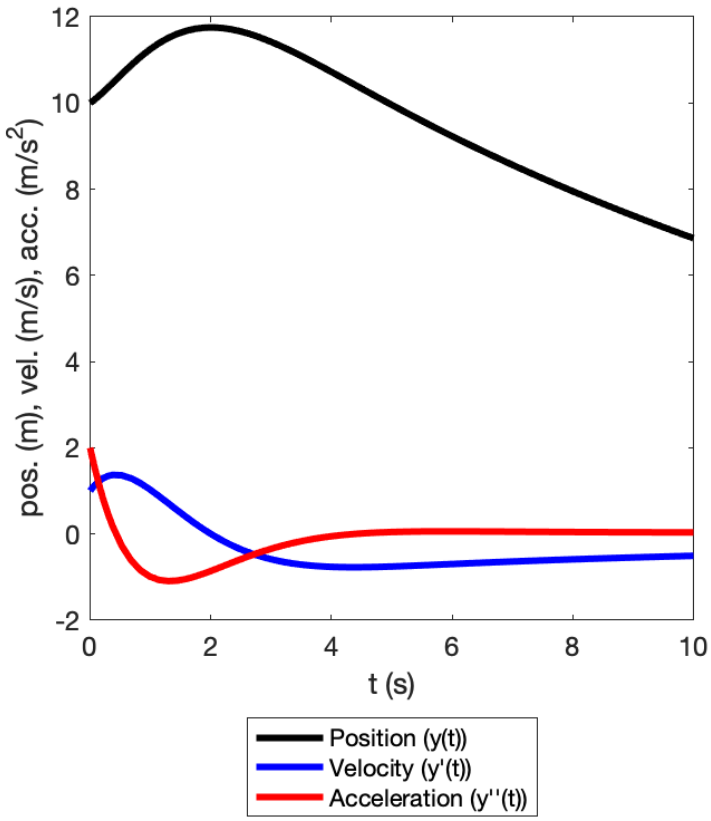
Notes:

- As before, time and x values serve as inputs
- As before, derivatives of the generalized variables have to be returned as a **column vector**

We can then solve the function by calling the `ode45` solver and pointing to the function that contains the state space representations of the ODE.

- Here, specify the following initial conditions: x(1) = 10, x(2) = 1, x(3) = 2.
- The number of initial conditions should equal the order of the ODE

```
% Solve system of linear ODES
[t,x] = ode45(@thirdOrderODE,[0,10],[10,1,2]);

% Plotting system
p = plot(t,x(:,1),'k-',...
    t,x(:,2),'b-',...
    t,x(:,3),'r-');
set(p,'LineWidth',3)
set(gca,'FontSize',12)
axis square
xlabel('t (s)')
ylabel('pos. (m), vel. (m/s), acc. (m/s^2)')
legend({'Position (y(t))','Velocity (y''(t))','Acceleration
(y''''(t))'},'location','southoutside')
```



Supplying additional arguments to `ode45`

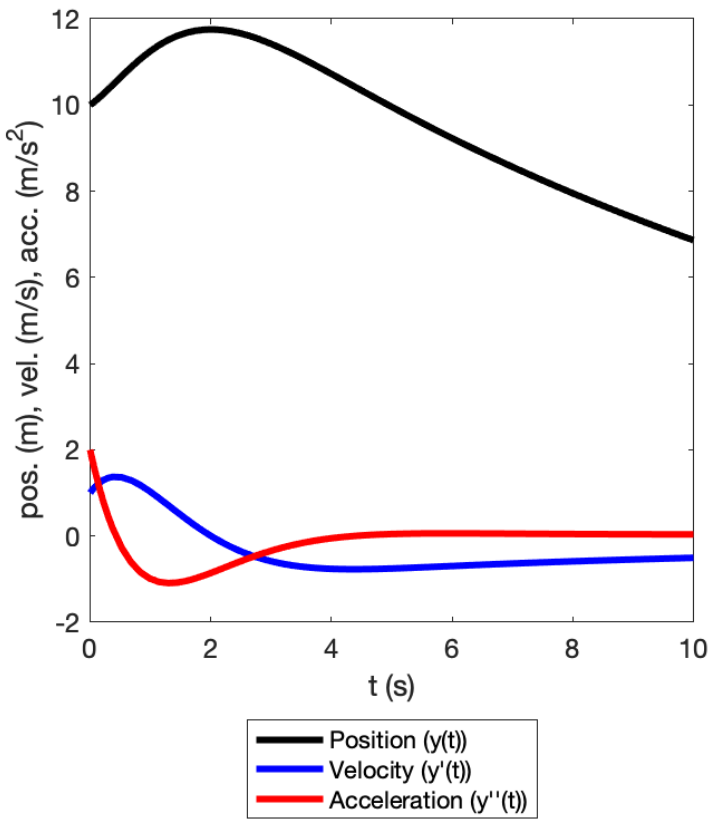
The values of *p*, *q*, and *r* were defined in the ODE function - we can also supply these values when calling the solver. The function must be modified to take three additional inputs.

```
function dxdt = thirdOrderODE_extraInputs(t,x,p,q,r)
    dxdt = zeros(3,1);
    dxdt(1) = x(2);
    dxdt(2) = x(3);
    dxdt(3) = -p*x(3)-q*x(2)-r*x(1);
end
```

```
p = 2;
q = 1.5;
r = 0.1;

% Solve the ODE
[t,x] = ode45(@thirdOrderODE_extraInputs,[0,10],[10,1,2],[],p,q,r);
```

- The empty set takes the place of any options (solution tolerance) we might enforce.
- Additional inputs into the ODE function are specified at the end.

```
% Plotting system
p = plot(t,x(:,1),'k-',...
    t,x(:,2),'b-',...
    t,x(:,3),'r-');
set(p,'LineWidth',3)
set(gca,'FontSize',12)
axis square
xlabel('t (s)')
ylabel('pos. (m), vel. (m/s), acc. (m/s^2)')
legend({'Position (y(t))','Velocity (y''(t))','Acceleration
(y''''(t))'},'location','southoutside')
```



Example problem: creating an interactive physics simulator

In Topic 6, we analytically solved for the motion of a projectile. Here, we will solve consider the differential equations representing this motion, which can be obtained from a force balance:

$$a_y = \frac{d}{dt} v_y = -g$$

$$v_y = \frac{d}{dt} y$$

$$a_x = 0$$

$$v_x = \frac{d}{dt} x$$

Produce an interface that enables you to interactively "launch" a projectile.

Step 1: Plot the trajectory of the ball

```
% Specify angle and velocity of the ball
th = 45;
v = 20;
vx = v*cosd(th);
vy = v*sind(th);
```

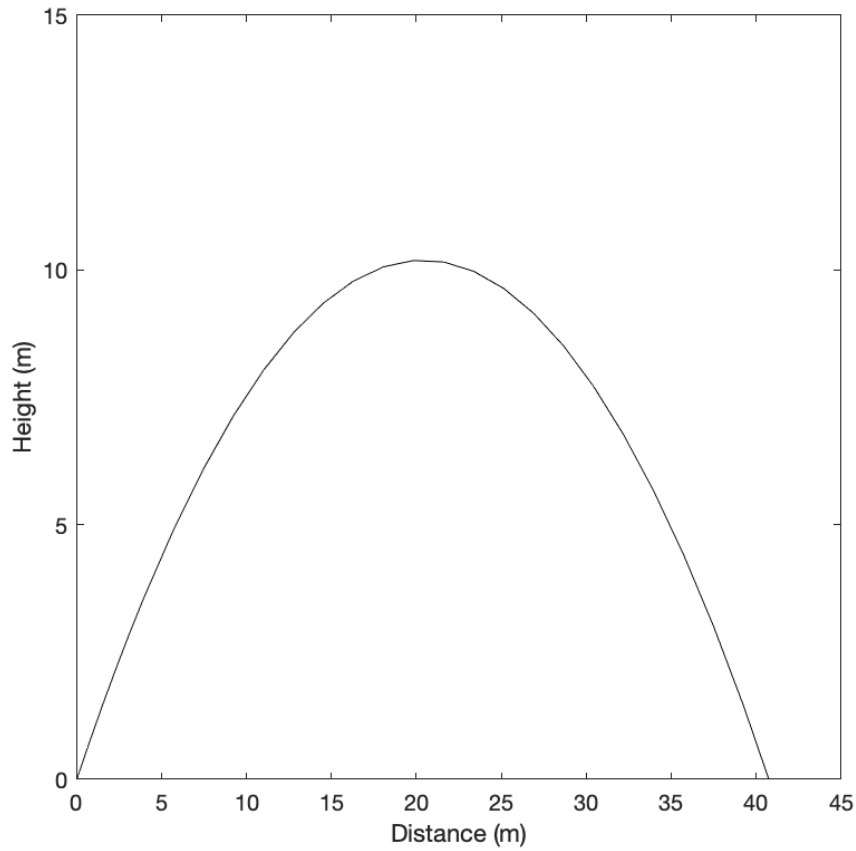
Writing the function to describe the ODEs:

```
function dz = ball_2D(t,z,vx)
    % Compute the x and y position of a ball subject to gravity and
    % in a vacuum. This function is called upon by MATLAB's ODE
    % solvers. t = time (s), z = [y-position, y-velocity, x-position]
    % in units of (m, m/s, m), and dz = [y-velocity; y-acceleration;
    % x-position] in units of (m/s, m/s^2, m).

    dz(1,1) = z(2); % z(1): y-position
    dz(2,1) = -9.81;% z(2): y-velocity
    dz(3,1) = vx;   % z(3): x-position
end
```

<u>Important note</u>: even though x and y are not explicitly related, **they describe the same system and must be incorporated into <u>one set of ODEs</u>.** If you (erroneously) call separate instances of ode45 for each, your solution is subject to the time points the solver selects and <u>these time points may not overlap.</u>

```
% Solving the set of ODEs. Note the additional input (x velocity)
[~,state] = ode45(@ball_2D,[0,5],[0,vy,0],[],vx);
x_ode = state(:,3); % x values
y_ode = state(:,1); % y values

figure
% Plot the height as a function of time
plot(x_ode,y_ode,'k-')
ylim([0,15])
xlabel('Distance (m)')
```

```
ylabel('Height (m)')
axis square
```



Step 2: visualize trajectory of "moving ball"

```
% Phyiscal parameters, as before
th = 60;
v = 15;
vx = v*cosd(th);
vy = v*sind(th);
g = 9.81;
% Estimate time to landing
t_end = 1.5*2*vy/g;

% Solve system of differential equations. Specify the time points to solve
% for
[t,state] = ode45(@ball_2D,[0:0.05:t_end],[2,vy,0],[],vx);
txy = [t,state(:,3),state(:,1)];
% Select time points corresponding to the ball not yet landing
```

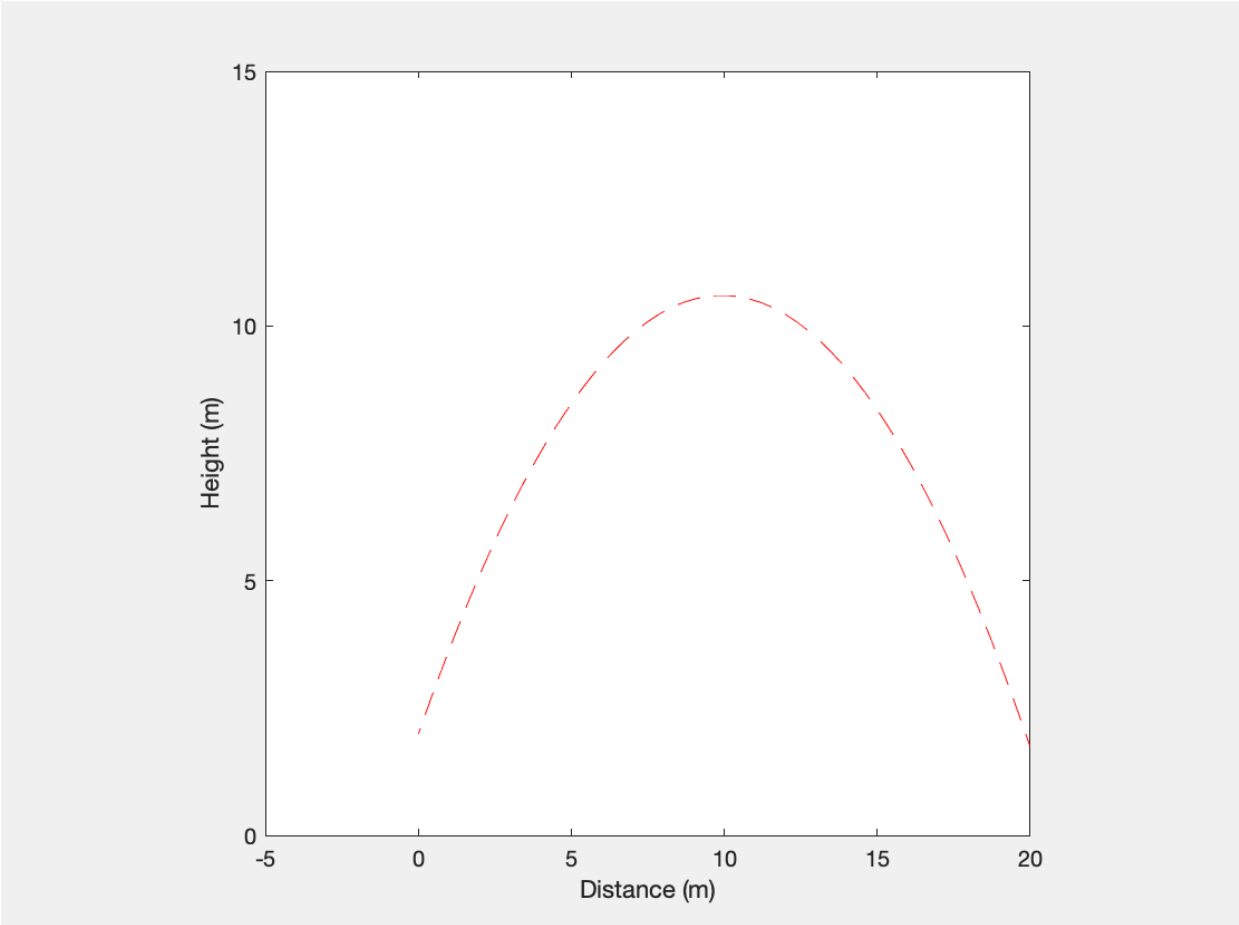
```
txy = txy(txy(:,3) > 0,:);
```

Note: we explicitly specify the time points to be solved for in the input to ode45.

To produce visual motion. we will use the following algorithm:

1. Solve the system of ODEs to obtain the ball's x and y position as a function of time.
2. Select data points that correspond to the ball having not landed
3. In each iteration, replot the ball position and add its position to the traveled points (trajectory)

```
close all

% Produce figure. Specify the plots of interest and the graphical
% parameters
h1 = figure('Visible','on');
hTrajectory = plot(txy(1,2),txy(1,3),'r--');
hold on
hBall = plot(txy(1,2),txy(1,3),'ro','MarkerFaceColor','r','MarkerSize',8);
hold off
ylim([0,15])
xlim([-5,20])
xlabel('Distance (m)')
ylabel('Height (m)')
axis square

% Loop through all ball positions and update plot
for k = 2:size(txy,1)

    % Update plot of ball and trajectory tracer
    hBall.XData = txy(k,2);
    hBall.YData = txy(k,3);
    hTrajectory.XData(k) = txy(k,2);
    hTrajectory.YData(k) = txy(k,3);

    % Pause for specified amount of time consisted with time points solved
    % for
    pause(0.05);

end
```



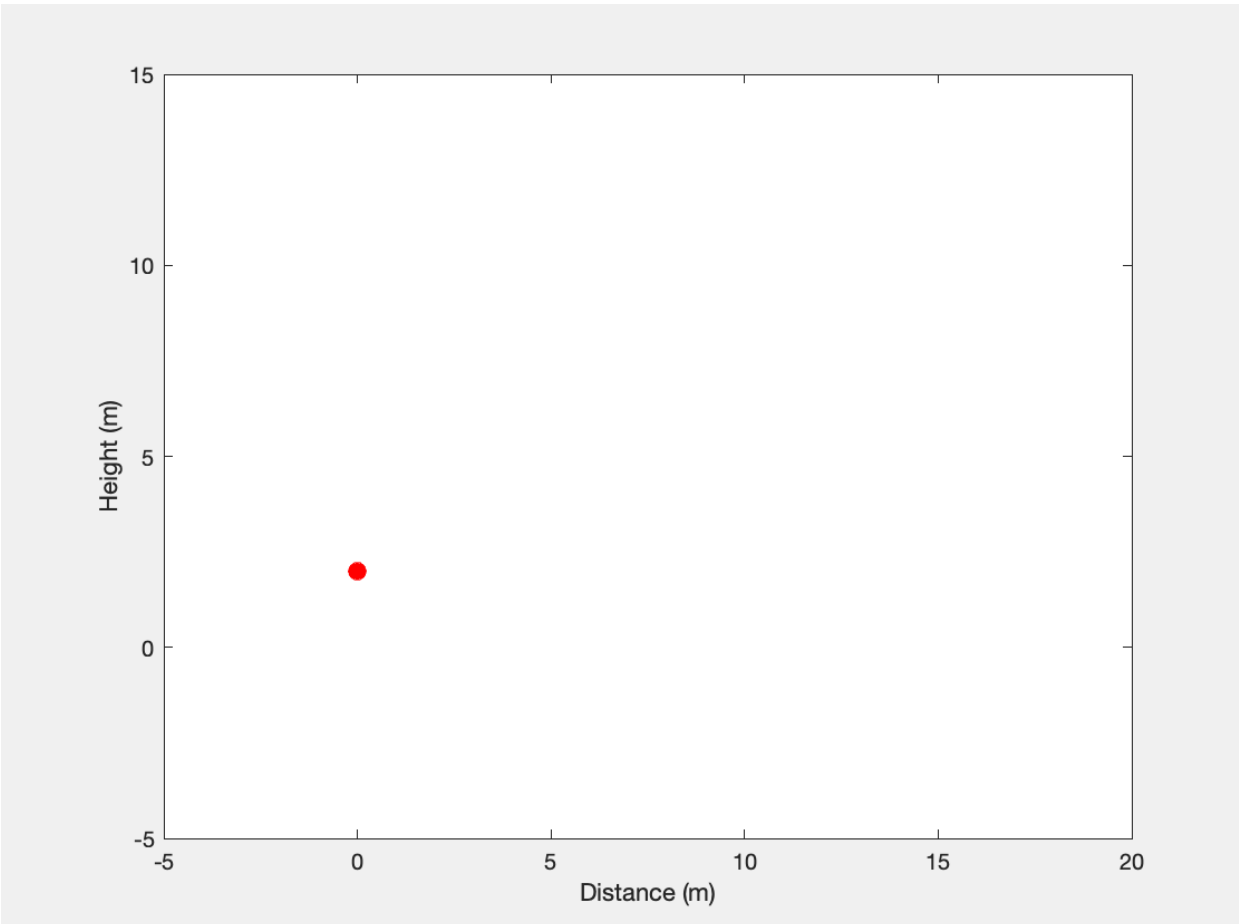
Step 3: produce an interface

Qualitatively, we want the following:

1. The location where the user clicks will determine both the angle and initial velocity of the ball.
2. Upon releasing the click, we wish to solve the ODE based on the initial parameters specified by the user click.
3. Upon releasing the click, we also to visualize the trajectory of the ball.

```
close all
% Create a figure
h1 = figure('Visible','on');

% Create and store GUI data structure
handles.h0 = 2;
guidata(h1,handles);

% Define plots for trajectory, ball, and aiming line
hTrajectory = plot(0,handles.h0,'r--');
hold on
```

```
hBall = plot(0,handles.h0,'ro','MarkerFaceColor','r','MarkerSize',8);
hAim = plot(0,handles.h0,'k-','LineWidth',2);
hold off

% Graphical parameters
ylim([-5,15])
xlim([-5,20])
xlabel('Distance (m)')
ylabel('Height (m)')
```




```
% Assign callback function for mouse click
h1.WindowButtonDownFcn = {@determineStart,hBall,hTrajectory,hAim};

% Assign callback function for mouse release
h1.WindowButtonUpFcn = {@throwBall,hBall,hTrajectory};
```

TOPIC 24: Symbolic variables, equations, and functions

ENGR105, 11/24/20

Note: for these notes, you will need the **`Symbolic Math Toolbox`** installed.

Symbolic math in MATLAB

- According to Mathworks, LiveScript documents are the recommended documents for performing symbolic operations.
- Symbolic math in MATLAB is based on the MuPAD computational engine.
- Reference documentation on changing output from symbolic operations
- `help symbolic` for MATLAB help documentation

In addtion to its capabilities with numerical processing, MATLAB provides built-in capabilities for solving, plotting, and manipulating symbolic math equations. **A main difference** between data types `symbolic` and `double` is that symbolic numbers are **exact**, whereas double values have **floating point precision** (see Lecture 15 / Topic 14).

Example. Calculate $\sin(\pi)$

```
% Using the numerical calculation, as before
sin(pi)
```

```
ans = 1.2246e-16
```

```
% Using symbolic
sin(sym(pi))
```

ans =

0

Here, we create a *symbolic number* by using `sym`.

Important note: while the numeric calculation does not return the exact result, it much more accurate than what we typically "need."

Declaring symbolic numbers and variables

You can create symbolic variables by using either `syms` or `sym`, or by assigning the **output from a symbolic expression** to a new variable.

- `syms` - use `syms` to symbolic **variables** that can be manipulated in symbolic expressions.

```
% Declare symbolic variables
syms x y
% Inspect properties of variables
whos x y
```

```
  Name      Size            Bytes  Class     Attributes

  x         1x1                 8  sym
  y         1x1                 8  sym
```

- `sym` – use `sym` to create **symbolic numbers**

```
% Create symbolic numbers
sym(5)
```

ans =
5

```
sym(pi)
```

ans =
π

```
sym(sqrt(2))
```

ans = $\sqrt{2}$

```
1+sym(pi)
```

ans = $\pi + 1$

Note: use sym on the "innermost" number only to create an exact number. Otherwise, the sym operator will be applied to the product of a floating point (numerical) operation.

```
% Correct: convert pi to symbolic, then operate on it.
sqrt(sym(pi))
```

ans = $\sqrt{\pi}$

```
% Incorrect: Convert the output from sqrt(pi) to a symbolic representation
sym(sqrt(pi))
```

ans =
$$\frac{3991211251234741}{2251799813685248}$$

- Assign an output from symbolic manipulations to a new variable.

```
% Create symbolic variable phi, also known as the "golden ratio"
phi = (1 + sqrt(sym(5)))/2
```

phi =
$$\frac{\sqrt{5}}{2} + \frac{1}{2}$$

Note: a symbolic number can be converted to double by using the `double` command

```
% Convert symbolic to double
```

```
double(phi)
```

```
ans = 1.6180
```

Symbolic expressions and functions

The above variable, phi, is created using a symbolic expression. You can now perform subsequent operations on `phi`.

```
% Operation 1
f1 = phi^2 - phi - 1
```

f1 =

$$\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^2-\frac{\sqrt{5}}{2}-\frac{3}{2}$$

```
% Operation 2
f2 = phi^2 + 1
```

f2 =

$$\left(\frac{\sqrt{5}}{2}+\frac{1}{2}\right)^2+1$$

Simplifying expressions. Use `simplify` to simplify an expression

```
% Simplify f1
simplify(f1)
```

ans =

$$0$$

```
% Simplify f2
simplify(f2)
```

ans =

$$\frac{\sqrt{5}}{2}+\frac{5}{2}$$

Symbolic expressions with symbolic variables are useful when you wish to perform mathematical manipulations on an equation.

```
% Create original equation
syms x
g = x^3 + 6*x^2 + 11*x + 6
```

g = $x^3+6\,x^2+11\,x+6$

```
% Manipulate equation by dividing it by (x+3)
```

```
g2 = g/(x+3)
```

g2 =

$$\frac{x^3 + 6\,x^2 + 11\,x + 6}{x + 3}$$

```
% Simplify equation
g2 = simplify(g2)
```

g2 = $x^2 + 3\,x + 2$

Note: to substitute a number in for a variable, you must use the `subs` command

```
% Parenthesis cannot be used to substitute for a variable
g2(1)
```

ans = $x^2 + 3\,x + 2$

```
% Instead, use subs to evaluate for particular values
subs(g2,x,1)
```

ans =

6

```
% Note: the original equation is unchanged
g2
```

g2 = $x^2 + 3\,x + 2$

Symbolic functions. If you intend to perform repeated evaluations, create a **symbolic function** instead. This can be declared using the syms function.

```
clear
syms f(x)
% Specify function. NOTE: must define f(x)
f(x) = x^2-1+sqrt(sym(pi));
% the function f(x) will be displayed if either f or f(x) is called
disp(f(x))
```

$x^2 + \sqrt{\pi} - 1$

```
disp(f)
```

$x^2 + \sqrt{\pi} - 1$

```
% In this case, direct substitution can occur with parenthesis
f(1)
```

ans = $\sqrt{\pi}$

```
f(2)
```

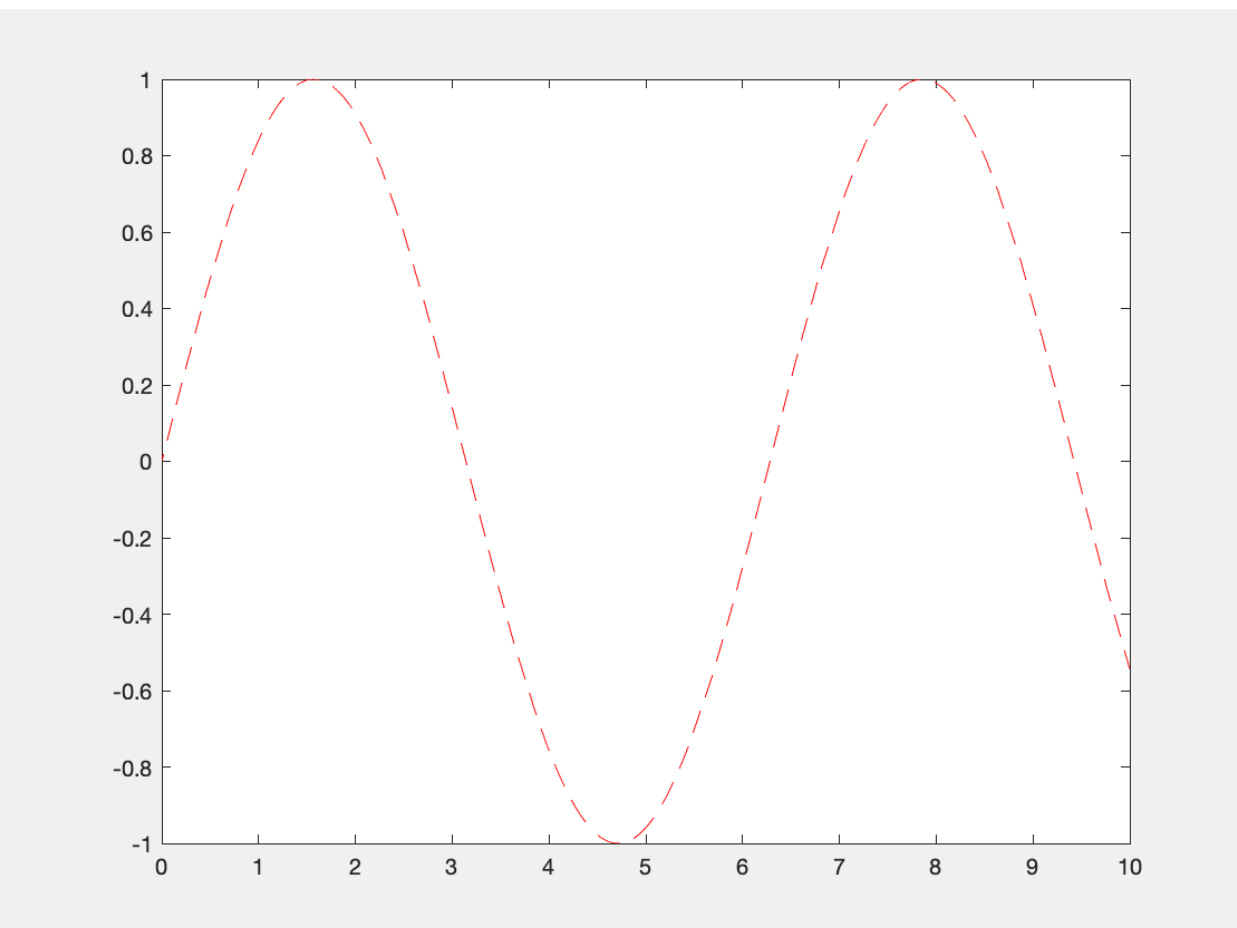
ans = $\sqrt{\pi} + 3$

Plotting symbolic functions

Plot symbolic equations using `fplot`. The most common method for plotting symbolic functions is to use `fplot()`. `fplot()` takes the following inputs:

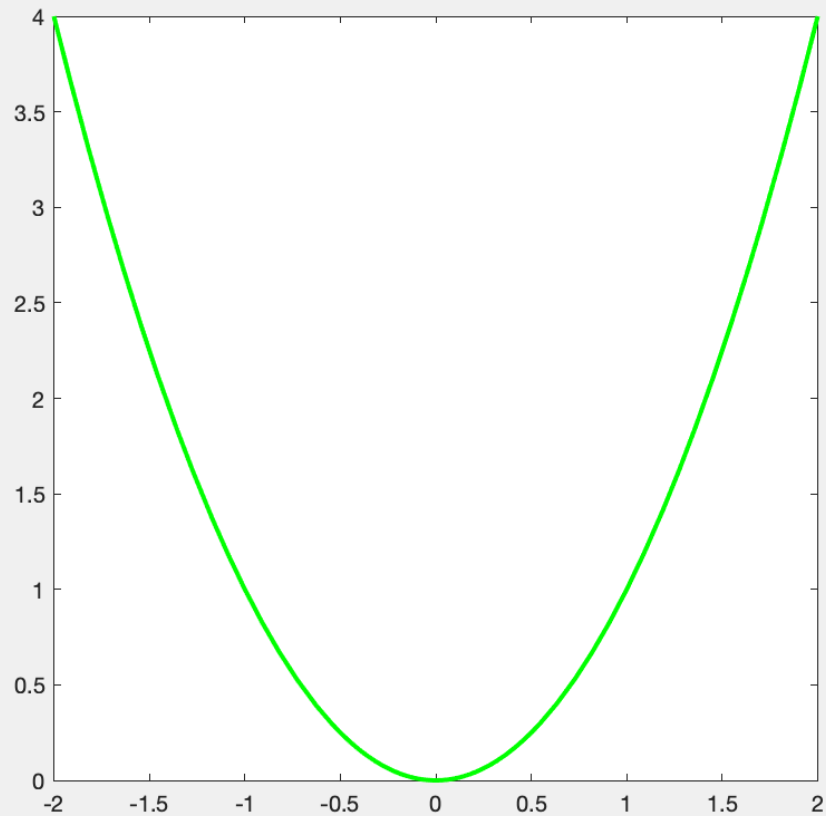
- *First input*: function to be plotted
- *Second input*: time domain,
- *Third input*: line specifications

```
clear
% Declare symbolic variables
syms x
% Plot sin(x) over the domain [0,10]
fplot(sin(x), [0,10], 'r--')
```



Plots of symbolic functions can also be modified as **graphics handle objects**.

```
% Declare symbolic variable
syms t

% Plot function and specify parameters
h1 = fplot(t^2,[-2,2]);
h1.LineWidth = 2;
h1.Color = 'g';
axis square
```



<u>Important note</u>: symbolic functions are **<u>not</u>** plotted with infinite precision. When you plot a symbolic function, the following happens:

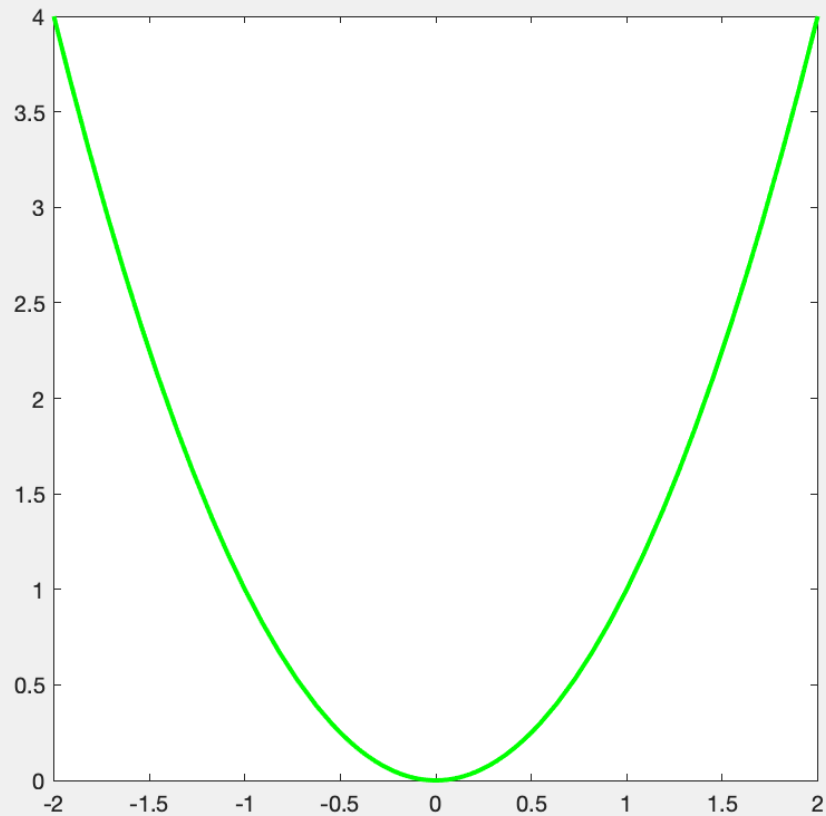
1. MATLAB automatically **creates vectors of x and y data** spanning the specified domain.
2. MATLAB **plots those two vectors** and assigns the specified formatting.

The x and y data that is created can be accessed using the graphics handle object.

```
% Acquire data vectors that were plotted to create the above plot.
xdata = h1.XData;
ydata = h1.YData;
```

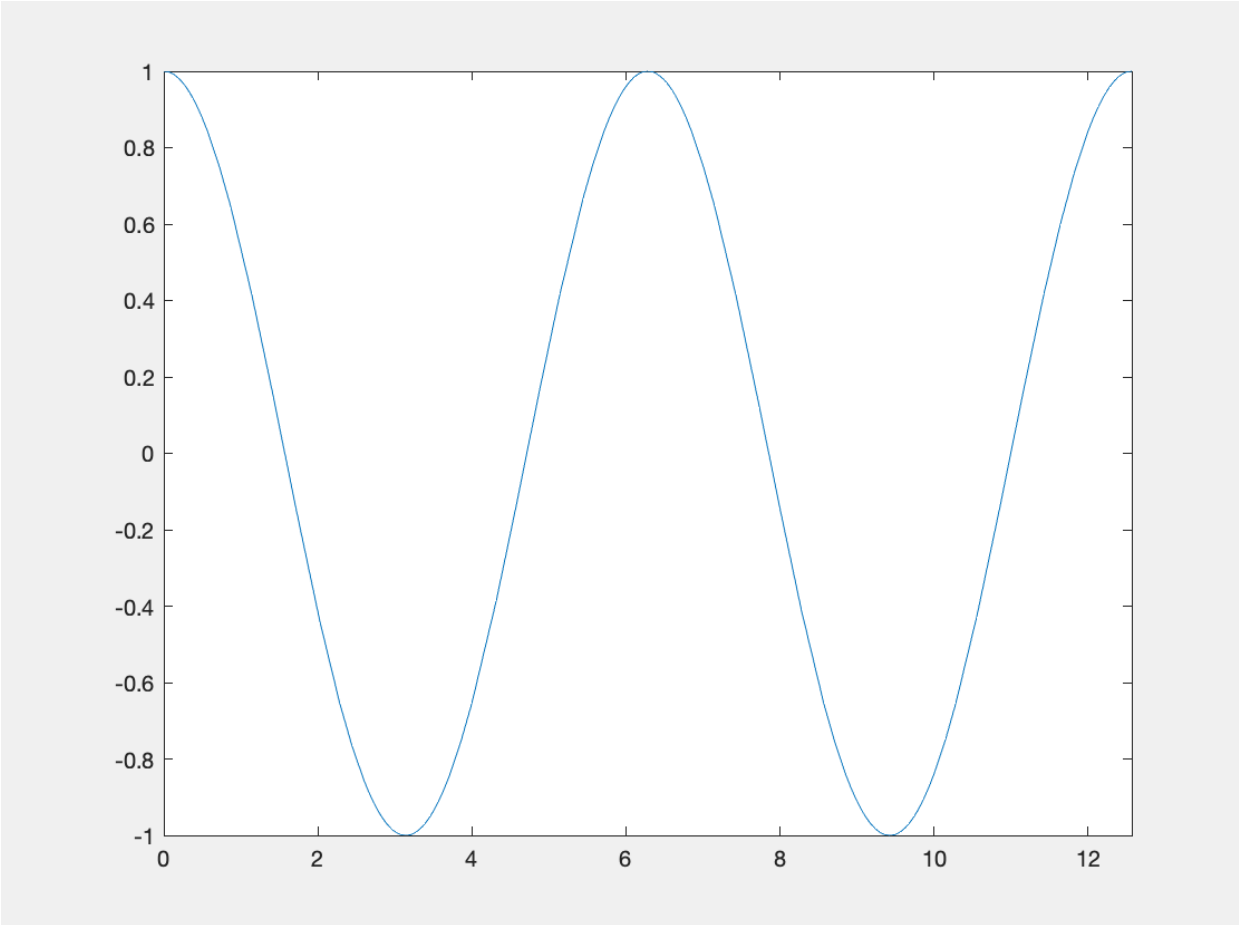
Thus, you could produce an **equivalent plot** using the standard `plot()` function.

```
% Equivalent plot
plot(xdata,ydata,'g-','LineWidth',2)
axis square
```



<u>Note</u>: you can also plot **anonymous functions** using `fplot`.

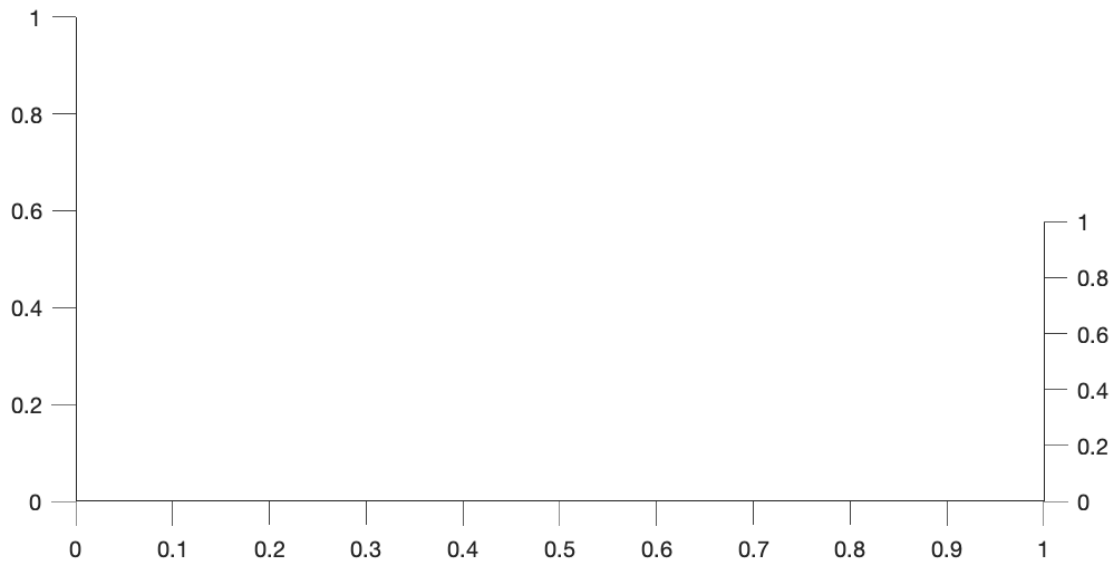
```
clear
% Produce a plot of cos(x) between 0 and 4*pi
fplot(@(x) cos(x),[0,4*pi])
```



Plot symbolic equations in 3D using `fplot3`. The symbolic analog to `plot3()` is `fplot3()`, which can be used to plot parametric curves.

Example. plot the curve given by $x(t) = t,\ y(t) = \sin(t),\ z(t) = \cos(t)$ for $t \in [-10, 10]$

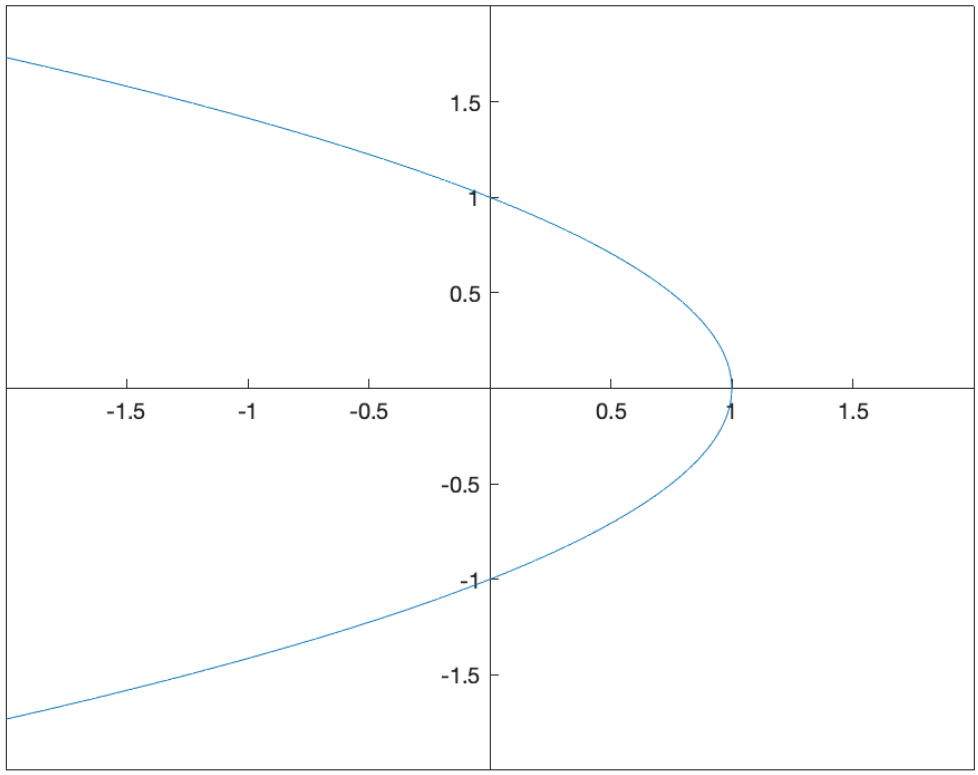
```
% Declare symbolic variable
syms t
% Produce a 3D plot corresponding to the above
fplot3(t,sin(t),cos(t),[-10,10])
xlabel('x')
ylabel('y')
zlabel('z')

% Add constant rotation to the plot
for k = 1:360
    % Update the view point
    view(k,30)
    % Pause MATLAB for 100 ms
    pause(0.1)
end
```



Plot implicit relationships using `fimplicit`. When you cannot easily solve for one variable in terms of another, you can plot the implicit form of the relationship.

Example. Plot $1 = y^2 + x$

```
syms x y
fimplicit(1==y^2+x,[-2,2])
set(gca,'XAxisLocation','Origin')
set(gca,'YAxisLocation','Origin')
```



Notes:

- The variable that comes **first** in alphanumeric order is assigned to the **x variable**.
- The above example specifies the **x domain** to be plotted over.

Solving equations of one variable

Exact solutions to an equation may be obtained using the `solve()` function.

```
clear
syms x   % Declare symbolic variable
% Example 1
solve(5*x == 1)
```

ans =

$$\frac{1}{5}$$

```
% Example 2: multiple solutions are present
solve(x^2 - 1 == 0)
```

ans =

$$\begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

```
% Example 3: can obtain complex solutions
solve(x^2 + 1 == 0)
```

ans =

$$\begin{pmatrix} -i \\ i \end{pmatrix}$$

Solving sets of linear equations

Consider the system of two equations used previously in Lecture 4:

$$\begin{aligned} y &= x \\ y &= 2x - 3 \end{aligned}$$

To solve the set of equation, each equation can be assigned to a symbolic variable.

```
clear
syms x y % declare variables as symbolic
eqn1 = y==2*x-3;    % first equation
eqn2 = y==x;    % second equation
```

Note: the double equals (==) says that two symbolic expression are equal to one another.

Use `solve()` to determine the **solution to the system of equations**. In this case, the output `res` is a data structure. The "`.x`" and "`.y`" are simply used to call the solution value to each variable.

```
res = solve([eqn1,eqn2],[x,y]);
xSol = res.x % retrieve the x solution
```

xSol =

3

```
ySol = res.y % retrieve the y solution
```

ySol =

3

Comparison to linear algebra. Consider the set of linear equations from **example 2** in **Lecture 3 / Topic 4.**

$$\begin{aligned} x_1 + 3x_2 + x_3 &= 1 \\ 2x_1 + 2x_2 - x_3 &= 2 \\ x_1 + x_2 + x_3 &= -2 \end{aligned}$$

Solve the following system of equations using symbolics and linear algebra.

1) Using symbolic variables

```
tic
syms x1 x2 x3
% Define equations
eqn1 = x1 + 3*x2 + x3 == 1;
eqn2 = 2*x1 + 2*x2 - x3 == 2;
eqn3 = x1 + x2 + x3 == -2;
% Solve set of linear equations
res = solve([eqn1,eqn2,eqn3],[x1,x2,x3]);
% Display the solution for each variable
disp(res.x1)
```

$$-\frac{3}{2}$$

```
disp(res.x2)
```

$$\frac{3}{2}$$

```
disp(res.x3)
```

$$-2$$

```
toc
```

```
Elapsed time is 0.227022 seconds.
```

2) Using linear algebra

```
tic
% Step 1: convert to matrix format
A = [1,3,1;2,2,-1;1,1,1]; %Coefficient matrix defined above
b = [1;2;-2]; % Constant matrix
% Step 2: solve for x
x = A\b
```

```
x = 3×1
   -1.5000
    1.5000
   -2.0000
```

```
toc
```

```
Elapsed time is 0.022148 seconds.
```

Takeaway: both work fine, but even for this *simple* set of 3 equations, the linear algebra approach is quicker by 1-2 orders of magnitude! **You will have to use linear algebra to solve large sets of equations**.

Solving sets of nonlinear equations

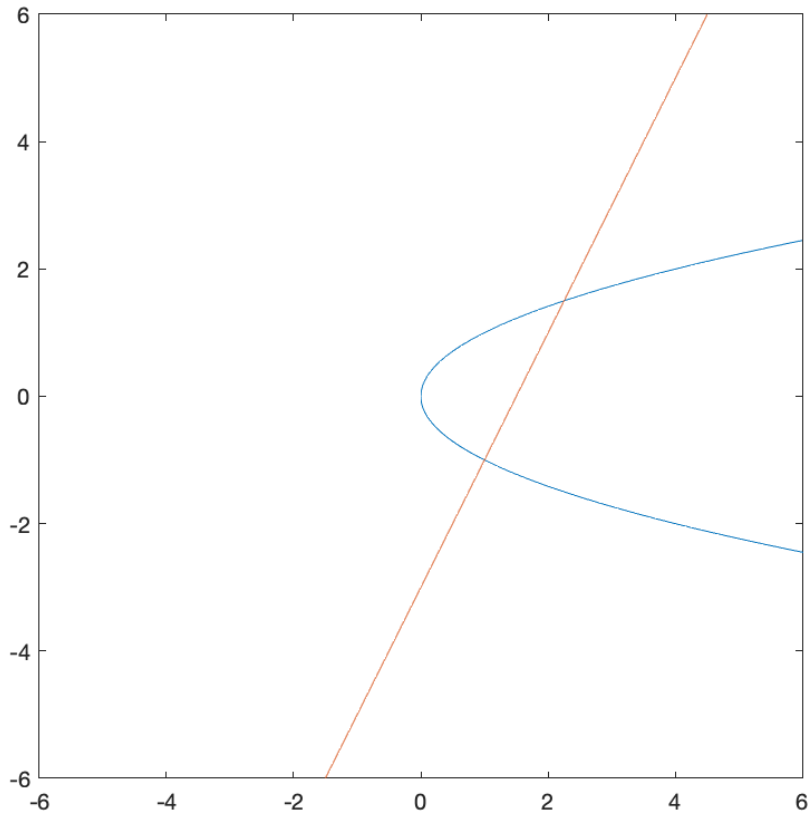
Sets of (simple) nonlinear equations can also be readily solved using the `solve()` function. Consider the following set of equations:

$$y^2 = x$$
$$y = 2x - 3$$

Especially when solving sets of nonlinear equations, it is advisable to plot them to get an idea of where the solutions should be.

- Use `fimplicit()` to plot symbolic equations in the form we defined above.

```
% Declare variables and equations
syms x y
eqn1 = y^2==x;
eqn2 = y==2*x-3;
% Plot equations on the interval for x and y of [-6,6]
fimplicit([eqn1,eqn2],[-6,6,-6,6])
axis square
```



- As before, we can solve the system of equations using `solve()`.

```
% Solve system of nonlinear equations
res2 = solve([eqn1,eqn2],[x,y]);
xSol2 = res2.x % retrieve the x solution
```

xSol2 =

$$\begin{pmatrix} 1 \\ \frac{9}{4} \end{pmatrix}$$

```
ySol2 = res2.y % retrieve the y solution
```

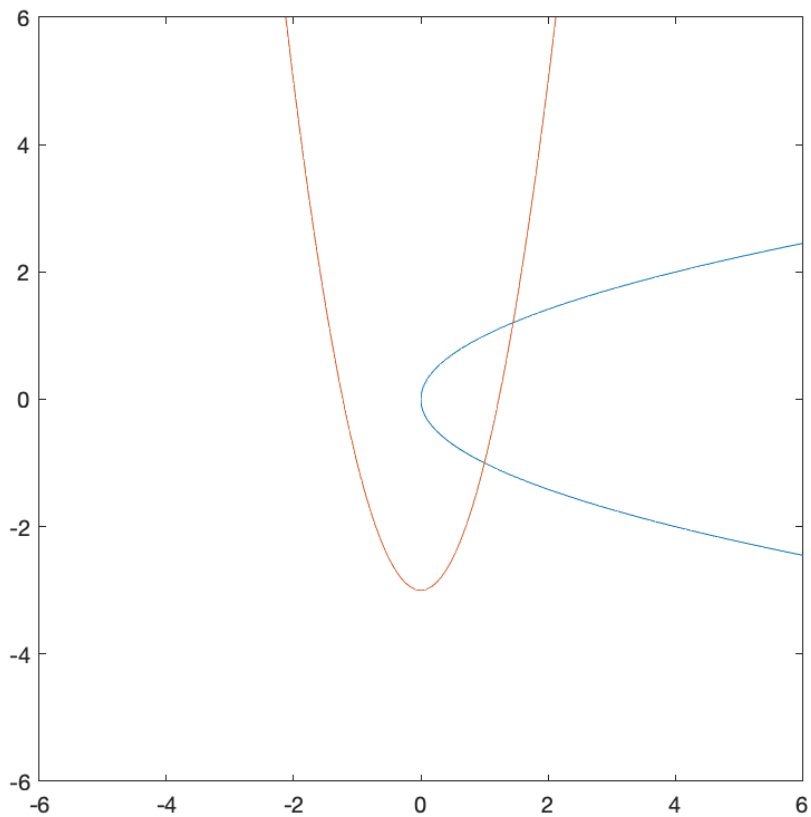
ySol2 =

$$\begin{pmatrix} -1 \\ \frac{3}{2} \end{pmatrix}$$

Comparison to numerically solving. Consider the following, slightly more nonlinear set of equations. We can plot it by running the same code as before.

$$y^2 = x$$
$$y = 2x^2 - 3$$

```
% Declare variables and equations
syms x y
eqn1 = y^2==x;
eqn2 = y==2*x^2-3;
% Plot equations on the interval for x and y of [-6,6]
fimplicit([eqn1,eqn2],[-6,6,-6,6])
axis square
```



1) Solve using symbolic math

```
% Solve system of nonlinear equations
res3 = solve([eqn1,eqn2],[x,y]);
xSol3 = res3.x % retrieve the x solution
```

```
xSol3 =
```

$$\begin{pmatrix} 1 \\ \mathrm{root}\left(z^3 - z^2 + z - \frac{3}{2}, z, 1\right)^2 \\ \mathrm{root}\left(z^3 - z^2 + z - \frac{3}{2}, z, 2\right)^2 \\ \mathrm{root}\left(z^3 - z^2 + z - \frac{3}{2}, z, 3\right)^2 \end{pmatrix}$$

```
ySol3 = res3.y % retrieve the y solution
```

ySol3 =

$$\begin{pmatrix} -1 \\ \mathrm{root}\left(z^3 - z^2 + z - \frac{3}{2}, z, 1\right) \\ \mathrm{root}\left(z^3 - z^2 + z - \frac{3}{2}, z, 2\right) \\ \mathrm{root}\left(z^3 - z^2 + z - \frac{3}{2}, z, 3\right) \end{pmatrix}$$

The solver returns the **exact** solution, which isn't necessarily useful.

2) Solve using numerical method. Using `fsolve` as before, we can solve this set of nonlinear equations.

```
% Determine solution 1
sol_1 = fsolve(@eqnSet,[0,0]);
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
```

```
% Determine solution 2
sol_2 = fsolve(@eqnSet,[1,1]);
```

```
Equation solved.

fsolve completed because the vector of function values is near zero
as measured by the value of the function tolerance, and
the problem appears regular as measured by the gradient.

<stopping criteria details>
```

```
disp(sol_1)
```

```
    1.0000   -1.0000
```

```
disp(sol_2)
```

```
    1.4498    1.2041
```

Takeaway: `solve()` is trying to find an exact solution, which is not always very useful. Often a numerical solution is *more than accurate enough* for a given application.

For more information on modifying outputs from symbolic manipulations, see the reference documentation from MATLAB

Calculus

Univariate symbolic differentiation can be accomplished by using the `diff` command.

- By directly inputting a symbolic function

```
syms x
diff(sin(x), x)
```

ans = $\cos(x)$

- Assigning an expression to new variable

```
F = sin(x);
diff(F,x)
```

ans = $\cos(x)$

- Calculating a second derivative

```
diff(F,x,2)
```

ans = $-\sin(x)$

Multivariate symbolic differentiation using the `diff` command

- Differentiation with respect to *x*

```
syms x y
diff(sin(x)*cos(y),x)
```

ans = $\cos(x)\cos(y)$

- Differentiation with respect to y

```
diff(sin(x)*cos(y),y)
```

ans = $-\sin(x)\sin(y)$

- Differentiate with respect to x and then y

```
diff(x*sin(x*y), x, y)
```

ans = $2\,x\cos(x\,y) - x^2\,y\sin(x\,y)$

Integration. Symbolic integration/antidifferentiation may be achieved using the `int()` function.

```
syms x % declare x symbolic
% Integrate x^2 + x
res1 = int(x^2+x)
```

res1 =

$$\frac{x^2\,(2\,x+3)}{6}$$

```
% Expand the polynomial
res1 = expand(res1)
```

res1 =

$$\frac{x^3}{3}+\frac{x^2}{2}$$

Integrating a multivariate function with respect to *y* and then *x*

```
syms x y z
res = expand(int(int(x^2+x+y^3+z^(1/2),y),x))
```

res =

$$\frac{x^2\,y}{2}+\frac{x^3\,y}{3}+\frac{x\,y^4}{4}+x\,y\,\sqrt{z}$$

<u>Note</u>: the variable to be integrated by is contained within each function call of `int()`.

Definite integration using `int()`. `int()` achieves numerical integration by evaluating the values of the indefinite integrals at each bound, equivalent to the following.

$$\int_{\pi/4}^{\pi/2}\sin(x)\,dx = \cos(x)\Big|_{x=\pi/2} - \cos(x)\Big|_{x=\pi/4}$$
$$= 0 - \sqrt{2}/2 \approx .7071$$

```
syms theta
res = int(sin(theta),[pi/4 pi/2])
```

res =

$$\frac{\sqrt{2}}{2}$$

Some functions do not yield a useful analytical form under integration - i.e. MATLAB returns a solution in integral notation.

```
syms x
int(sin(sinh(x)), x)
```

ans =

$$\int \sin(\sinh(x))\,\mathrm{d}x$$

This highlights that some functions can only be **numerically integrated** (using numerical integration methods) or approximated using a Taylor expansion

Differential equations

Differential equations can be solved using the `dsolve` function. This function takes an input of symbolic variable corresponding to the differential equation to be solved.

Solving first order ODEs. The `diff` function (used for differentiating symbolic expressions) is used to create the differential equation.

```
clear
% Declare symbolic variables
syms y(t) a
% Define differential equation
eqn = diff(y,t) == a*y;
S = dsolve(eqn)
```

S = $C_1\,e^{a\,t}$

<u>Note</u>: we **do not specify initial conditions** and so MATLAB produces a *general solution* to the ODE.

Initial conditions can be provided as a second input.

```
% Define initial conditions
cond = y(0) == 5;
ySol(t) = dsolve(eqn,cond)
```

ySol(t) = $5\,e^{a\,t}$

Second order ODEs can be created by using the `diff` function and specifying the second order derivative.

```
% Define second order ODE. % Note the additional input to diff() to create
% a second order derivative.
eqn = diff(y,t,2) == a*y;
% Solve for general solution to differential equation
```

```
ySol(t) = dsolve(eqn)
```

ySol(t) = $C_1\,e^{-\sqrt{a}\,t} + C_2\,e^{\sqrt{a}\,t}$

```
% Specify initial conditions

% Define the first derivative so that it can be used as an initial
% condition
Dy = diff(y,t)
```

Dy(t) =

$$\frac{\partial}{\partial t}\,y(t)$$

```
syms b
% Define vector containing initial conditions
cond = [y(0)==b, Dy(0)==1];
% Solve for the specific form of the ODE solution
ySol(t) = dsolve(eqn,cond)
```

ySol(t) =

$$\frac{e^{\sqrt{a}\,t}\,(\sqrt{a}\,b+1)}{2\,\sqrt{a}} + \frac{e^{-\sqrt{a}\,t}\,(\sqrt{a}\,b-1)}{2\,\sqrt{a}}$$

Solving sets of differential equations. Explicit solutions to sets of ODEs can be solved by inputting a vector of ODEs into `dsolve`.

```
% Define ODEs
syms y(t) z(t)
% Define vector of ODEs
eqns = [diff(y,t) == z, diff(z,t) == -y];
% Find solution
S = dsolve(eqns);
S.z
```

ans = $C_2\cos(t) - C_1\sin(t)$

```
S.y
```

ans = $C_1\cos(t) + C_2\sin(t)$

Assigning initial conditions

```
S = dsolve(eqns,[y(0) == 1,z(0) == 0]);
S.z
```

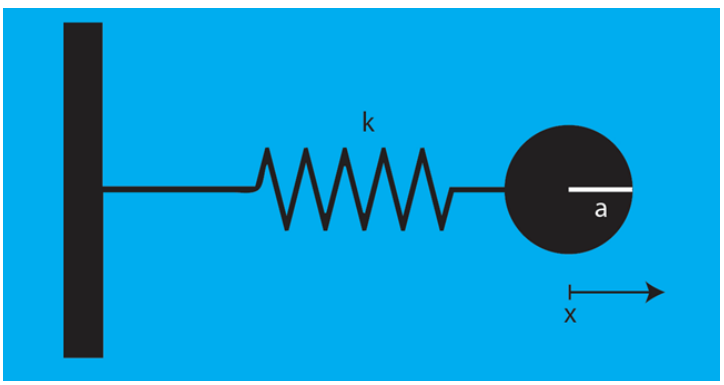
ans = $-\sin(t)$

```
S.y
```

ans = $\cos(t)$

Example problem: Harmonic oscillator

Consider a 1 cm (radius) carbon ball in water that is attached to linear spring, which is attached to a surface, as shown below. Assuming $x(t) = 0$ is the equilibrium position and the ball is pulled back to $x(0) = 2\,\text{cm}$ and released, plot the x position of the ball as a function of time for spring constants of $k_1 = 10^{-5}\,\frac{N}{m}$ and $k_2 = 10^{-6}\,\frac{N}{m}$.



Consider only 1-D motion in the x direction, neglect any forces in the y-direction (*e.g.* related to gravity), and ignore mass of the spring.

Step 1: Write down what we know and determine the differential equation describing the system.

Values that we know:

- Radius of ball: $a = 0.01\,m$
- Density of carbon: $\rho = 2,260\,\frac{kg}{m^3}$
- Viscosity of water: $\eta = 10^{-3}\,Pa \cdot s$

Creating the differential equation:

From our knowledge of **physics**, we know that the force exerted by a linear spring is directly proportional to the displacement of the spring. This corresponds to the well-known **Hooke's law**:

$$F_{\text{spring}}(t) = -kx(t)$$

From **fluid mechanics**, we know that the drag force exerted by a sphere in some fluid can be represented by **Stokes's law**. This relationship states that the drag force is directly proportional to the velocity of the sphere, the sphere's radius (a), and the viscosity of the fluid:

$$F_{\text{drag}}(t) = -6\pi a\eta v(t) = -6\pi a\eta \frac{dx}{dt}$$

From **Newton's second law of motion**, we can thus relate all forces acting on the object:

$$ma = m\frac{d^2x}{dt^2} = \rho\frac{4}{3}\pi a^3\frac{d^2x}{dt^2} = \sum_i F_i(t)$$

Simplifying this differential equation:

$$\rho\frac{4}{3}\pi a^3\frac{d^2x}{dt^2} + 6\pi a\eta\frac{dx}{dt} + kx(t) = 0$$

$$\frac{d^2x}{dt^2} + \frac{6\pi a\eta}{\rho\frac{4}{3}\pi a^3}\frac{dx}{dt} + \frac{k}{\rho\frac{4}{3}\pi a^3}x(t) = 0$$

Or, by definining and substituting variables b and c:

$$\frac{d^2x}{dt^2} + b\frac{dx}{dt} + cx(t) = 0$$

Step 2: Convert the above to code

```
%Step 1: Setup
%write down parameters

a = 0.01; % m
eta = 10^-3; % Pa*
rho = 2260;
k1 = 1*10^-5;
k2 = 1*10^-6;


%declare symbols
%x(t): position of ball
% b: coefficient in front of first derivative
% c: coefficient in front of x(t)
syms x(t) b c k

%coeffiecent in front each derivative
b = 6*sym(pi)*a*eta/(rho*4/3*sym(pi)*a/3);
c = k/(rho*4/3*sym(pi)*a^3);
```

Step 3a: solve differential equation for one value of k

```
% Define first derivative of x
Dx = diff(x,t);

eq = diff(Dx,t) + b*Dx +c*x == 0
```

eq(t) =

$$\frac{\partial^2}{\partial t^2}\, x(t) + \frac{27\,\frac{\partial}{\partial t}\, x(t)}{4520000} + \frac{37500\, k\, x(t)}{113\,\pi} = 0$$

```
%substitute in one vale
eq1 = subs(eq,k,k1)
```

eq1(t) =

$$\frac{\partial^2}{\partial t^2}\, x(t) + \frac{27\,\frac{\partial}{\partial t}\, x(t)}{4520000} + \frac{3\, x(t)}{904\,\pi} = 0$$

```
sol1 = dsolve(eq1,[x(0) == 0.2, Dx(0) == 0])
```

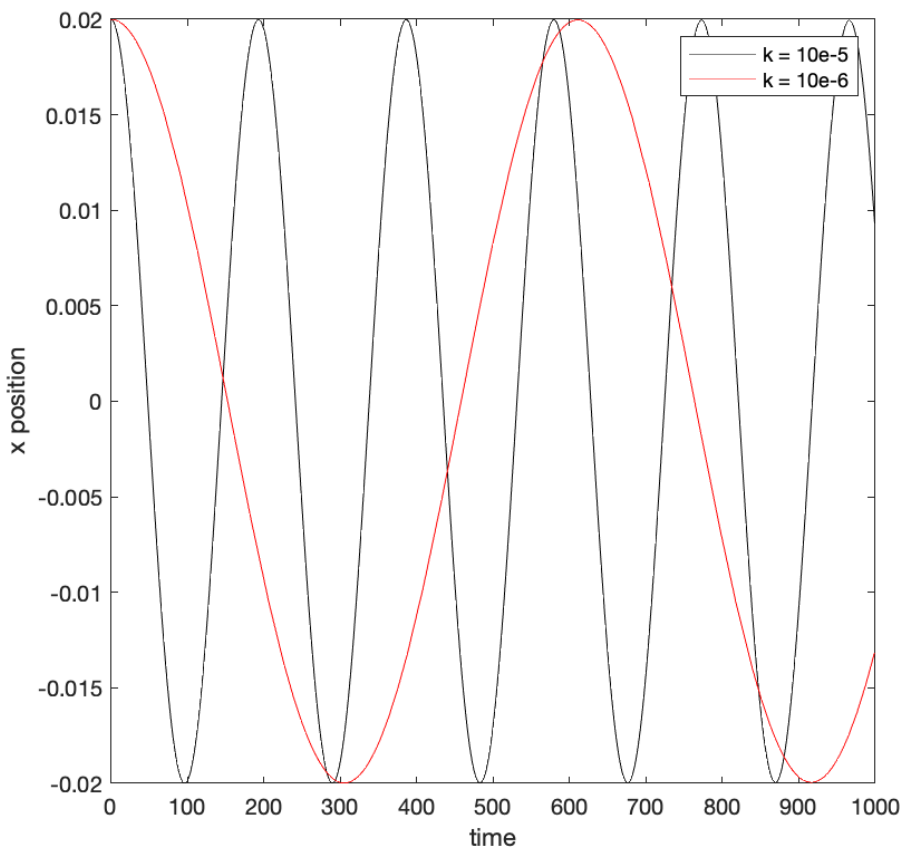
sol1 =

$$\frac{e^{-\frac{27\,t}{9040000}} \cos\left(\frac{\sqrt{3}\; t\; \sqrt{90400000000 - 243\,\pi}}{9040000\; \sqrt{\pi}}\right)}{5} + \frac{9\; \sqrt{3}\; \sqrt{\pi}\; e^{-\frac{27\,t}{9040000}} \sin\left(\frac{\sqrt{3}\; t\; \sqrt{90400000000 - 243\,\pi}}{9040000\; \sqrt{\pi}}\right)}{5\; \sqrt{90400000000 - 243\,\pi}}$$

Step 3: Solve the differential equation

```
% Solve differential equation
% IC: x(0) = 0.02
% IC: dx/dt(0) = 0 (the ball does not have any initial velocity
col = {'k','r'};
kval = [k1,k2];
for m = 1:2

    % Substitute in the variable
    eq1 = subs(eq,k,kval(m));

    % Solve the differential equation
    sol1 = dsolve(eq1, [x(0) == 0.02, Dx(0) == 0]);

    % Plot the solution over time
    fplot(sol1,[0,1000],col{m});
    hold on
```

```
end

hold off
legend('k = 10e-5', 'k = 10e-6')
ylim([-0.02, 0.02])
axis square
ylabel('x position')
xlabel('time')
```



Question: how can we solve the above using `ode45`?

Start by **changing variables** to convert the higher order ODE into a set of first order ODEs.

$$\frac{d^2x}{dt^2} + b\frac{dx}{dt} + cx(t) = 0$$

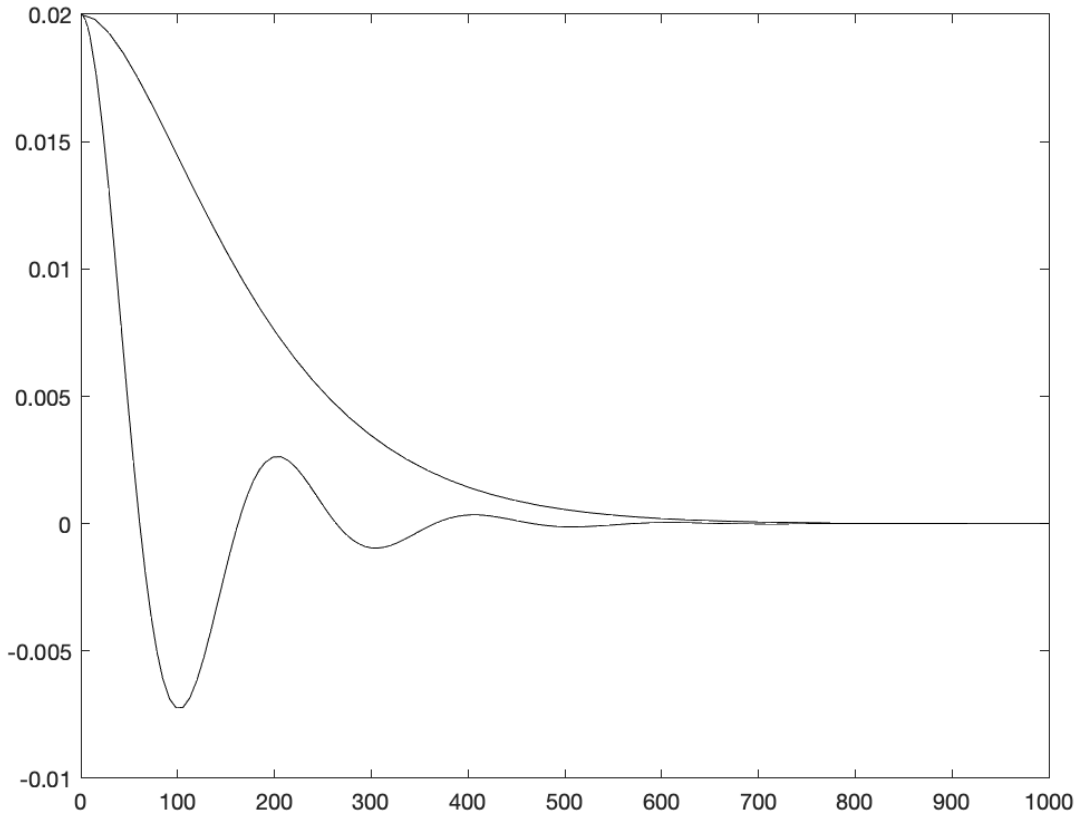
$$z_1 = x$$

$$z_2 = \frac{dx}{dt}$$

$$\frac{dz_2}{dt} = -bz_2 - cz_1$$

$$\frac{dz_1}{dt} = z_2$$

```
clear

% STEP 1: Setup
% Write down paramters describing the system

a = 0.01; % m
eta = 10^-3; % Pa*s
rho = 2260;
k1 = 1*10^-5;
k2 = 1*10^-6;

% Coefficient in front of each derivative
b = 6*(pi)*a*eta/(rho*4/3*(pi)*a^3);
c = [k1,k2]/(rho*4/3*(pi)*a^3);

for m = 1:2
    [t,out] = ode45(@springProblem,[0,1000],[0.02,0],[],b,c(m));

    plot(t,out(:,1),'k-')
    hold on
end
hold off
```



TOPIC 25: Animations

ENGR105, 11/30/20

Reminder: animating simple graphical objects

So far, we have produced visualizations and simulations involving the movement of simple **graphics handle objects**, such as **circles** and **squares**. For example, in Topic 20 we produced a particle-based simulation to represent the spread of infectious disease. The code for this is shown here.

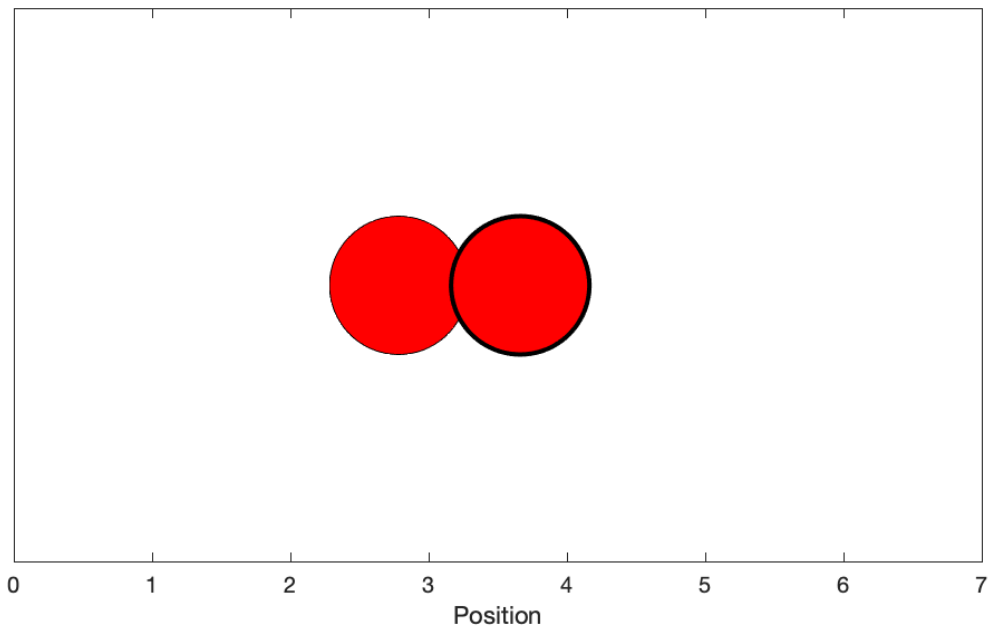
Key features:

- The objects that are moving are declared as **graphics handle objects**. The position and color of each is updated by changing the appropriate **fields** of the **handle**.
- The order of the code is important. In this case, it is important to first create the "background." That is, it is important to first create the axes, set the limits, etc. **In other cases, you may need to create an object first**. A useful troubleshooting approach is think about the order in which you create things.

```
close all
```

```
% 1) Vector representing positions of two particles
pos = [1,4];

% 2) Define particle identities.
% Particle 1 will be infected
% Particle 2 will initially be healthy

% 3) Produce visualization of each particle
g1 = axes;
axis image              % Setting this ensures that the x and y scales are
equal on the resulting image
xlim([0,7])             % X limits
ylim([-2,2])            % Y limits
set(g1,'ytick',[])      % Turn off y tick marks
set(g1,'Box','on')
xlabel('Position')      % X label

% Plot both particles initial position
p1 = rectangle('Position',[pos(1),-0.5,1,1],'Curvature',[1,1],'FaceColor','r');
p2 = rectangle('Position',[pos(2),-
0.5,1,1],'Curvature',[1,1],'FaceColor','w','LineWidth',2);

% Step through the simulation. In each step, update the particles positions
while abs(pos(1)-pos(2)) >= 1

    % A) Create vector determining particle step
    d_k = (rand(1,2)-0.5);

    % B) Update particle position
    pos = pos + d_k;

    % C) Ensure particle doesn't move out of bounds
    pos = max(min(pos,6),0);

    % If difference between particles is less than one diameter, an
    % infection has occurred
    if abs(pos(1)-pos(2)) < 1
        p2.FaceColor = 'r';
    end

    % D) Update visualization
    p1.Position(1) = pos(1);
    p2.Position(1) = pos(2);

    % E) Pause MATLAB
```

```
    pause(0.2)
end
```



Thoughts on creating more sophisticated objects from combinations of objects:

- It is possible to group objects together.
- However, remember that a graphics handle object (whether an individual or group) can only be modified by changing *fields* of the handle object. When in doubt, use `get()` to retrieve possible fields to change.
- If a field is not available (e.g. for a group), you will need to change the desired field for each of the individual objects.

<u>Takeaway</u>: use `get()` to help troubleshoot possible modifications to an object.

Reminder: the flow of information in an interface

Furthermore, remember that consideration must be put into *how* information is transferred between objects and functions within an interface object.

Graphics objects we've discussed so far:

1. `figure` objects

2. `axes` objects
3. `plot`/`bar`/(data visualization) objects
4. `rectangle` objects
5. `uicontrol` objects
6. `text` objects
7. `legend` objects
8. `ylabel, xlabel, title, xlim, ylim` objects
9. Others?

- **Rule of thumb**: *if it creates something visual on a graph, plot, figure window, etc., then it can be stored and modified as a graphics handle object.*

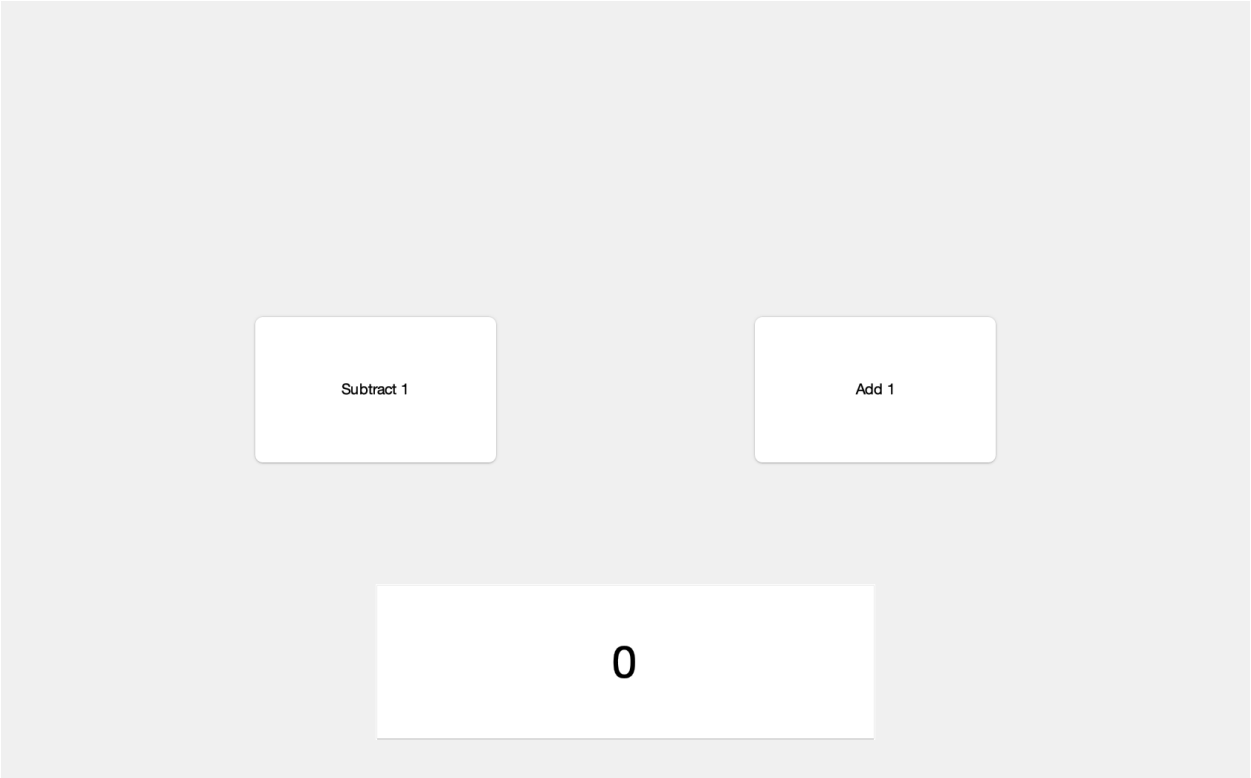
Interfaces are based on a `figure` object. The figure is the parent that *contains* axes, uicontrols, text, callback functions, etc.

Example. updating a "score" counter

- **Rule of thumb:** wherever you pass a callback function (*e.g.* mouse, keyboard, uicontrol, timer, etc.), you can specify additional arguments by passing a cell array that contains the function and additional inputs, e.g. with `{}`.

Note: this example uses `uicontrols`, which we discussed in Topic 21 / Lecture 22.

```
% Initialize the figure/gui window and center
ss = get(0,'ScreenSize');
fig = figure('Visible','on','Position',ss.*.5);
movegui(fig,'center')

% Initialize the counter value
cVal = 0;

% Add an "edit text" box to show the current value of the counter
hedit = uicontrol('Style','edit','Units','Normalized','Position',...
    [0.3 0.05 0.4 0.2],'FontSize',30,'String',cVal);

% Pushbutton to subtract 1 from the current value of the counter
uicontrol('Style','pushbutton','Units','Normalized','Position',...
    [0.2 0.4 0.2 0.2],'String','Subtract
1','Callback',{@subtractVal,cVal,hedit});

% Pushbutton to add 1 to the current value of the counter
uicontrol('Style','pushbutton','Units','Normalized','Position',...
    [0.6 0.4 0.2 0.2],'String','Add 1','Callback',{@addVal,cVal,hedit});
```



Issue: information is passed to the appropriate objects, but the counter is not actually stored anywhere. Remember, what happens in a function, stays in a function (unless otherwise appropriately passed).

Workaround: `guidata`

- Remember that guidata can be used to store a data structure associated with a parent figure (interface).
- This data structure should be initially stored in your script, then loaded and restored inside each function.
- Storing `guidata`:

```
% Initially declare structure with a field corresponding to the counter
handles.cVal = 0;
% Store guidata
% First input: figure handle
guidata(h1,handles);
```

- Loading `guidata` inside a function:

```
% Load guidata to a structure array object
handles = guidata(h1);
```

Animations

Whether you are designing a simulation, interactive visualization, or game, you may want to incorporate (and **animate**) more sophisticated objects than simply square, rectangles, or circles. To incorporate more sophisticated objects, you could:

- Draw them from square, circles, lines, etc. in MATLAB
- **Import images** into the plot space. We will focus on this here.

Revisiting image processing

Recall from before:

- **images are just "integer matrices,"** in which the each index corresponds to a unique **pixel**, and the value of the corresponding element is the **gray value** of that pixel.
- The bit depth of an image corresponds to the range of possible integer values each element can take. Color images are 3D matrices.
- An image **may be read into MATLAB** using the `imread` function.
- An image may be **displayed** using `imshow`

```
close all
fishPic = imread('fish.png');
imshow(fishPic)
```

Source: the above image is of a Porgy fish, courtesy of the Encyclopedia Britannica.

```
% A color image is a 3D matrix
size(fishPic)
```

```
ans = 1×3
         866        1544           3
```

Images as graphical handles objects

In general, all visualizations can be stored as graphics handle objects. Images are no different and may be placed in precise **positions** on a pair of axes and **stretched** by modifying the `xData` and `yData` properties.

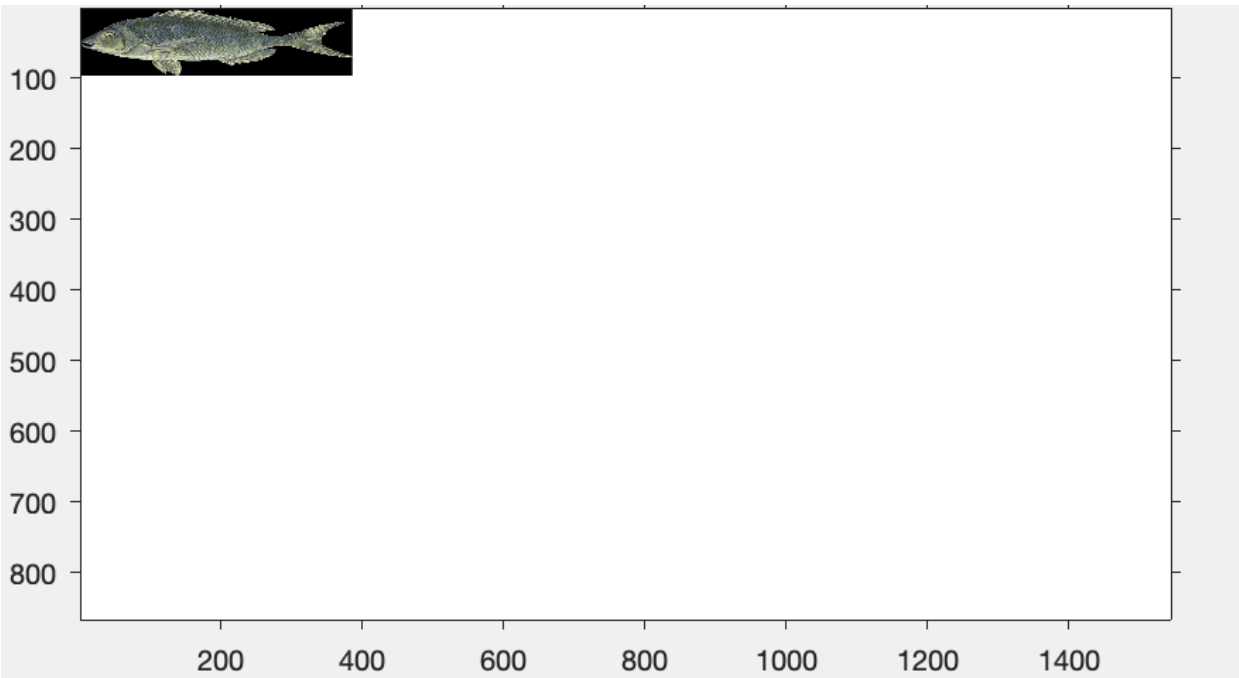
In the following example, we:

- **Create a graphics handle object** corresponding to the fish.
- **Scale the object's size** by changing the `XData` and `YData` fields.

```
close all

% Create figure and axes that are visible
ss = get(0,'ScreenSize');
h1 = figure('Visible','on','Position',ss.*.5);
g1 = axes;

% Create graphics handle object describing fish
hFish = imshow(fishPic)
```

```
hFish 
  Image with properties:

           CData: [866×1544×3 uint8]
    CDataMapping: 'direct'

  Show all properties
```

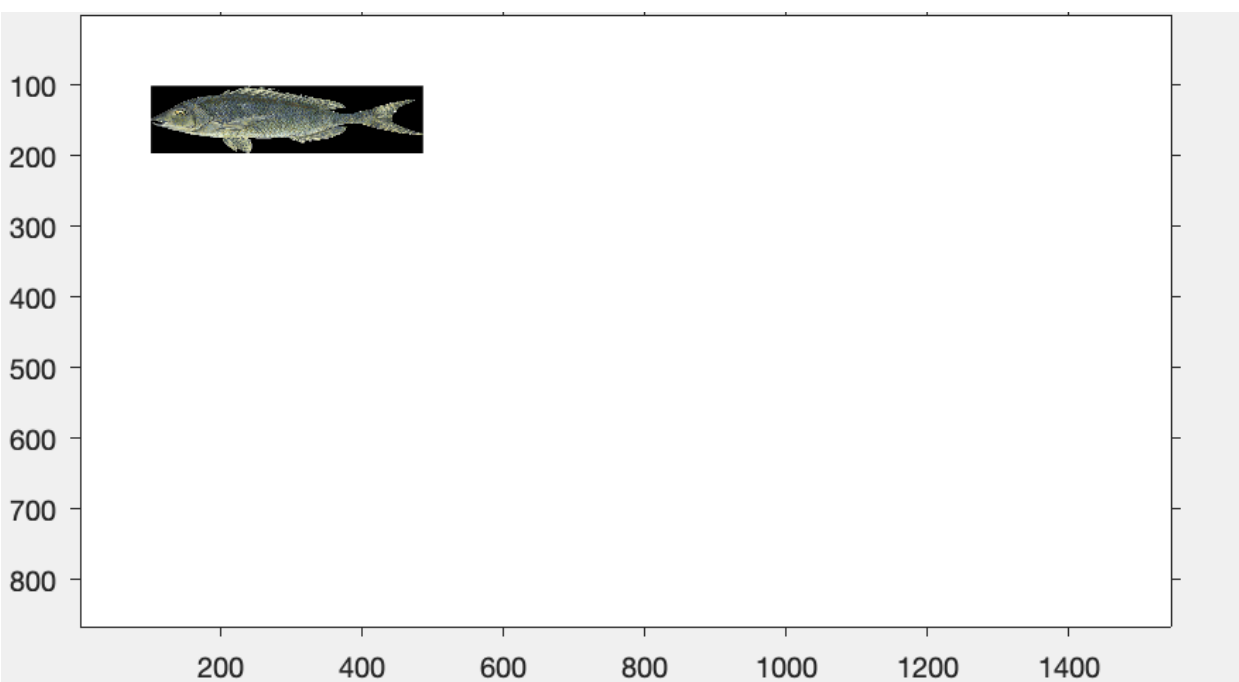
```
% Define scaling factor
sf = 0.25;

% Change scaling of image on axes
hFish.XData = [1,sf*hFish.XData(2)];
hFish.YData = [1,sf*hFish.XData(2)];

% Ensure axes are visible
g1.Visible = 'on';

% Reduce figure window size
h1.Position(3:4) = 0.5*h1.Position(3:4);
```



Note: the units of the above plot are pixels. By definition, the upper left corner is the origin at (0,0)

Furthermore, we can change the **position** of fish by translating the values in the XData and YData fields of the fish object.

```
% Move the object away from the origin
hFish.XData = hFish.XData + 100;
hFish.YData = hFish.YData + 100;
```



Note: `XData` and `YData` are two element vectors and contain the X and Y "extent" of an object. If the entries in one vector are **reversed**, the object is reversed.

- A simple way to do this is with array manipulation commands, e.g. `fliplr`

```
% Flip the orientation of the object
hFish.XData = fliplr(hFish.XData);
```

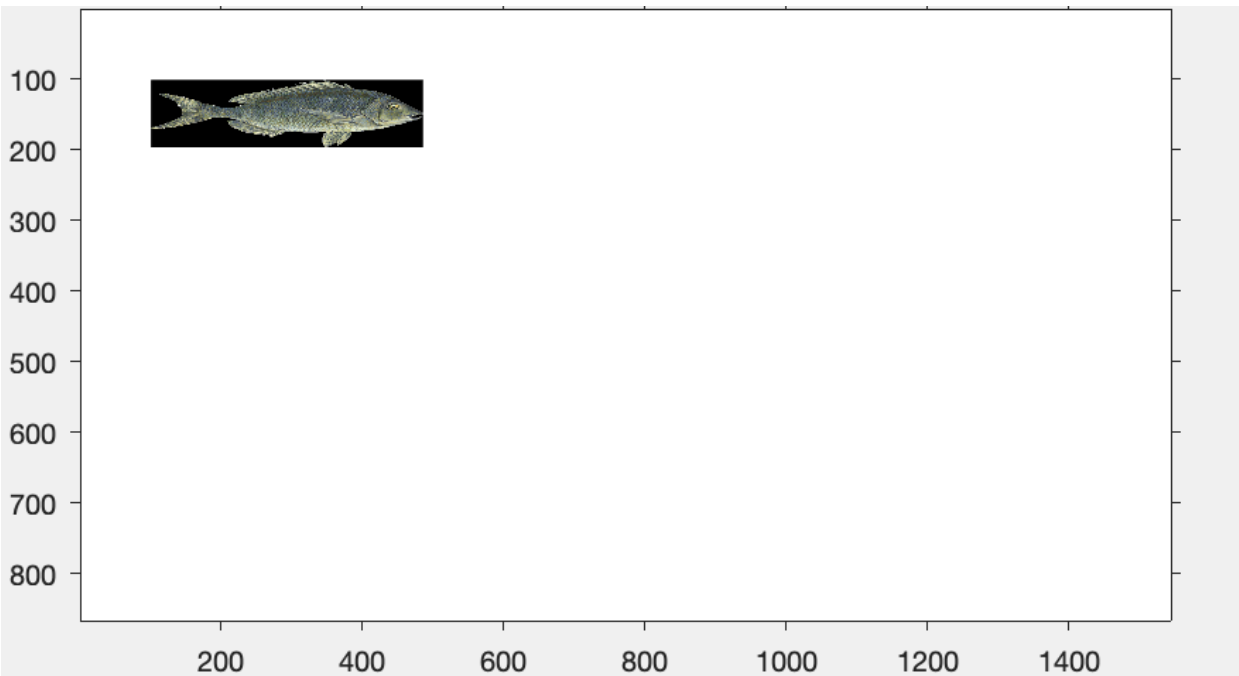
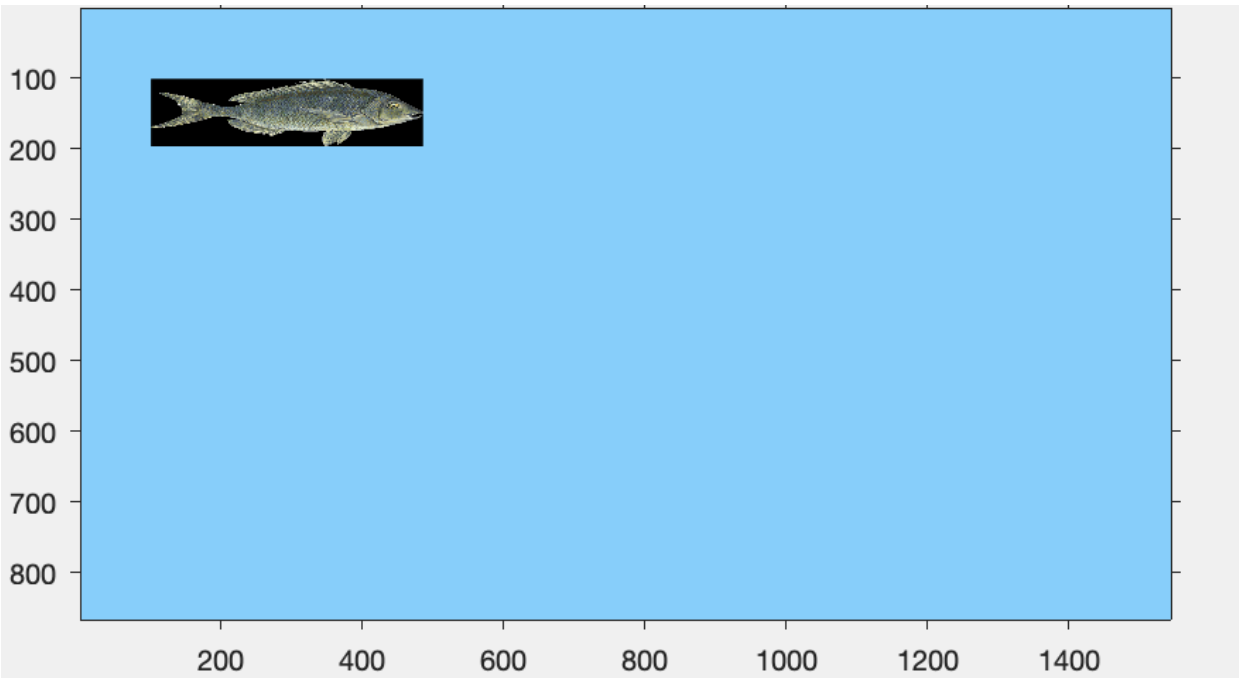


Image transparency

Consider the above fish object. Say we want to create a visualization of fish in blue water by changing the background color of the axes object.

```
% Change the axes background to blue
g1.Color = [135,206,250]/255;
```

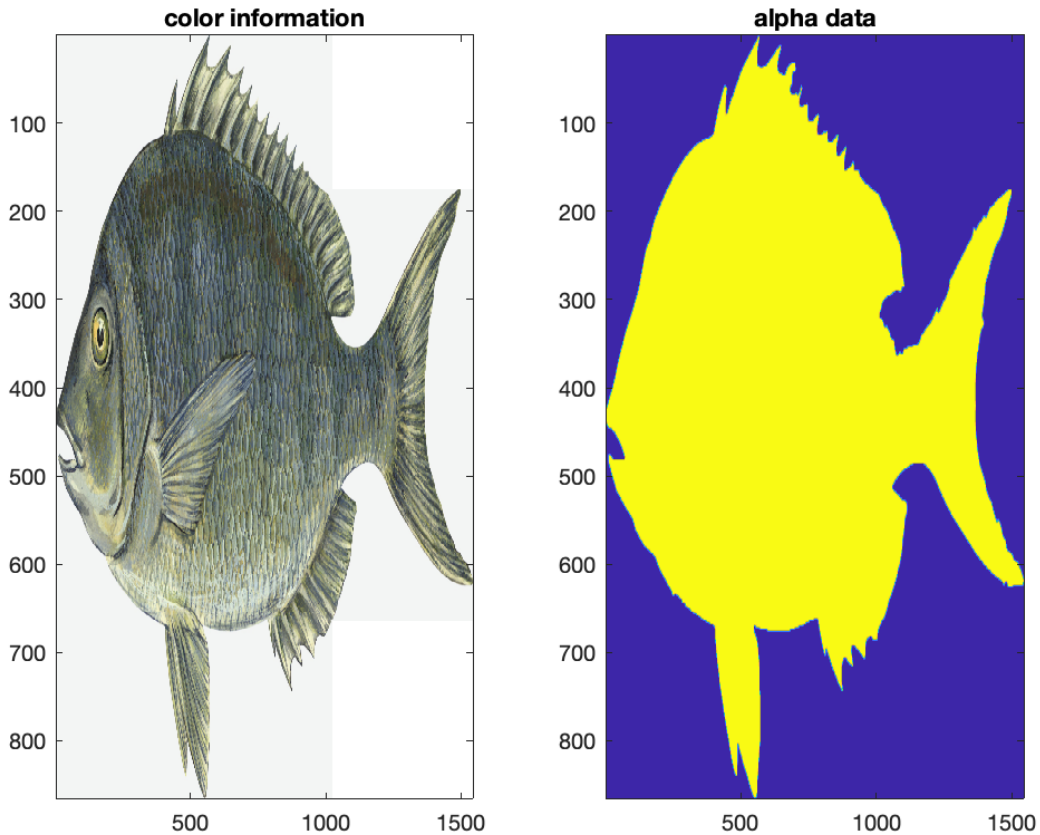


<u>Issue</u>: the black background of the fish object disrupts the overall visualization.

Fortunately, **`.png` images often contain transparency data** (`alphaData`) that can be read as the third output argument from `imread()`.

- An `AlphaData` value of 0 represents a fully transparent pixel and an `AlphaData` value of 255 indicates an opaque (fully visible) pixel – values in between indicate a semi-transparent pixel.
- Typically, jpg, jpeg, bmp, and tif image formats do not support transparency – only png image type.

```
close all

% Read the fish picture and associated alpha data
[fishPic,~,alphaFish] = imread('fish.png');

subplot(1,2,1)
image(fishPic)
title('color information')

subplot(1,2,2)
image(alphaFish)
title('alpha data')
```

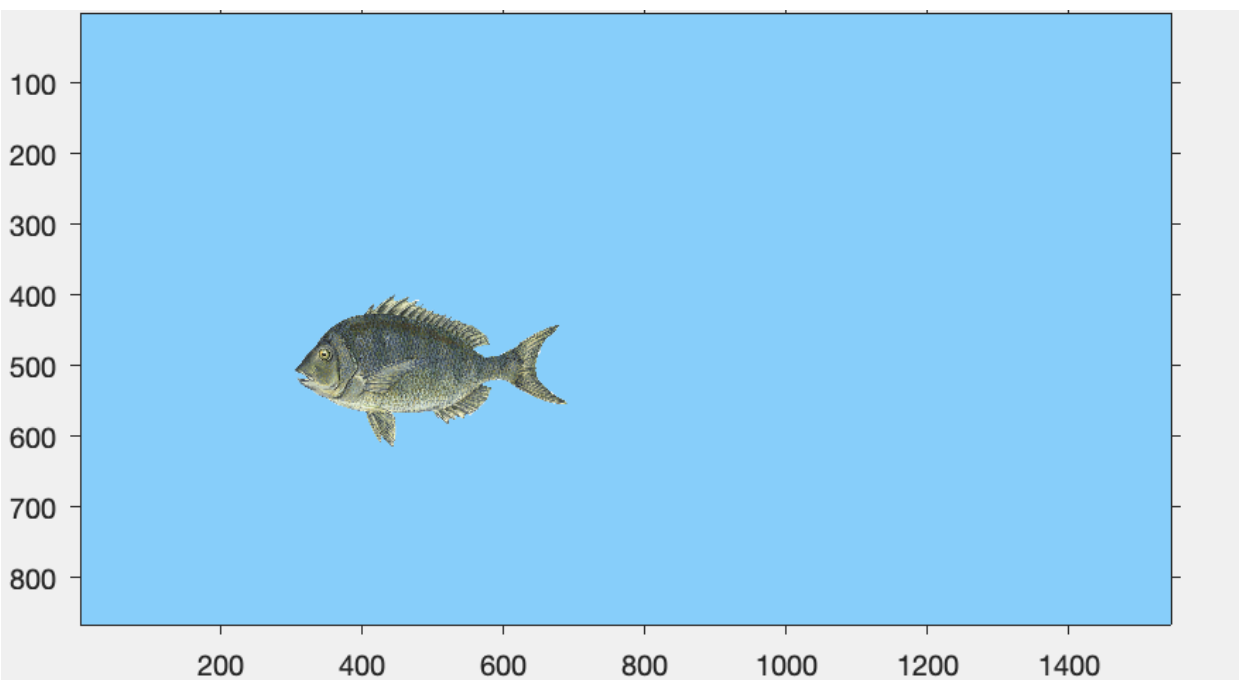


The transparency information in `.png` images may be used to remove image backgrounds and overlap images without the interference from background borders.

- The following code incorporates this tranparency information.
- The `AlphaData` field in a graphics handle object associated with an image can be used to specify the alpha data

```
close all

% Create figure and axes that are visible
h1 = figure('Visible','on');
g1 = axes;

% Create graphics handle object describing fish
hFish(1) = imshow(fishPic);

% Assign alpha data
hFish(1).AlphaData = alphaFish;

% Define scaling factor
```

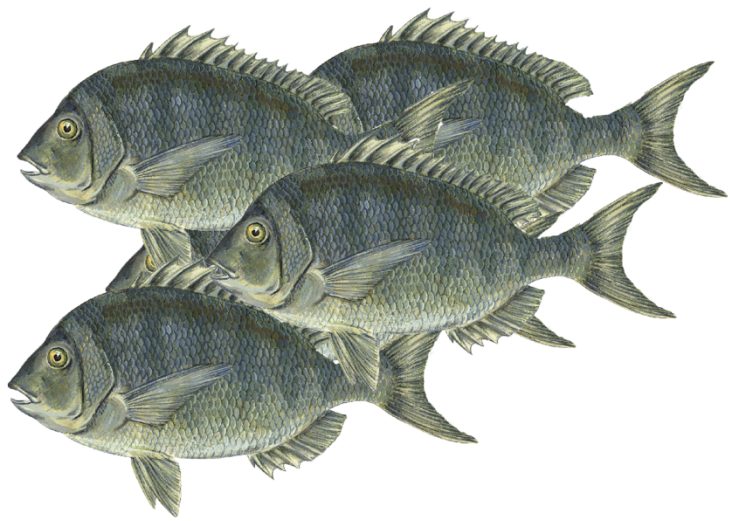
```
sf = 0.25;

% Change scaling of image on axes
hFish(1).XData = [1,sf*hFish(1).XData(2)]+randi(500);
hFish(1).YData = [1,sf*hFish(1).YData(2)]+randi(500);

% Ensure axes are visible
g1.Visible = 'on';

% Reduce figure window size
h1.Position(3:4) = 0.5*h1.Position(3:4);

% Change the axes background to blue
g1.Color = [135,206,250]/255;
```



Adding more fish to the image. Since the only the region specified by the alpha data is depicted, you can potentially overlap many fish.

```
close all
% Create five fish objects
for k = 1:5
    hold on
    % Create graphics handle object describing fish
    hFish(k) = imshow(fishPic);

    % Assign alpha data
    hFish(k).AlphaData = alphaFish;
```

```
    % Define scaling factor
    sf = 0.25;

    % Change scaling of image on axes
    hFish(k).XData = [1,sf*hFish(k).XData(2)]+randi(500);
    hFish(k).YData = [1,sf*hFish(k).YData(2)]+randi(500);

end
hold off
```



Note: in the above code, `hold on` and `hold off` are required to ensure that all fish objects are retained on the current axes.

Creating your own `.png` images with transparency data

`.png` images can be created in image editing or artistic software, such as **Adobe Photoshop** or **GIMP (*https://www.gimp.org/*)**. Penn has many resources on Photoshop, including free access on campus computers. Please see the following for more information:
https://guides.library.upenn.edu/photoshop

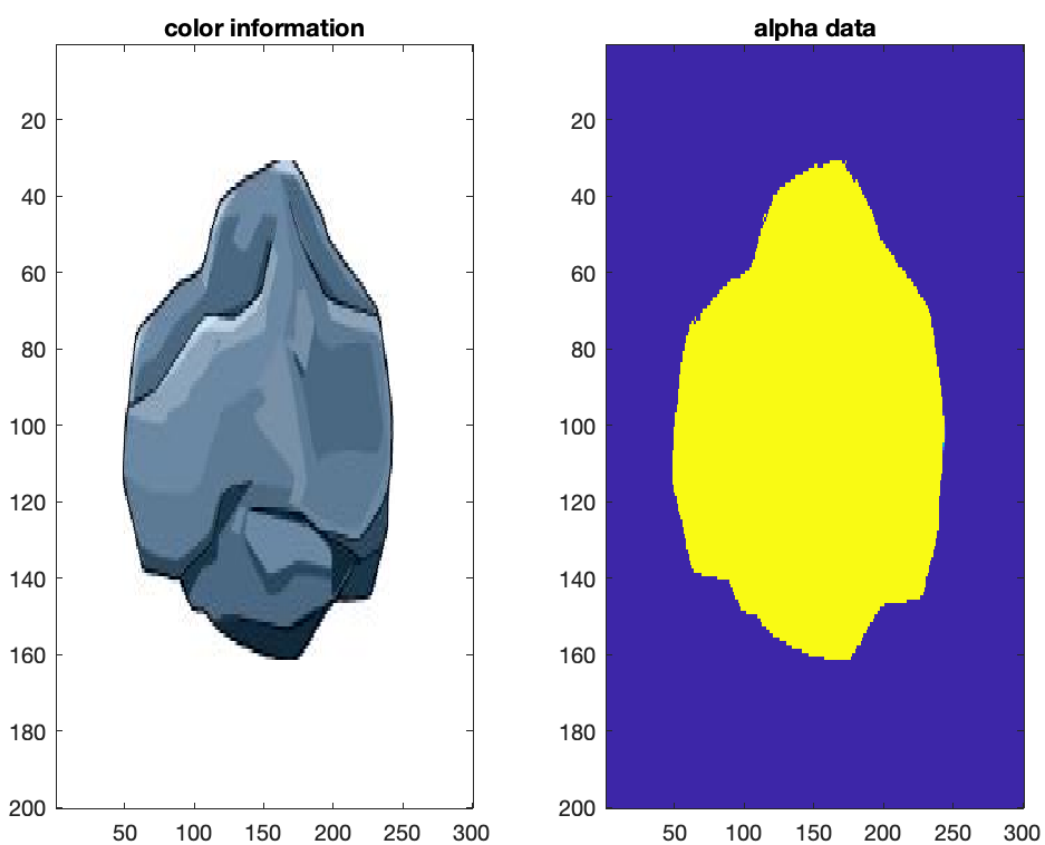
Adobe Photoshop can also be downloaded as a free trial from Adobe. To create a `.png` file in Photoshop, use the following procedure:

1. Open the image in Photoshop
2. Remove image elements using the eraser, magic lasso, box selection tool, etc
3. Disable the white background by clicking on the ***eye icon*** in the ***Layers*** panel – the background will turn into a checkerboard
4. Crop the image to remove any unnecessary pixels

5. Save the image as `.png` image.

Example. Create a .png file from a rock.

```
close all

% Read the fish picture and associated alpha data
[rockPic,~,alphaRock] = imread('Rock.png');

subplot(1,2,1)
image(rockPic)
title('color information')

subplot(1,2,2)
image(alphaRock)
title('alpha data')
```



Object animation

Animating fish motion by programming the movement patterns. Using the tools that we have used so far, we can program in fish motion.

In this example, we:

- Create three fish objects
- Enable each to move randomly through 200 ms time steps
- Change the background of the axes to be blue.

```
close all

% Step 1: set up figure

% Create figure and axes that are visible
h1 = figure('Visible','on');
g1 = axes;

% Create three fish objects
for k = 1:3
    hold on
    % Create graphics handle object describing fish
    hFish(k) = imshow(fishPic);

    % Assign alpha data
    hFish(k).AlphaData = alphaFish;

    % Define scaling factor
    sf = 0.25;

    % Change scaling of image on axes
    hFish(k).XData = [1,sf*hFish(k).XData(2)]+randi(500);
    hFish(k).YData = [1,sf*hFish(k).YData(2)]+randi(500);

end
hold off

% Ensure axes are visible
g1.Visible = 'on';

% Reduce figure window size
h1.Position(3:4) = 0.5*h1.Position(3:4);

% Change the axes background to blue
g1.Color = [135,206,250]/255;
xlim([-200,1500])
ylim([-200,1000])

% Step 2: loop through 50 simulation time steps
for m = 1:25
```

```
    % For each fish, update position
    for k = 1:3
        % Change position of image on axes
        hFish(k).XData = hFish(k).XData + randi(100) - 50;
        hFish(k).YData = hFish(k).YData + randi(100) - 50;
    end
    pause(0.1)
end
```

Note. The fish move, but it looks a little silly how they "back up" when moving to the right. One option to improve this would be to flip the image when it moves to the right.

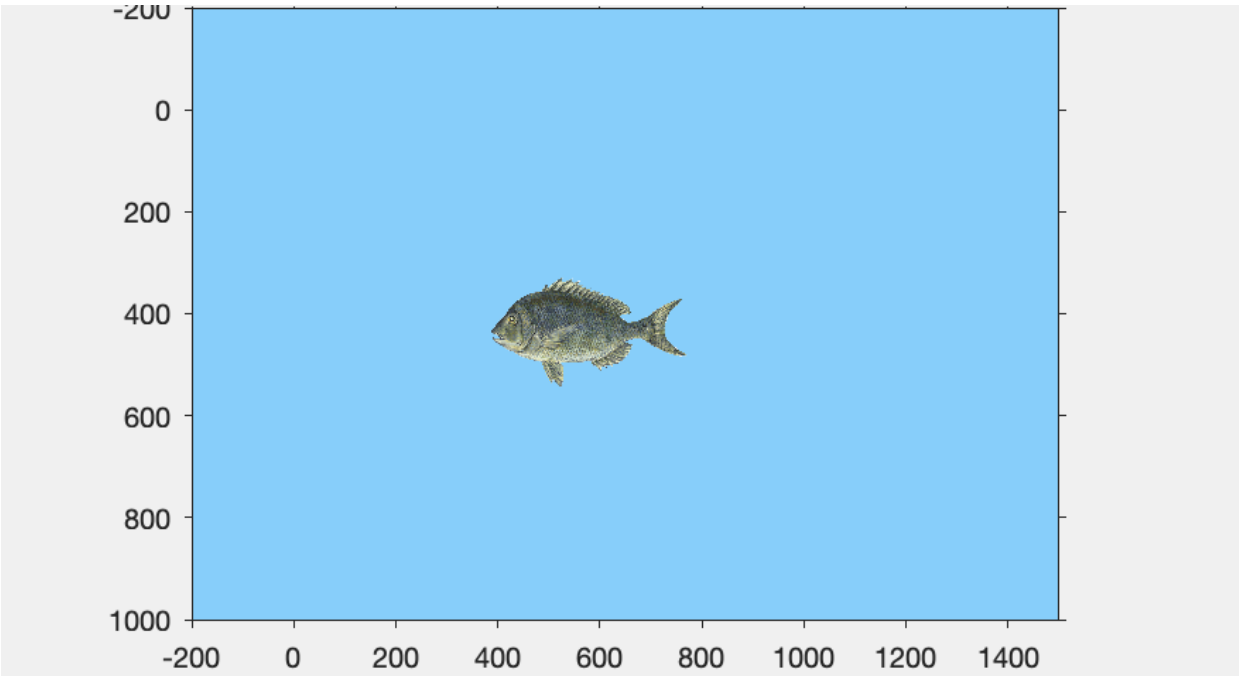
Animating fish motion by keyboard callback. Using the techniques described in Topic 20 and 21, we can create a keyboard callback object to move the fish in response to arrow key keystrokes.

```
function DoWhenKeyIsPressed(h1,eventdata,hFish)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.

    % Load GUI data
    % Create field to store the current xy direction of fish
    handles = guidata(h1);

    % Movement increment
    inc = 10;

    % Move up
    if strcmp(eventdata.Key,'uparrow')
        hFish.YData = hFish.YData - inc;

    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hFish.YData = hFish.YData + inc;

    % Move left
    elseif strcmp(eventdata.Key,'leftarrow')
        % If the current direction is set to move left
        if handles.moveDir == "left"
            hFish.XData = hFish.XData - inc;
        % Otherwise, flip the orientation of the fish
        else
            hFish.XData = fliplr(hFish.XData) - inc;
            handles.moveDir = "left";
        end

    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        % If the current direction is set to move left
        if handles.moveDir == "right"
            hFish.XData = hFish.XData + inc;
        % Otherwise, flip the orientation of the fish
        else
```

```
            hFish.XData = fliplr(hFish.XData) + inc;
            handles.moveDir = "right";
        end

    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end

    % Store the data structure as GUI data
    guidata(h1,handles);
end
```

```
close all

% Create figure
% Create figure and axes that are visible
h1 = figure('Visible','on');
g1 = axes;

% Create data structure to store info about interface
handles = struct;
% Create field to store the current xy direction of fish
handles.moveDir = "left";
% Store the data structure as GUI data
guidata(h1,handles);

% Create fish object
% Create graphics handle object describing fish
hFish = imshow(fishPic);
% Assign alpha data
hFish.AlphaData = alphaFish;
% Define scaling factor
sf = 0.25;
% Change scaling of image on axes
hFish.XData = [1,sf*hFish.XData(2)]+randi(500);
hFish.YData = [1,sf*hFish.YData(2)]+randi(500);

% Specify additional graphical parameters
% Ensure axes are visible
g1.Visible = 'on';

% Reduce figure window size
h1.Position(3:4) = 0.5*h1.Position(3:4);
```

```
% Change the axes background to blue
g1.Color = [135,206,250]/255;
xlim([-200,1500])
ylim([-200,1000])
```




```
% Assign key press function
h1.KeyPressFcn = {@Topic25DoWhenKeyIsPressed,hFish};
```

Animation with timer objects

Events can be timed/initiated at specific intervals using `tic/toc`, `clock`, or **timer objects**.

- `tic/toc` and `clock` are effective when code naturally involves loops, but timer objects can initiate events even without a loop
- Timer objects are not loops, run independently "in the background", and allow for more precise timing/execution of events.
- Just like when creating mouse and keyboard callbacks, timers utilize callbacks to perform actions

Example. Print `Hello person!` to the command line 5 times after a 2 second delay and in 3 second intervals using a timer object.

```
% Set up the timer object t.
t = timer('TimerFcn', @sayIt, 'StartFcn', @startItUp, 'StopFcn', @stopIt,
'ExecutionMode', 'fixedRate', 'StartDelay', 2, 'Period', 3, 'TasksToExecute',
5);
```

```
% start timer
start(t)
```

The timer associates local functions (subfunctions) with the start (`'StartFcn'`), firing (`'TimerFcn'`), and stop (`'StopFcn'`) of the timer

- In the above, the following function is executed **at timer start**:

```
function startItUp(~,~)
    % Timer StartFcn callback.
    disp('A timer has been started.')
    disp('It will wait 2 seconds to fire the first time...')
    disp('and then it will fire every 3 seconds for 5 rounds!')
end
```

- In the above, the following function is the "TimerFcn" who is called based on the commands in the timer

```
function sayIt(~,~)
    % Timer function callback.
    disp('Hello person!')
end
```

- In the above, the following function is executed **at timer end**. <u>Note: the timer is deleted at the end.</u>

```
function stopIt(t,~)
    % Timer StartFcn callback.
    disp('The timer ''StopFcn'' just executed.')
    disp(['It utilizes the first input of the callback, '...
        'the handle of the timer, to stop the timer.'])
    disp('Always delete timers after you are done using them.')

    delete(t)
end
```

<u>Note</u>: as with all objects, the fields can be modified using dot notation.

IMPORTANT: ALWAYS STOP AND DELETE TIMERS WHEN DONE WITH THEM - NOT DOING SO CAN LEAD TO A LOT OF TIMERS LEFT RUNNING IN THE BACKGROUND AND SUCKING COMPUTER RESOURCES

Specific timers can be stopped by reference to their handles

```
stop(<timer_handle>)
delete(<timer_handle>)
```

All timers can be stopped and deleted by specifying

```
stop(timerfindall)
```

```
delete(timerfindall)
```

Example problem: rock simulator

Animate a sequence of falling rocks in water based on a **timer object**.

This code should:

- Create a timer that makes a rock fall three times, with one rock falling every 4 seconds
- Create one rock object that falls multiple times
- Use material properties where possible to ensure that the simulation is physically reasonable

Step 1: Plan, pseudocode, and prepare. Write down what we know. Assume the stone is **granite** and treat it as a sphere of radius **150 m**.

Values that we know:

- Density of granite: $\rho = 2,670 \ \frac{kg}{m^3}$
- Density of water: $\rho_w = 1,000 \ \frac{kg}{m^3}$
- Viscosity of water: $\eta = 10^{-3} \ Pa \cdot s$
- Radius of rock: $a = 150 \ m$

Creating the differential equation:

From our knowledge of **physics**, we know that the force exerted on an object by gravity (and including buoyancy) is given by the following equation, where g is the acceleration due to gravity and V is the volume of the rock (assumed to be a sphere).

$$F_g(t) = -(\rho - \rho_w)gV = -\frac{4\pi}{3}a^3(\rho - \rho_w)g$$

Last Tuesday, we used **Stokes's law** to describe the drag force induced by a liquid on a sphere. We will use this relationship again here:

$$F_{\text{drag}}(t) = -6\pi a\eta v(t) = -6\pi a\eta \frac{dx}{dt}$$

From **Newton's second law of motion**, we can thus relate all forces acting on the object:

$$ma = m\frac{d^2x}{dt^2} = \rho\frac{4}{3}\pi a^3\frac{d^2x}{dt^2} = \sum_i F_i(t)$$

Simplifying this differential equation:

$$\rho\frac{4}{3}\pi a^3\frac{d^2x}{dt^2} + 6\pi a\eta\frac{dx}{dt} + \frac{4\pi}{3}a^3(\rho - \rho_w)g = 0$$

$$\frac{d^2x}{dt^2} + \frac{6\eta}{\rho\frac{4}{3}a^2}\frac{dx}{dt} + \frac{(\rho - \rho_w)g}{\rho} = 0$$

Or, by definining and substituting variables b and c:

$$\frac{d^2x}{dt^2} + b\frac{dx}{dt} + c = 0$$

$$b = \frac{6\eta}{\rho\frac{4}{3}a^2}$$

$$c = \frac{(\rho - \rho_w)g}{\rho}$$

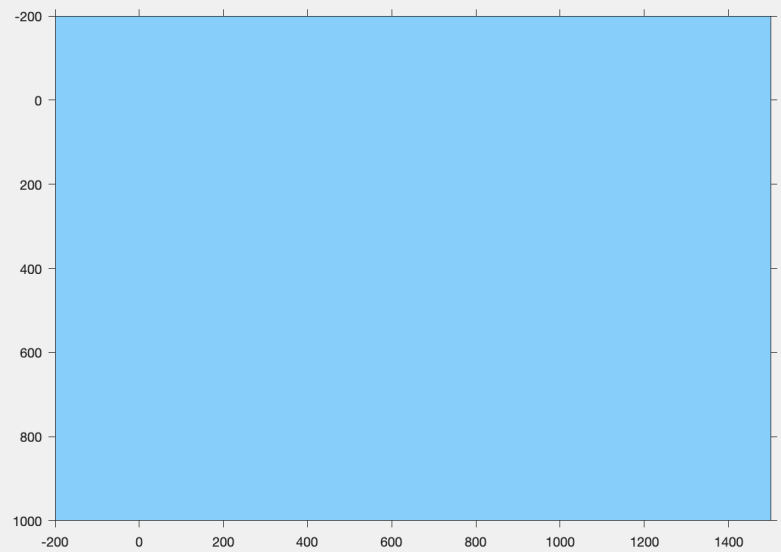
Step 2: Create a "base" visualization (that does not include the physics)

This code should incorporate a timer and timer function that drops a rock every four seconds. The timer callback function is given here:

```
function dropRock(~,~,hRock,hRockSize)
    % Timer function that is called everytime the rock should be dropped

    % Reset the rock's initial position to be above the water line
    hRock.XData = [1,hRockSize(1)] + randi(1000);
    hRock.YData = [-hRockSize(2),-1];

    % Ensure rock is visible
    hRock.Visible = 'on';

    % Loop through time steps and move the rock down by some known
amount
    % of time
    for k = 1:50
        hRock.YData = hRock.YData + 100;
        pause(0.1)
    end
end
```

```
close all

% Create figure
% Create figure and axes that are visible
h2 = figure('Visible','on');
g2 = axes;

% Create rock object using the previously loaded rock
% Create graphics handle object describing rock
hRock = imshow(rockPic);
hRock.Visible = 'off';
% Assign alpha daya
```

```
hRock.AlphaData = alphaRock;
% Store XY dimmensions of rock
hRockSize = [hRock.XData(2), hRock.YData(2)];

% Ensure axes are visible
g2.Visible = 'on';

% Change the axes background to blue
g2.Color = [135,206,250]/255;
xlim([-200,1500])
ylim([-200,1000])
h2.Position = [672,238,941,698];
```



Note: the above code uses the the same `stopIt` function as above to ensure that the timer is deleted once it runs to completion

Step 3: Incorporate the physics

To solve using `ode45`, a higher order differential equation must be rewritten as a set of differential equations. Here, the differential equation is:

$$\frac{d^2x}{dt^2} + b\frac{dx}{dt} + c = 0$$

Defining new variables:

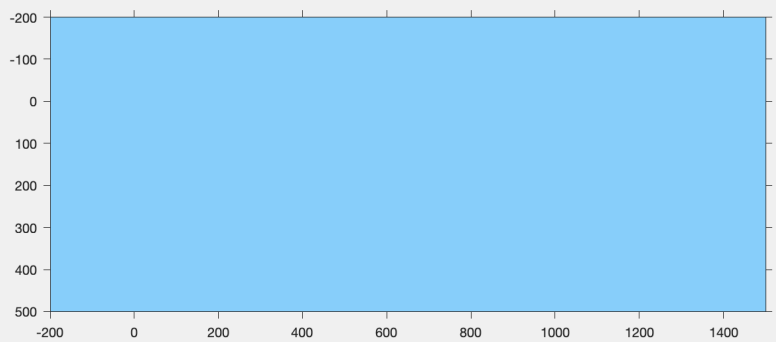
$$x = y_1$$
$$\frac{dx}{dt} = \frac{dy_1}{\mathrm{dt}} = y_2$$
$$\frac{dy_2}{\mathrm{dt}} = -by_2 - c$$

Thus, we can use the following function to represent the system of ODEs:

```
function dYdt = rockODE(t,y,b,c)

    dYdt = zeros(2,1);
    dYdt(1) = y(2); % y(1) = x
    dYdt(2) = -b*y(2) - c; % z(2) = dy/dt

end
```

```
close all

% Create figure
% Create figure and axes that are visible
h2 = figure('Visible','on');
g2 = axes;

% Physical parameters
rho_g = 2670;
rho_w = 1000;
eta = 10^-3;
a = 150;
g = 9.81;

% Constants
b = 6*eta/(rho_g*4/3*a^2);
c = (rho_g - rho_w)*g/rho_g;
% Estimate time to stop ODE calculation
t_end = 5*sqrt(1000/c);

%Solve ODE
[t,out] = ode45(@rockODE, 0:0.05:t_end, [0,0], [],b,c);
```

```
out = -out;
y = out(out(:,1)<600);
t = t(out(:,1)<600);

% Create rock object using the previously loaded rock
% Create graphics handle object describing rock
hRock = imshow(rockPic);
hRock.Visible = 'off';

% Assign alpha data
hRock.AlphaData = alphaRock;
% Store XY dimmensions of rock
hRockSize = [hRock.XData(2),hRock.YData(2)];

% Ensure axes are visible
g2.Visible = 'on';

% Change the axes background to blue
g2.Color = [135, 206, 250]/255;
xlim([-200,1500])
ylim([-200,500])
h2.Position = [672,238,941,698];
```




```
function dropRock_2(~,~,hRock,hRockSize,y_data)
    % Timer function that is called everytime the rock should be dropped

    % Reset the rock's initial position to be above the water line
    hRock.XData = [1,hRockSize(1)] + randi(1000);
    hRock.YData = [-hRockSize(2),-1];

    initialPos = hRock.YData;

    % Ensure rock is visible
    hRock.Visible = 'on';

    % Loop through time steps and move the rock down by some known
amount
    % of time
    for k = 1:length(y_data)
        hRock.YData = initialPos + y_data(k);
        pause(0.05)
    end
end
```

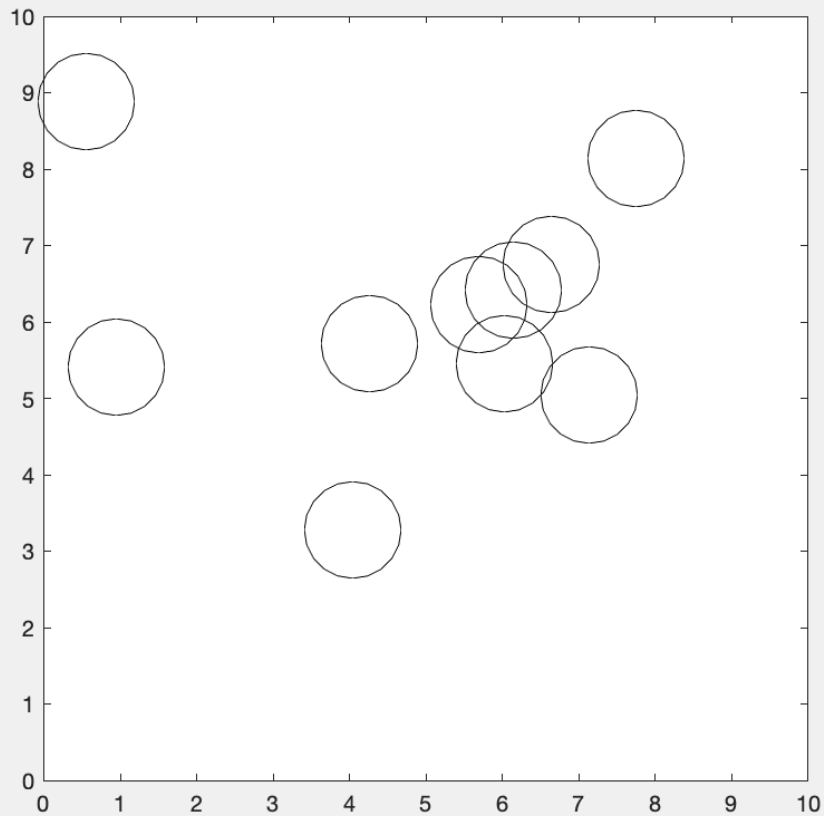
Example problem: drag and drop

A useful feature of interfaces in the capability for "drag-and-drop" user interactions. Produce a script that enables the user to "drag-and-drop" circles on an interface.

This script requires mouse callbacks to perform the following operations:

- Left click within the boundary of the nearast marker to grab it
- Use the `'WindowButtonMotionFcn'` call back to move it
- Release the mouse button releases control of the marker

```
close all

% Establish the figure and the button down/up and motion
% callbacks.
h1 = figure('Visible','on');

% The diameter of the markers.
D = 1;

% 'ind' will be used to track the index of the marker that is to
% be moved when a mouse movement is detected. An empty set
% indicates that no index has been selected.
handles.ind = [];

% Store guidata
guidata(h1,handles);

% Initialize the center locations of 10 markers.
x = (10-D)*rand(1,10)+D/2;
y = (10-D)*rand(1,10)+D/2;

% Plot the markers.
p = plot(x,y,'ko');

% Adjust the axis scaling so that the radius of the markers is
% scaled to the size of the axes.
axis equal
axis([0 10 0 10])
ax = gca;
ax.Units = 'points';
p.MarkerSize = D/diff(xlim)*ax.Position(3);
```




```
% Declare callback functions
h1.WindowButtonDownFcn = {@pick,D,p};
h1.WindowButtonMotionFcn = {@move,p};
h1.WindowButtonUpFcn = @place;
```

Functions

Topic 1 Functions:

```
function c = Pythagorean(a,b)
    % Compute c for the relation
    % a^2+b^2=c^2.
    % a, b, and c are scalars.

    % Compute c.
    c = sqrt(a^2+b^2);
```

```
end
```

Topic 3 Functions:

```
function [p,s] = myFunction(a,b,c)
    % p and s are computed within the function
    p = a+b;
    s = b*c;
end
```

- Unused, but contained here

```
function out = mySquareRoot(a)
    % Returns the square root of a scalar,
    % vector, or multidimensional array
    % input, a, to out
    % M. Siedlik
    % 9/6/2020

    % Calculates the square root
    % of the inputout = sqrt(a);
    out = sqrt(a);
end
```

- To be used with line 63

```
function out = returnOddElements(v)
    % Select the odd indices of the input vector
    out = v(1:2:end);
end
```

- To be used with line 65

```
function F = eqnSet(x)
    % System of two non linear equations
    % F(1) and F(2): "equation" 1 and 2, respectively
    % x(1) and x(2): variable 1 and 2, respectively
    F(1) = x(2)-2*x(1)^2+3;
    F(2) = x(2)^2-x(1);
end
```

Topic 7 Functions

```
function [n, diceVal] = diceRollSimulator(max_n, n_sides)
%H1 help line, running a simulation until any number comes up max_n times
%in a row

%Simpler: roll dice until 3 1's land in a row

%roll one dice
n = 0;
numInARow = 0;
%previous value not 1-6
preDiceVal  = 0;

%evaluate dice face value
% If dice face = 1 it is good unless restart
%continue to iterate until true
    while numInARow < max_n
        %have a variable to hold the pervious dice value

        diceVal = randi(6);
        if diceVal == preDiceVal
            numInARow = numInARow + 1;
        else
            numInARow = 1;
            preDiceVal = diceVal;
        end
        %counter
        n = n+1;
    end
end
```

Topic 8 Functions

```
function out = checkEndGame(gameboard,player)
    %check to see if any player has one
    out = 1; %check if game is still going
    %checks the first column
    if prod(gameboard(1:3) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
```

```
    elseif prod(gameboard(4:6) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
    elseif prod(gameboard(7:9) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
    elseif prod(gameboard([1,4,7]) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
    elseif prod(gameboard([2,5,8]) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
    elseif prod(gameboard([3,6,9]) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
    elseif prod(gameboard([1,5,9]) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
    elseif prod(gameboard([3,5,7]) == player)
        disp(strcat(num2str(player), "wins"))
        out = 0;
    end

end
```

Topic 11 Functions

```
function res1 = square_add1(a,b)
    % Example function. In this case there are two inputs and one
    % output.
    res1 = a^2 + b^2 + 1;
end
```

```
function res2 = operateOnKnownValues(fxn)
    % Example function: input 1 is another function
    res2 = fxn(1,2);

end
```

```
function res3 = exampleA(fxn,A)
    % Example function: input 1 is a function, input 2 is an array
    res3 = fxn(A) + 1;

end
```

```
function res4 = exampleB(fxn,A)
    % Example function: input 1 is a function, input 2 is an array
    res4 = fxn(A) - 1;

end
```

Functions for Spring example

```
function force = springForce1(delta)
    % Calculate the force required to cause a deformation of delta
    % Spring constant = 0.5
    force = 0.5*delta;
end
```

```
function force = springForce2(delta)
    % Calculate the force required to cause a deformation of delta
    % in a nonlinear spring
    force = .2*delta^2 + .5*delta;
end
```

```
function force = parallelSprings(spring1,spring2,delta)
    % Force for two springs loaded in parallel config.
    % spring1 = handle to force/deflection eqn for spring 1
    % spring2 = handle to force/deflection eqn for spring 2
    % delta = displacement of springs

    force = spring1(delta) + spring2(delta);
end
```

Topic 12 Functions

Functions stored at the end of the document.

```
function fxn1(f,x)
    % Example function
    output = f(x);
    disp(strcat('The output of this function is... ',num2str(output)));
end
```

Functions for practice problem: Newton-Raphson example

```
% % Script for using the Newton-Raphson method to solve for f(x)
% function xn = newtonSolve(F, xn, tol)
% % Step 1: specify initial guess
%     xn = 5; % initial guess
%     xprev = inf; % Variable to store previous xn value -> ensure it is
%                 % not xn to enter while loop
%
%     % Step 2: iterate until xn -> xn+1
%     while abs(xn - xprev) > 10^-5
%
%         % Step 3: evaluate function at xn
%         f = F(xn);
%         % Step 3a: evaluate derivative at xn
%         f_prime_n = 4*xn^3 + 4*(xn-2);
%
%         % Instead of calculating the "next" x value, store the previous xn
%         % value
%         xprev = xn;
%
%         % Step 3b: update xn to next value for the iteration
%         xn = xn - f(1)/f(2);
%     end
%     disp(xn)
%
% end
```

```
function F = originalEqn(x)
    %function
    F(1) = x^4 + 2*(x-2)^2 - 20;
    %derivative of function
```

```
    F(2) = 4*x^3 + 4*(x-2);
end
```

```
function G = newEqn(x)
    % Function of interest
    G(1) = x + exp(-x) - 4;
    % Derivative of function of interest
    G(2) = 1 - exp(-x);
end
```

Function examples for recursion

```
function out = recursiveAdd(in)
    % Adds a number to the input up to
    % a value of 5

    if in<5
        in = in + 1;
        out = recursiveAdd(in);
    else
        out = in;
    end
end
```

```
function x = recursiveFib(x,maxVal)
    % Function using recursion to create a Fibonacci sequence vector

    % If the last element is less that maxVal, add the next element of the
    % Fibonacci sequence to the vector
    if  x(end-1) + x(end) < maxVal
        % Calculate next entry
        x(end+1) = x(end)+x(end-1);
        % Use recursion to repeat the process
        x = recursiveFib(x,maxVal);
    end
end
```

```
function y = fact(n)
    % Recursive function for determining the factorial

    % Specify the number we are interested in solving the factorial for
    y = n;

    % Specify the "lower limit" in the factorial calculation
    if n == 0
        y = 1;
    else
        % We multiply by all the integers before the input,
        % one at a time...
        y = y*fact(n-1);
    end
end
```

Topic 14 (Legrange Function)

```
function pn = legendreP(x,n)
    %calculate the n-th vector legendre polynomial
    %vector 1: vector of positions
    %vector 2: order of legendre polynomial

    %n = 0 -> pn should be n for all x
    if n == 0
        %make a vector full of ones based on the size of x
        pn = ones(size(x));

    % if n = 1 -> pn = x
    elseif n == 1
        pn = x;

    %pn depends on previous values
    else
        pn = 1/n*((2*n-1)*x.*legendreP(x,n-1) - (n-1)*legendreP(x,n-2));
    end

end
```

Topic 15 Functions

Practice problem: Reactor cooling

```
function xn = newtonSolve(F,xn,tol)
    % The purpose of this function is to apply the Newton-Raphson method to
    % solve any arbitrary function, specified in an input F.
    % Inputs:
    % F: function and derivative
    % x: initial guess
    % tol: specified tolerance

    % Step 1: specify initial guess
    % Input xn -> initial guess
    xprev = inf; % Variable to store previous xn value -> ensure it is
                % not xn to enter while loop

    % Step 2: iterate until xn -> xn+1
    while abs(xn - xprev) > tol
        % Apply input function to algorithm. NOTE: first output is function
        % evaluated at xn, the second output is the derivative of the
        % function evaluated at xn
        f = F(xn);

        % Instead of calculating the "next" x value, store the previous xn
        % value
        xprev = xn;

        % Step 3b: update xn to next value for the iteration
        xn = xn - f(1)/f(2);
    end
end
```

```
function xn = newtonSolveReturnXvalues(F,xn,tol)
    % The purpose of this function is to apply the Newton-Raphson method to
    % solve any arbitrary function, specified in an input F.
    % Inputs:
    % F: function and derivative
    % x: initial guess
    % tol: specified tolerance

    % Step 1: specify initial guess
    % Input xn -> initial guess
    xprev = inf; % Variable to store previous xn value -> ensure it is
                % not xn to enter while loop
```

```
    % Step 2: iterate until xn -> xn+1
    while abs(xn(end) - xprev) > tol
        % Apply input function to algorithm. NOTE: first output is function
        % evaluated at xn, the second output is the derivative of the
        % function evaluated at xn
        f = F(xn(end));

        % Instead of calculating the "next" x value, store the previous xn
        % value
        xprev = xn(end);

        % Step 3b: update xn to next value for the iteration
        xn(end+1) = xn(end) - f(1)/f(2);
    end

end
```

```
function G = quiz3(t,h)
    % These two equations represent a function and its derivative that can
    % be solved by
    % G(1): function
    % G(2): derivative of function

    G(1) = 50*(1./(t+1) + 5*(t+1).^(-4*h)) + 300 - 400;
    G(2) = 50*(-1*(t+1).^-2 + 5*(-4*h)*(t+1).^(-4*h-1));
end
```

Topic 19 Functions

```
function FctnToCall(~,eventdata)
    if strcmp(eventdata.Key,'w')
        disp('Pressed a w!')
    end
end
```

- Moving ball example

```
function DoWhenKeyIsPressed(~,eventdata,hBall)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.
```

```
    % Movement increment
    inc = 0.3;

    % Move up
    if strcmp(eventdata.Key,'uparrow')
        hBall.Position(2) = hBall.Position(2)+inc;
    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hBall.Position(2) = hBall.Position(2)-inc;
    % Move left
    elseif strcmp(eventdata.Key,'leftarrow')
        hBall.Position(1) = hBall.Position(1)-inc;
    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        hBall.Position(1) = hBall.Position(1)+inc;
    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end
end
```

Topic 20 Functions

```
function numSteps = simulateDisease(moveModifier)
    % Inputs:
    % [0,1]: particle 1 is in quarantine
    % [1,1]: particles can move freely
    % [1,0]: particle 2 is in quarantine

    % 1) vector representing posititions of two particles
    pos = [1,4];

    % 2) Define particle identities
    % Particle 1 will be infected
    % Particle 2 will initially be healthy

    % Define counter for steps
    numSteps = 0;

    % Step through the simulation. in each step update the particles
```

```
    % positions
    while abs(pos(1) - pos(2)) >=1

        % A) Create vector determining particle step
        d_k = (rand(1,2)-0.5).*moveModifier;

        % B) Update particle position
        pos = pos + d_k;

        % C) Ensure particle doesn't move out of bounds
        pos = max(min(pos,6),0);

        % D) Update counter
        numSteps = numSteps + 1;
    end
end
```

Topic 21 Functions

Functions for this Lecture

- **Example 1**. Moving ball example (from before)

```
function Topic21DoWhenKeyIsPressed(~,eventdata,hBall)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.

    % Movement increment
    inc = 0.3;

    % Move up
    if strcmp(eventdata.Key,'uparrow')
        hBall.Position(2) = min(hBall.Position(2)+inc,9);

    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hBall.Position(2) = max(hBall.Position(2)-inc,-10);
    % Move left
    elseif strcmp(eventdata.Key,'leftarrow')
        hBall.Position(1) = max(hBall.Position(1)-inc,-10);
    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        hBall.Position(1) = min(hBall.Position(1)+inc,9);
```

```
    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end
end
```

- **Example 2**. Key press function

```
function ex3_keyPress(~,eventdata,hBall)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.

    % Movement increment
    inc = 0.3;

    % Reset FaceColor to be black
    hBall.FaceColor = 'k';

    % Move up
    if strcmp(eventdata.Key,'uparrow')
        hBall.Position(2) = min(hBall.Position(2)+inc,9);

    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hBall.Position(2) = max(hBall.Position(2)-inc,-10);
    % Move left
    elseif strcmp(eventdata.Key,'leftarrow')
        hBall.Position(1) = max(hBall.Position(1)-inc,-10);
    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        hBall.Position(1) = min(hBall.Position(1)+inc,9);
    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end
end
```

- **Example 2.** key release function

```
function ex3_keyRelease(~,~,hBall)
    % Function to be executed when key is released. Takes a graphics object
    % corresponding to the ball as an input

    % Increase the whiteness of the ball until it becomes fully white
    while hBall.FaceColor(1) < 0.9

        % Make FaceColor whiter
        hBall.FaceColor = min(hBall.FaceColor + 0.1,1);

        % Pause for 100 ms
        pause(0.1)
    end

    % If ~1 second passes without moving, game over
    t1 = text(-10,0,'GAME OVER');
    t1.FontSize = 50;
    t1.Color = 'r';

end
```

- **Example 3**. Mouse callback - Depict position of mouse on screen

```
function moveText(~,~,g)
    % The mouse button motion callback.

    % Get the current mouse location.
    pos = get(gca, 'CurrentPoint');
    x = pos(1,1); y = pos(1,2);

    % Update the text object readout to the current location
    % and place its location at the cursor tip.
    g.String = ['x = ',num2str(x),', y = ',num2str(y)];
    g.Position = [x, y, 0]; % position 0 in 3rd dimension
end
```

- **Example 4**. Mouse callback - display selection type, depending on mouse click

```
function click(h,~)
```

```
    % The callback for a mouse button press. Print the mouse
    % button identifier to the command window.
    h.SelectionType
end
```

- **Example 5**. Mouse callback - select points and plot on a graph

```
function plotPoints1(~,~,h)
    % Call back function for updating a plot

    % Get the current mouse location
    pos = get(gca,'CurrentPoint');

    % Update the plot with the new position
    h.XData(end+1) = pos(1,1);
    h.YData(end+1) = pos(1,2);
    drawnow
end
```

- **Example 6**. Mouse callback - select points and plot depending on right click or left click

```
function plotPoints2(h,~,hRed,hGreen)
    % Callback function for updating a plot with multiple colored points

    % Get the current mouse location.
    pos = get(gca, 'CurrentPoint');

    % If a left click: add to the red plot
    if strcmp(h.SelectionType,'normal')

        % Add to the data in the red plot
        hRed.XData(end+1) = pos(1,1);
        hRed.YData(end+1) = pos(1,2);
        drawnow

    % If a right click: add to the green plot
    elseif strcmp(h.SelectionType,'alt')

        % Add to the data in the red plot
        hGreen.XData(end+1) = pos(1,1);
```

```
        hGreen.YData(end+1) = pos(1,2);
        drawnow

    end
end
```

Example 7. Mouse callback - removing last point

```
function plotPoints3(h,~,hRed,hGreen)
    % Callback function for updating a plot with multiple colored points

    % Get the current mouse location.
    pos = get(gca, 'CurrentPoint');

    % Load GUI information
    handles = guidata(h);

    % If a left click: add to the red plot
    if strcmp(h.SelectionType,'normal')

        % Add to the data in the red plot
        hRed.XData(end+1) = pos(1,1);
        hRed.YData(end+1) = pos(1,2);
        drawnow

        % Specify that the most recent point was red
        handles.order{end+1} = 'r';

    % If a right click: add to the green plot
    elseif strcmp(h.SelectionType,'alt')

        % Add to the data in the red plot
        hGreen.XData(end+1) = pos(1,1);
        hGreen.YData(end+1) = pos(1,2);
        drawnow

        % Specify that the most recent point was red
        handles.order{end+1} = 'g';

    % If a middle click (and note empty: delete the most recent point
    elseif strcmp(h.SelectionType,'extend') && ~isempty(handles.order)
```

```
        % If last point was red
        if strcmp(handles.order{end},'r')
            % Remove last data poitn
            hRed.XData(end) = [];
            hRed.YData(end) = [];
            drawnow

            % Delete last entry
            handles.order(end) = [];

        % If last point was green
        else
            % Remove last data poitn
            hGreen.XData(end) = [];
            hGreen.YData(end) = [];
            drawnow

            % Delete last entry
            handles.order(end) = [];

        end
    end

    % Update the guidata with the data structure
    guidata(h,handles);
end
```

- **Tic Tac Toe code from HW 6**

```
function ticTacToe(compStrategy)
    % Define initial variables and arrays
    gameboard = zeros(3,3); % gameboard that we will populate

    % Repeat: alternate for players
    player = 1; % player one

    % Logical variable to determine if game should stil be playing
    gameStillGoing = 1;

    % Iterate until someone has won (or the game is a draw)
    while gameStillGoing

        % Player 1: human player. If it is player 1's turn, specify a
        % position to be played
```

```
        if player == 1

            % Logical variable corresponding if the move is valid
            invalidMove = 1;

            % Make player pick a valid position
            while invalidMove

                % Position to play, in single index notation
                playPos = input('Select a position to play...');

                if ~isempty(playPos)
                    % Determine if move is allowable
                    invalidMove = all((gameboard(:) == 0)'.*(1:9) ~= playPos);
                end
            end

        % Player 2: computer player. If it is player 2's turn, determine the
        % position played based on the algorithm contained within
compStrategy().
        else
            % Call a function that will output a position based upon the
            % strategy defined for the computer
            playPos = compStrategy(gameboard);
        end

        % Update the gameboard
        gameboard(playPos) = player;

        % Step 3: display game board
        disp(gameboard)

        % Step 4: evaluate stopping condition -> did someone win?
        gameStillGoing = checkEndGame2(gameboard, player);

        % Step 5: change player
        player = mod(player,2)+1;
    end
end
```

```
function out = checkEndGame2(gameboard,pl)
    % The purpose of this function is to determine if anyone has won the
    % game or if a draw has occured.
    % Input 1, gameboard: matrix containing moves
```

```
    % Input 2, which player's turn is it

    % Default condition: game needs to keep going
    out = 1;

    % Each row represents a different way in which the player could win
    testMatrix =
[gameboard;gameboard';diag(gameboard)';diag(rot90(gameboard))'];

    if any(all(testMatrix==pl,2))
        disp(strcat(num2str(pl),' wins!'))
        out = 0;
    elseif sum(gameboard(:)>0) == 9
        disp('Draw!')
        out = 0;
    end
end
```

```
function posPlayed = playRandom(gameboard)
    % This function contains code describing the computer player's strategy
    % Note: Students should optimize this and ensure it is well commented

    % Determine available positions to play
    availablePos = (gameboard(:) == 0)'.*(1:9);
    positions = availablePos(availablePos > 0);

    % Return a random (available) position. This code thus has the computer
    % play randomly based upon the available positions
    posPlayed = positions(randi(length(positions)));
end
```

- **Tic Tac Toe user interface**

```
function placeX(h1,~,x1)
    % The purpose of this function is assign an X once a player has clicked
    % an appropriate location

    % Retrieve current X and Y coordinates
    pos = get(gca,'CurrentPoint');
    pos = pos(1,1:2);

    % Convert the mouse click location into one of 9 possible game play
```

```
    % positions. NOTE: this is based on the visual gameboard created in the
    % above script

    % If click corresponds to index position 3
    if pos(1)<-1 && pos(2)<-1
        playPos = 3;
        % Update position vector. This is what will be assigned to the
        % text that will be added to the figure panel
        pos = [-2.5,-2];

    % If click corresponds to index position 2
    elseif pos(1)<-1 && pos(2)<1
        playPos = 2;
        pos = [-2.5,0];

    % If click corresponds to index position 1
    elseif pos(1)<-1
        playPos = 1;
        pos = [-2.5,2];

    % If click corresponds to index position 6
    elseif pos(1)<1 && pos(2)<-1
        playPos = 6;
        pos = [-0.5,-2];

    % If click corresponds to index position 5
    elseif pos(1)<1 && pos(2)<1
        playPos = 5;
        pos = [-0.5,0];

    % If click corresponds to index position 4
    elseif pos(1)<1
        playPos = 4;
        pos = [-0.5,2];

    % If click corresponds to index position 9
    elseif pos(2)<-1
        playPos = 9;
        pos = [1.5,-2];

    % If click corresponds to index position 8
    elseif pos(2)<1
        playPos = 8;
        pos = [1.5,0];

    % If click corresponds to index position 7
```

```
    else
        playPos = 7;
        pos = [1.5,2];

    end


    % Load gameboard information
    handles = guidata(h1);

    % Update internal gameboard. Note: the click corresponds to player 1
    handles.gameboard(playPos) = 1;

    % Evaluate stopping condition -> did someone win?
    % NOTE: no output is necessary
    checkEndGame3(handles.gameboard,1,h1);

    % Create a matrix containing positional information for all text that
    % has been played
    textPos = reshape([x1.Position],3,5)';

    % Use the above matrix to determine the turn number from the non NaN values
    numTurn = 6-sum(isnan(textPos(:,1)));

    % Update the plot with the new position
    x1(numTurn).Position(1:2) = pos;

    % Save data
    guidata(h1,handles);

end
```

```
function placeO(h1,~,x2,compStrategy)
    % The purpose of this function is assign an O once a player has clicked
    % an appropriate location

    % Load gameboard information
    handles = guidata(h1);

    % Determine position to play based on input strategy
    playPos = compStrategy(handles.gameboard);

    % Update internal gameboard
```

```
    handles.gameboard(playPos) = 2;

    % Determine where to place text
    % Note: this is determined entirely by figure that was created
    if playPos == 1
        pos = [-2.5,2];
    elseif playPos == 2
        pos = [-2.5,0];
    elseif playPos == 3
        pos = [-2.5,-2];
    elseif playPos == 4
        pos = [-0.5,2];
    elseif playPos == 5
        pos = [-0.5,0];
    elseif playPos == 6
        pos = [-0.5,-2];
    elseif playPos == 7
        pos = [1.5,2];
    elseif playPos == 8
        pos = [1.5,0];
    else
        pos = [1.5,-2];
    end

    % Create a matrix containing positional information for all text that
    % has been played
    textPos = reshape([x2.Position],3,4)';

    % Use the above matrix to determine the turn number from the non NaN values
    numTurn = 5-sum(isnan(textPos(:,1)));

    % Update the plot with the new position
    x2(numTurn).Position(1:2) = pos;

    % Step 4: evaluate stopping condition -> did someone win?
    checkEndGame3(handles.gameboard,2,h1);

    % Save data
    guidata(h1,handles);
end
```

```
function checkEndGame3(gameboard,pl,h1)
    % The purpose of this function is to determine if anyone has won the
    % game or if a draw has occured.
```

```
    % Input 1, gameboard: matrix containing moves
    % Input 2, which player's turn is it
    % Input 3, graphics object handle corresponding to figure window

    % Each row represents a different way in which the player could win
    testMatrix =
[gameboard;gameboard';diag(gameboard)';diag(rot90(gameboard))'];

    % If any three in a rows have happened
    if any(all(testMatrix==pl,2))
        % Remove click press and release function handles
        h1.WindowButtonDownFcn = '';
        h1.WindowButtonUpFcn = '';
        % Add text saying that the player won
        x3 = text(-5,5,['Player ',num2str(pl),' wins!']);
        x3.FontSize = 50;
        x3.Color = 'r';


    % If nine turns have occured, then the game is a draw
    elseif sum(gameboard(:)>0) == 9
        % Remove click press and release function handles
        h1.WindowButtonDownFcn = '';
        h1.WindowButtonUpFcn = '';
        % Add text saying that the player won
        x3 = text(-2,5,['Draw!']);
        x3.FontSize = 50;
        x3.Color = 'r';
    end
end
```

- User interface control object to reset game

```
function resetGame(~,~,h,x1,x2,f1,f2,f3)
    % To be assign to some pushbutton for resetting the game

    % Create data structure to store gameboard
    handles.gameboard = zeros(3);
    % Store data structure
    guidata(h,handles);

    % Create two empty text containers. On every turn, we will update the the
    % position of either the next X or next O marker from NaN to the
    % appropriate coordinates
```

```
    for k = 1:4
        x1(k).Position(1:2) = nan(1,2);
        x2(k).Position(1:2) = nan(1,2);
    end
    x1(5).Position(1:2) = nan(1,2);

    % Assign callback function for mouse click
    h.WindowButtonDownFcn = {f1,x1};
    % Assign callback function for mouse release
    h.WindowButtonUpFcn = {f2,x2,f3};


end
```

Topic 23 Functions

Functions for Topic 23 stored here:

Example 1: simple ODE

```
function dxdt = myODE(t,x)
    dxdt = x(1) - t^2 + 1;
end
```

Example 2: coupled chemical reactions

```
function dxdt = rxnEqns(t,x)
    % Here, x(1) : A
    % x(2): B
    % x(3): C
    dxdt = zeros(3,1);
    dxdt(1) = -x(1) + 2*x(2);
    dxdt(2) = x(1) - 4*x(2) + 0.5*x(3);
    dxdt(3) = 2*x(2) - 0.5*x(3);

end
```

Example 3: third order ODE

```
function dxdt = thirdOrderODE(t,x)
```

```
    % Parameters
    p = 2;
    q = 1.5;
    r = 0.1;

    % Note:
    % x(1) corresponds to y
    % x(2) corresponds to y'
    % x(3) corresponds to y''

    % Preallocating the output as a column vector
    dxdt = zeros(3,1);

    dxdt(1) = x(2);
    dxdt(2) = x(3);
    dxdt(3) = -p*x(3)-q*x(2)-r*x(1);
end
```

Example 4: third order ODE, specifying additional inputs

```
function dxdt = thirdOrderODE_extraInputs(t,x,p,q,r)
    dxdt = zeros(3,1);
    dxdt(1) = x(2);
    dxdt(2) = x(3);
    dxdt(3) = -p*x(3)-q*x(2)-r*x(1);
end
```

Example 5: third order ODE, specifying additional inputs

```
function dz = ball_2D(t,z,vx)
    % Compute the x and y position of a ball subject to gravity and
    % in a vacuum. This function is called upon by MATLAB's ODE
    % solvers. t = time (s), z = [y-position, y-velocity, x-position]
    % in units of (m, m/s, m), and dz = [y-velocity; y-acceleration;
    % x-position] in units of (m/s, m/s^2, m).

    dz(1,1) = z(2); % z(1): y-position
    dz(2,1) = -9.81;% z(2): y-velocity
    dz(3,1) = vx;   % z(3): x-position
end
```

Example 6: mouse click function

```
function determineStart(hObj,~,hBall,hTrajectory,hAim)
    % Mouse click: determine kinematics parameters

    % Load information associated with a data structure
    handles = guidata(hObj);

    % Get the current mouse location
    pos = get(gca,'CurrentPoint');
    pos = pos(1,1:2);

    % Specify the inital x and y positions
    hBall.XData = 0;
    hBall.YData = handles.h0;
    hTrajectory.XData = 0;
    hTrajectory.YData = handles.h0;

    % Calculate difference in x and y values
    dy = abs(pos(2) - handles.h0);
    dx = abs(pos(1) - 0);
    % Calculate angle to throw
    handles.th = atan2d(dy,dx);

    % Define initial velocity to be proportional to distance away clicked
    handles.v = 3*sqrt(dy^2 + dx^2);

    % Create a line showing aim
    hAim.XData(2) = pos(1);
    hAim.YData(2) = pos(2);

    % Store object data
    guidata(hObj,handles);

end
```

Example 7: mouse release function

```
function throwBall(hObj,~,hBall,hTrajectory)
    % Mouse release: calculate and visualize trajectory

    % Load GUI data
    handles = guidata(hObj);
    th = handles.th;
```

```
    h0 = handles.h0;
    v = handles.v;

    % Calculate parameters
    vx = v*cosd(th);
    vy = v*sind(th);
    g = 9.81;
    t_end = 2*2*vy/g;

    % Solve system of differential equations. Specify the time points to solve
    % for
    [t,state] = ode45(@ball_2D,[0:0.05:t_end],[2,vy,0],[],vx);
    txy = [t,state(:,3),state(:,1)];
    % Select time points corresponding to the ball not yet landing
    txy = txy(txy(:,3) > 0,:);


    % Loop through all ball positions and update plot
    for k = 2:size(txy,1)

        % Update plot of ball and trajectory tracer
        hBall.XData = txy(k,2);
        hBall.YData = txy(k,3);
        hTrajectory.XData(k) = txy(k,2);
        hTrajectory.YData(k) = txy(k,3);

        % Pause for specified amount of time consisted with time points solved
        % for
        pause(0.05);

    end

end
```

Topic 24 Functions:

```
% function F = eqnSet(x)
%     % System of two non linear equations
%     % F(1) and F(2): "equation" 1 and 2, respectively
%     % x(1) and x(2): variable 1 and 2, respectively
%     F(1) = x(2)-2*x(1)^2+3;
%     F(2) = x(2)^2-x(1);
% end
```

```
function dZdt = springProblem(t,z,b,c)

    dZdt = zeros(2,1);
    dZdt(1) = z(2); % z(1) = x
    dZdt(2) = -b*z(2) - c*z(1); % z(2) = dx/dt

end
```

Topic 25 Functions

Functions for Topic 25 stored here:

- **Example: GUI with counter**

```
function subtractVal(hObj,~,hedit,h1)
    % Subtract 1 from the counter value
    handles = guidata(h1);
    cVal = handles.cVal - 1;

    set(hedit,'String',cVal)
    guidata(handles,h1);
end
```

```
function addVal(hObj,~,hedit,h1)
    % Add 1 to the counter value
    handles = guidata(h1);
    cVal = handles.cVal + 1;

    set(hedit,'String',cVal)
    guidata(handles,h1);
end
```

- **Example: animating a fish**

```
function Topic25DoWhenKeyIsPressed(h1,eventdata,hFish)
    % Utilizes keypresses to move the x/y coordinates of the
    % ball or closes the figure.

    % Load GUI data
```

```
    % Create field to store the current xy direction of fish
    handles = guidata(h1);

    % Movement increment
    inc = 10;

    % Move up
    if strcmp(eventdata.Key,'uparrow')
        hFish.YData = hFish.YData - inc;

    % Move down
    elseif strcmp(eventdata.Key,'downarrow')
        hFish.YData = hFish.YData + inc;

    % Move left
    elseif strcmp(eventdata.Key,'leftarrow')
        % If the current direction is set to move left
        if handles.moveDir == "left"
            hFish.XData = hFish.XData - inc;
        % Otherwise, flip the orientation of the fish
        else
            hFish.XData = fliplr(hFish.XData) - inc;
            handles.moveDir = "left";
        end

    % Move right
    elseif strcmp(eventdata.Key,'rightarrow')
        % If the current direction is set to move left
        if handles.moveDir == "right"
            hFish.XData = hFish.XData + inc;
        % Otherwise, flip the orientation of the fish
        else
            hFish.XData = fliplr(hFish.XData) + inc;
            handles.moveDir = "right";
        end

    % Close figure
    elseif strcmp(eventdata.Key,'q')
        close()
    % Display invalid keystroke
    else
        disp('Key not mapped')
    end

    % Store the data structure as GUI data
    guidata(h1,handles);
```

```
end
```

- **Example: timer function 1**

```
function startItUp(~,~)
    % Timer StartFcn callback.
    disp('A timer has been started.')
    disp('It will wait 2 seconds to fire the first time...')
    disp('and then it will fire every 3 seconds for 5 rounds!')
end
```

```
function sayIt(~,~)
    % Timer fxn callback.
    disp('Hello person!')
end
```

```
function stopIt(t,~)
    % Timer StopFcn callback.
    disp('The timer ''StopFcn'' just executed.')
    disp(['It utilizes the first input of the callback, '...
        'the handle of the timer, to stop the timer.'])
    disp('Always delete timers after you are done using them.')

    delete(t)
end
```

- **Example: falling rock**

```
function dropRock(~,~,hRock,hRockSize)
    % Timer function that is called everytime the rock should be dropped

    % Reset the rock's initial position to be above the water line
    hRock.XData = [1,hRockSize(1)] + randi(1000);
    hRock.YData = [-hRockSize(2),-1];

    % Ensure rock is visible
    hRock.Visible = 'on';
```

```
    % Loop through time steps and move the rock down by some known amount
    % of time
    for k = 1:50
        hRock.YData = hRock.YData + 100;
        pause(0.1)
    end
end
```

```
function dYdt = rockODE(t,y,b,c)

    dYdt = zeros(2,1);
    dYdt(1) = y(2); % y(1) = x
    dYdt(2) = -b*y(2) - c; % z(2) = dy/dt

end
```

```
function dropRock_2(~,~,hRock,hRockSize,y_data)
    % Timer function that is called everytime the rock should be dropped

    % Reset the rock's initial position to be above the water line
    hRock.XData = [1,hRockSize(1)] + randi(1000);
    hRock.YData = [-hRockSize(2),-1];

    initialPos = hRock.YData;

    % Ensure rock is visible
    hRock.Visible = 'on';

    % Loop through time steps and move the rock down by some known amount
    % of time
    for k = 1:length(y_data)
        hRock.YData = initialPos + y_data(k);
        pause(0.05)
    end
end
```

- **Example: drag and drop**

```
function pick(h1,~,D,p)
```

```
    % Mouse button down callback. Determines if the mouse was
    % clicked within the limits of a marker and, if so, takes
    % control of the marker.

    handles = guidata(h1);

    % Get the location of the mouse.
    pos = get(gca,'CurrentPoint');
    xMouse = pos(1,1);
    yMouse = pos(1,2);

    % Determine which marker centerpoint the mouse is closest
    % to and record the vector index number.
    [distance,minInd] = min(sqrt((xMouse-p.XData).^2+(yMouse-p.YData).^2));

    % If the mouse is within a radius of the marker, then
    % take control of the x/y marker centerpoint index and
    % calculate the difference between the centerpoint of the
    % marker and where the user actually clicked.
    if distance <= D/2
        handles.ind = minInd;
    end

    guidata(h1,handles);

end
```

```
function move(h1,~,p)
    % Mouse motion callback. Moves the marker centerpoint
    % held in x/y index 'ind' if 'ind' is not an empty set.
    % Updates visuals accordingly.

    handles = guidata(h1);
    ind = handles.ind;

    % Get the location of the mouse.
    pos = get(gca,'CurrentPoint');
    xMouse = pos(1,1);
    yMouse = pos(1,2);

    % Update the x/y centerpoint corresponding to 'ind'.
    p.XData(ind) = xMouse;
    p.YData(ind) = yMouse;
    drawnow
```

```
end
```

```
function place(h1,~)
    % Mouse up callback. Release control of 'ind'.
    handles = guidata(h1);
    handles.ind = [];
    guidata(h1,handles);
end
```