Problems Week 1

P1-1

In MATLAB, create vectors x = (1,2,3,4,5) and y = (1,2,3,4,5) using any two different methods.

```
x = 1:5;
y = linspace(1, 5, 5);
```

P1-2

Add the two vectors together and examine the output.

```
result = x + y;

disp('ans = ');
```

```
ans =
```

```
disp(result);
```

```
    2     4     6     8    10
```

P1-3

Sum the elements of the answer to question 2.

```
totalSum = sum(result);

disp('ans = ');
```

```
ans =
```

```
disp(totalSum);
```

```
    30
```

P1-4

Add the 3rd element of x to the last element of y, and assign the output to the variable z.

```
z = x(3) + y(end);

disp('z = ');
```

```
z =
```

```
disp(z);
```

```
     8
```

P1-5

Replace the 4th element of x with 7.

```
x(4) = 7;

disp('x = ');
```

```
x =
```

```
disp(x);
```

```
     1     2     3     7     5
```

P1-6

Extend the vector y with the value 6 so that it has 6 elements.

```
y = horzcat(y, 6);


disp('y = ');
```

```
y =
```

```
disp(y);
```

```
    1    2    3    4    5    6
```

P1-7

Shorten the vector y by assigning the empty matrix to element 3.

```
y(3) = [];


disp('y = ');
```

```
y =
```

```
disp(y);
```

```
    1    2    4    5    6
```

P1-8

Create three column vectors of length 3 named c1, c2, and c3. Create a 3x3 matrix called A using these column vectors.

```
c1 = [1; 2; 3];
c2 = [4; 5; 6];
c3 = [7; 8; 9];

A = [c1, c2, c3];
disp('A = ');
```

```
A =
```

```
disp(A);
```

```
    1    4    7
    2    5    8
    3    6    9
```

P1-9

Calculate the determinant of the matrix B.

```
B = [1 2 3; 4 5 6; 7 8 9];
%B = [1, 2, 3;
%     4, 5, 6;
%     7, 8, 9];

determinantB = det(B);


disp('ans: ');
```

```
ans:
```

```
disp(num2str(determinantB));
```

```
6.6613e-16
```

P1-10

Calculate the eigenvalues of the matrix B.

```
eigenvaluesB = eig(B);


disp('ans = ');
```

```
ans =
```

```
disp(eigenvaluesB);
```

```
   16.1168
   -1.1168
   -0.0000
```

P1-11

Create a row vector and a column vector.

```
a = [3, 6, 9];
disp('a = ');
```

```
a =
```

```
disp(a);
```

```
    3    6    9
```

```
b = [3; 6; 9];


disp('b = ');
```

```
b =
```

```
disp(b);
```

```
    3
    6
    9
```

P1-12

Perform matrix and then element-wise multiplication on these two vectors. Can you explain the error in the second case if your Matlab version is before 2017, or can you explain what is happening in later versions of Matlab?

```
% Matrix multiplication
matrixProduct = a * b;

% Element-wise multiplication
elementwiseProduct = a .* b;

disp('Matrix multiplication gives:');
```

```
Matrix multiplication gives:
```

```
disp('ans = ');
```

```
ans =
```

```
disp(matrixProduct);
```

```
   126
```

```
disp('Element-wise multiplication gives (after Matlab 2017):');
```

```
Element-wise multiplication gives (after Matlab 2017):
```

```
disp('ans = ');
```

```
ans =
```

```
disp(elementwiseProduct);
```

```
     9    18    27
    18    36    54
    27    54    81
```

P1-13

Transpose the vector a and create a 3x2 matrix called myMat using it with b.

```
aTransposed = a.';

myMat = [aTransposed, b];

disp('myMat = ');
```

```
myMat =
```

```
disp(myMat);
```

```
     3     3
     6     6
     9     9
```

P1-14

Add the second element on the first row to the first element of the second row.

```
result = myMat(2, 1) + myMat(1, 2);

disp('ans = ');
```

```
ans =
```

```
disp(result);
```

```
     9
```

P1-15 *

Create a random 2x2 array of non-integer numbers and give it a name.
rand(): This function generates random numbers between 0 and 1 (inclusive of 0, exclusive of 1). It can take one or two arguments: rand() generates a single random number; rand(size) generates an array of random numbers with the specified dimensions.

```
x = rand(2, 2);

disp('x = ');
```

```
x =
```

```
disp(x);
```

```
    0.8147    0.1270
    0.9058    0.9134
```

P1-16

Round the array to the nearest integer.

```
xRounded = round(x);

disp('ans = ');
```

```
ans =
```

```
disp(xRounded);
```

```
     1     0
     1     1
```

P1-17

Find the number of elements in the array.

```
numElements = numel(x);

disp('ans = ');
```

```
ans =
```

```
disp(numElements);
```

```
     4
```

P1-18 *

Find its maximum and minimum values.

```
maxValue = max(x(:));
minValue = min(x(:));

disp('ans = ');
```

```
ans =
```

```
disp(maxValue);
```

```
    0.9134
```

```
disp('ans = ');
```

```
ans =
```

```
disp(minValue);
```

```
    0.1270
```

P1-19

Find its maximum and minimum values.

```
maxValue_secondRow = max(x(2, :));

disp('ans = ');
```

```
ans =
```

```
disp(maxValue_secondRow);
```

```
    0.9134
```

P1-20

Create a random vector of non-integers of length 8 and give it a name.

```
randomVector = rand(1, 8);

disp('z = ');
```

```
z =
```

```
disp(randomVector);
```

```
    0.6324    0.0975    0.2785    0.5469    0.9575    0.9649    0.1576    0.9706
```

P1-21

Sort the vector into ascending order.

```
sortedVector = sort(randomVector);

disp('z = ');
```

```
z =
```

```
disp(sortedVector);
```

```
    0.0975    0.1576    0.2785    0.5469    0.6324    0.9575    0.9649    0.9706
```

P1-22

Round the 2nd, 4th, 6th and 8th elements up to the nearest integer, and the 1st, 3rd, 5th and 7th elements down to the nearest integer.

```
roundedVector = sortedVector;

roundedVector([1, 3, 5, 7]) = floor(roundedVector([1, 3, 5, 7]));
roundedVector([2, 4, 6, 8]) = ceil(roundedVector([2, 4, 6, 8]));

disp('z = ');
```

```
z =
```

```
disp(roundedVector);
```

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

P1-23

Generate a random 9x9 matrix of integers called C.

randi(): This function is used to generate random integers from a specified range. It takes one or two arguments: randi([min, max]) generates a single random integer between min and max, inclusive; randi([min, max], size) generates an array of random integers with the specified dimensions.

```
C = randi([1, 100], 9, 9);


disp('C = ');
```

```
C =
```

```
disp(C);
```

```
    96     4    71     4    71    35    55    35    92
    49    85     4    44    76    59    14    20    29
    81    94    28    39    28    23    15    26    76
    15    68     5    77    68    76    26    62    76
    43    76    10    80    66    26    85    48    39
    92    75    83    19    17    51    26    36    57
    80    40    70    49    12    70    82    84     8
    96    66    32    45    50    90    25    59     6
    66    18    96    65    96    96    93    55    54
```

P1-24

Add the 4th column of C to the 6th column.

```
C(:, 6) = C(:, 6) + C(:, 4);


disp('C = ');
```

```
C =
```

```
disp(C(:, 6));
```

```
    39
   103
    62
   153
   106
    70
   119
   135
   161
```

P1-25

Add the 4th column of C to the 6th column.

```
D = C(1:3, 1:3);
E = C(7:9, 6:9);

disp('D = ');
```

```
D =
```

```
disp(D);
```

```
    96     4    71
    49    85     4
    81    94    28
```

```
disp('E = ');
```

```
E =
```

```
disp(E);
```

```
   119    82    84     8
   135    25    59     6
   161    93    55    54
```

P1-26

Perform the element-wise multiplication DE and ED.

```
try
    DE_elementwise = D .* E;
    disp('Element-wise multiplication DE produces:');
    disp(DE_elementwise);
catch
    disp('the dimensions don't match.');
end
```

```
the dimensions don't match.
```

```
try
    ED_elementwise = E .* D;
    disp('Element-wise multiplication ED produces:');
    disp(ED_elementwise);
catch
    disp('the dimensions don't match.');
end
```

```
the dimensions don't match.
```

P1-27

Perform the matrix multiplication DE and ED.

```
try
    DE = D * E;
    disp('DE produces:');
    disp('ans = ');
    disp(DE);
catch
    disp('DE produces an error.');
end
```

```
DE produces:
ans =
       23395       14575       12205        4626
       17950        6515        9351        1118
       26837       11596       13890        2724
```

```
try
    ED = E * D;
    disp('ED produces:');
    disp('ans = ');
    disp(ED);
catch
    disp('ED produces an error.');
end
```

```
ED produces an error.
```

P1-28

Create two character arrays of the same length (number of letters) called s1 and s2.

```
s1 = 'hello';
s2 = 'world';
disp('s1 = ');
```

```
s1 =
```

```
disp(s1);
```

```
hello
```

```
disp('s2 = ');
```

```
s2 =
```

```
disp(s2);
```

```
world
```

P1-29

Create a new character array s3 by concatenating s1 with s2.

```
s3 = [s1 s2];
disp('s3 = ');
```

```
s3 =
```

```
disp(s3);
```

```
helloworld
```

P1-30

Sort the character array s3 into alphabetical order.

```
s3Sorted = sort(s3);


disp('s3 = ');
```

```
s3 =
```

```
disp(s3Sorted);
```

```
dehlllloorw
```

P1-31

Create a vector x = (1,2,3,4,5,6,7,8).

```
x = 1:8;


disp('Vector x:');
```

```
Vector x:
```

```
disp(x);
```

```
     1     2     3     4     5     6     7     8
```

P1-32

Create a vector y = 2x.

```
y = 2 * x;


disp('Vector y:');
```

```
Vector y:
```

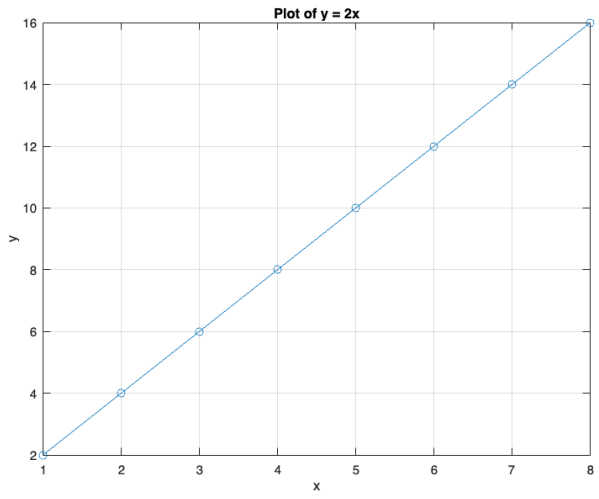
```
disp(y);
```

```
     2     4     6     8    10    12    14    16
```

P1-33

Plot y against x.

```
plot(x, y, 'o-');

xlabel('x');
ylabel('y');
title('Plot of y = 2x');

grid on;
```



P1-34

Create a vector of 100 points from 0 to π called x.

```
n = 100;
x = linspace(0, pi, n);

disp('Vector x:');
```

```
Vector x:
```

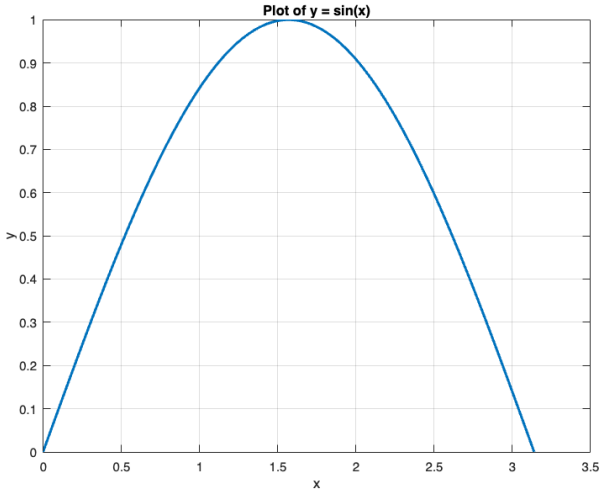
```
disp(x);
```

```
         0    0.0317    0.0635    0.0952    0.1269    0.1587    0.1904    0.2221    0.2539    0.2856    0.3173    0.3491    0.3808    0.4125    0.4443
    0.4760    0.5077    0.5395    0.5712    0.6029    0.6347    0.6664    0.6981    0.7299    0.7616    0.7933    0.8251    0.8568    0.8885    0.9203
    0.9520    0.9837    1.0155    1.0472    1.0789    1.1107    1.1424    1.1741    1.2059    1.2376    1.2693    1.3011    1.3328    1.3645    1.3963
    1.4280    1.4597    1.4915    1.5232    1.5549    1.5867    1.6184    1.6501    1.6819    1.7136    1.7453    1.7771    1.8088    1.8405    1.8723
    1.9040    1.9357    1.9675    1.9992    2.0309    2.0627    2.0944    2.1261    2.1579    2.1896    2.2213    2.2531    2.2848    2.3165    2.3483
    2.3800    2.4117    2.4435    2.4752    2.5069    2.5387    2.5704    2.6021    2.6339    2.6656    2.6973    2.7291    2.7608    2.7925
    2.8243    2.8560    2.8877    2.9195    2.9512    2.9829    3.0147    3.0464    3.0781    3.1099    3.1416
```

P1-35

Plot y = sin(x).

```
y = sin(x);

plot(x, y, 'LineWidth', 2);

xlabel('x');
ylabel('y');
title('Plot of y = sin(x)');

grid on;
```



P1-36

Find out in the documentation how to change the line colour and style.

Line Style

- "-": Solid line
- "--": Dashed line
- ":": Dotted line
- "-.": Dash-dotted line

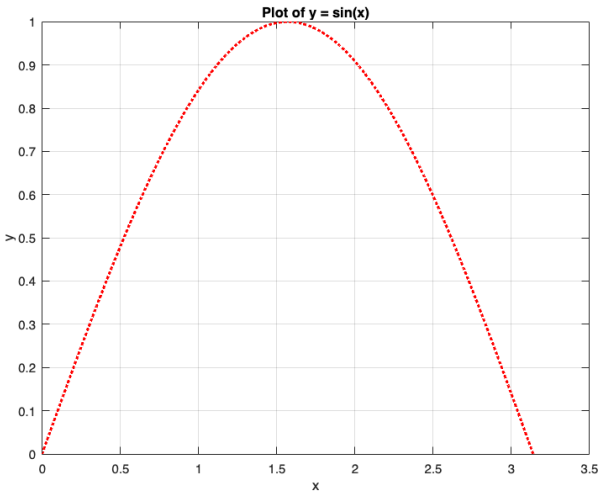
Marker

- "o": Circle
- "+": Plus sign
- "*": Asterisk
- ".": Point
- "x": Cross
- "_": Horizontal line
- "|": Vertical line
- "square": Square
- "diamond": Diamond
- "^": Upward-pointing triangle
- "v": Downward-pointing triangle
- ">": Right-pointing triangle
- "<": Left-pointing triangle
- "pentagram": Pentagram
- "hexagram": Hexagram

Color

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

```
y = sin(x);

plot(x, y, 'r:', 'LineWidth', 2);

xlabel('x');
ylabel('y');
title('Plot of y = sin(x)');

grid on;
```

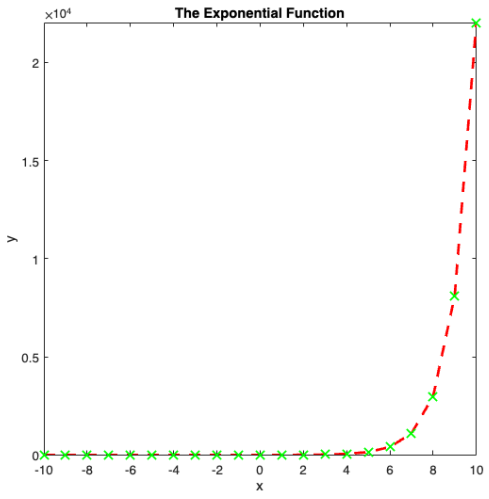


Problems Week 2

P2-1

Plot the exponential function between x = -10 and 10 using a red dashed line, then plot the points at each integer value of x in large green crosses. Remember to add axes labels and a figure title.

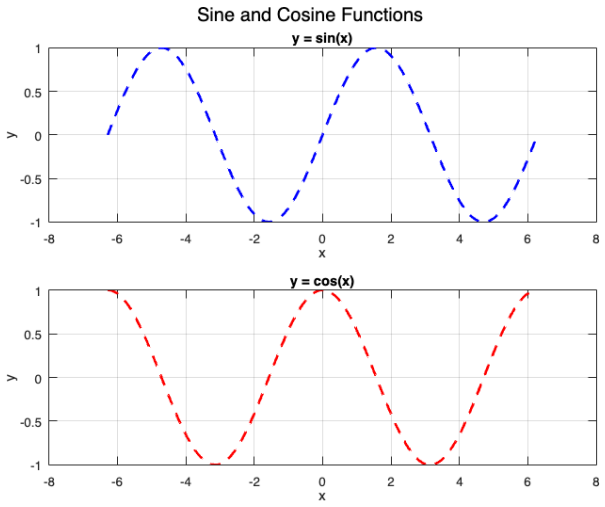
```
x = -10:10;
y = exp(x);

figure;

plot(x, y, 'r--', 'LineWidth', 2);

hold on;

plot(x, y, 'gx', 'MarkerSize', 10, 'LineWidth', 1.5);

hold off;

xlabel('x');
ylabel('y');
title('The Exponential Function');

xlim([-10, 10]);
ylim([0, exp(10)]);
xticks(-10:2:10);
axis square;
```



P2-2

Use 2 subplots above and below each other to plot both sin(x) and cos(x) between -2π and 2π. Customise the plots any way you like.

```
x = linspace(-2*pi, 2*pi, 1000);

ySin = sin(x);
yCos = cos(x);

figure;

subplot(2, 1, 1);
plot(x, ySin, 'b--', 'LineWidth', 2);
title('y = sin(x)');
xlabel('x');
ylabel('y');
grid on;

subplot(2, 1, 2);
plot(x, yCos, 'r--', 'LineWidth', 2);
title('y = cos(x)');
xlabel('x');
ylabel('y');
grid on;

sgtitle('Sine and Cosine Functions');
```

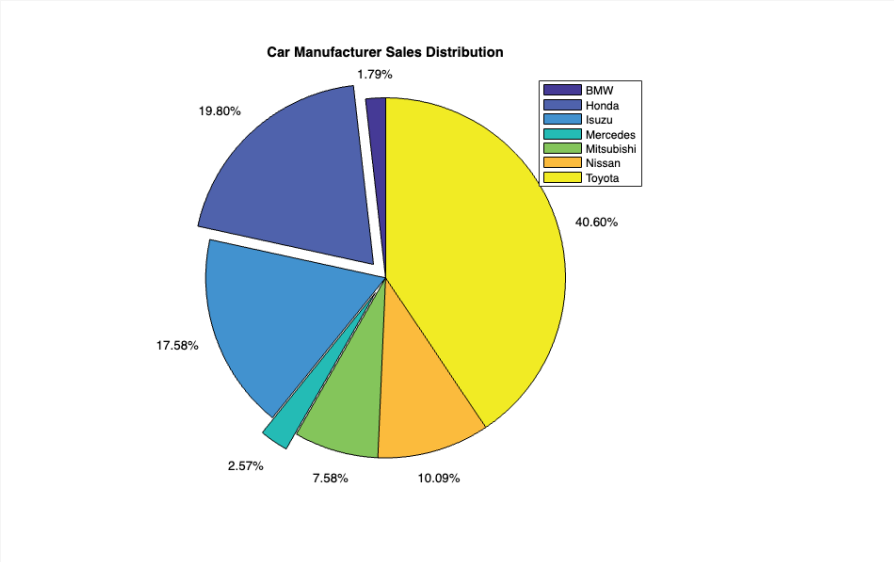


P2-3

Plot the following data in a pie chart and explode the Honda and Mercedes segments.
Syntax

1. pie(X)
2. pie(X,explode)
3. pie(X,labels)
4. pie(X,explode,labels)

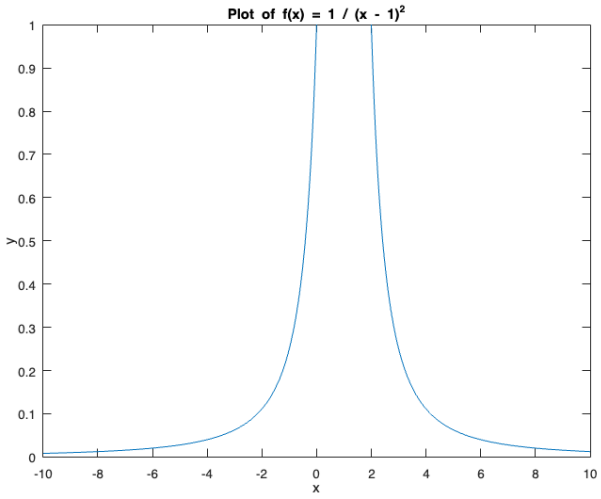
```
carManufacturers = {'BMW', 'Honda', 'Isuzu', 'Mercedes', 'Mitsubishi', 'Nissan', 'Toyota'};
sales = [37210, 412178, 366040, 53442, 157803, 210000, 845213];
explode = [0, 1, 0, 1, 0, 0, 0]; % Explode Honda and Mercedes segments

figure;

p = pie(sales, explode, carManufacturers);

title('Car Manufacturer Sales Distribution');

colorRGB = [69, 58, 150;
            79, 98, 172;
            66, 146, 207;
            36, 187, 181;
            132, 197, 91;
            251, 187, 61;
            241, 235, 36];

colorMap = colorRGB / 255;

colormap(colorMap);

legend(carManufacturers, 'Location', 'best');

for i = 1:length(p)/2
    percentage = sprintf('%.2f%%', 100 * sales(i) / sum(sales));
    p(i * 2).String = percentage;
end
```



P2-4

Plot $\frac{1}{(x-1)^2}$ between -10 and 10, but only display the y-axis between 0 and 1.

```
% '@(x)': This indicates the creation of an anonymous function with input variable x.
% '1 ./ (x - 1).^2': This is the mathematical expression for the function.
f = @(x) 1 ./ (x - 1).^2;

x = -10:0.01:10;

y = f(x);

plot(x, y);

ylim([0, 1]);

xlabel('x');
ylabel('y');
title('Plot of f(x) = 1 / (x - 1)^2');
```



P2-5

Create a character array called x that contains the first 6 letters of the alphabet. The vector should contain 6 separate character elements.

```
x = ["a", "b", "c", "d", "e", "f"];
%x = ['a', 'b', 'c', 'd', 'e', 'f'];

disp('x = ');
```

```
x =
```

```
disp(x);
```

```
    "a"    "b"    "c"    "d"    "e"    "f"
```

P2-6

Display every odd element in the command window with a 2 second pause in between.

```
for i = 1:2:length(x)
    disp(x(i));
    pause(2);
end
```

```
a
c
e
```

```
%Matlab should display a c e in the command window with a gap of 2 seconds in between
```

P2-7

Create a 3 x 3 string matrix called A containing the first 9 letters of the alphabet.

```
alphabetforA = 'adgbehcfi';
A = reshape(alphabetforA, 3, 3);
disp('A = ');
```

```
A =
```

```
disp(A);
```

```
abc
def
ghi
```

P2-8

Print the diagonal entries of *A* with a 3 second pause in between.

```
for i = 1:3
    disp(A(i, i));
    pause(3);
end
```

```
a
e
i
```

```
%Matlab should display a e i in the command window with a gap of 3 seconds in between
```

P2-9

Create a 1 x 8 vector of random integers called *y*.

```
minValue = 0;
maxValue = 10;
y = randi([minValue, maxValue], 1, 8);
disp('y: ');
```

```
y:
```

```
disp(y);
```

```
    1    6    5    0    3    1    8    3
```

P2-10

Write a for loop that replaces elements of *y* that have an even value with 0 and elements that have an odd value with 1.

```
for i = 1:length(y)
    if mod(y(i), 2) == 0
        y(i) = 0;
    else
        y(i) = 1;
    end
end
disp('y: ');
```

```
y:
```

```
disp(y);
```

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

P2-11

Create a 6 x 1 vector of random integers between 20 and 50 called *z*.

```
z = randi([20, 50], 6, 1);
disp('z = ');
```

```
z =
```

```
disp(z);
```

```
    36
    25
    38
    28
    40
    41
```

P2-12

Write a for loop that subtracts 12 from an element of *z* that has a value between 27 and 43.

```
for i = 1:length(z)
    if z(i) >= 27 && z(i) <= 43
        z(i) = z(i) - 12;
    end
end
disp('z: ');
```

```
z:
```

```
disp(z);
```

```
    24
    25
    26
    16
    28
    29
```

P2-13

Write a for loop that doubles the 2nd, 4th and 5th entries of *y*.

```
for i = [2, 4, 5]
    y(i) = y(i) * 2;
end
disp('y: ');
```

```
y:
```

```
disp(y);
```

```
    1    0    1    0    2    1    0    1
```

P2-14

Repeat the task in question without using a loop (just vector notation).

```
indices2double = [2, 4, 5];
y(indices2double) = y(indices2double) * 2;

disp('y: ');
```

```
y:
```

```
disp(y);
```

```
    1    0    1    0    4    1    0    1
```

P2-15

Write a while loop that multiplies a number by 2 until the answer is greater than 100. Test your code with numbers of your choice.

```
number = 7;

while number <= 100
    number = number * 2;
    disp("number: " + number);
end
```

```
number: 14
number: 28
number: 56
number: 112
```

```
disp("Final number: " + number);
```

```
Final number: 112
```

```
%For example if you start with 7 then the final number before 100 is reached is 56.
```

P2-16

Write a while loop that decreases a number by 10 until it is less than 0.

```
number = 27;

while number >= 0
    number = number - 10;
    disp("number: " + number);
end
```

```
number: 17
number: 7
number: -3
```

```
disp("Final number: " + number);
```

```
Final number: -3
```

```
%For example if you start with 27 then the final number before 0 is reached is 7.
```

P2-17

Write a function that accepts a number as an argument. Check if it is positive or negative. If it is positive then double it, if it is negative then halve it. Return the value. Remember to give your function a descriptive name.

```
input1 = 18;
output1 = doubleIfPositiveHalveIfNegative(input1);
disp("Output for " + input1 + ": " + output1);
```

```
Output for 18: 36
```

```
input2 = -18;
output2 = doubleIfPositiveHalveIfNegative(input2);
disp("Output for " + input2 + ": " + output2);
```

```
Output for -18: -9
```

```
%For example if the input is 18 then the output should be 36, and if the input is -18 then the output should be -9.
```

P2-18

Write a function that accepts a number as an argument. Check if it is even or odd. If it is even, add 1 to it. If it is odd, subtract 1 from it. Return the value.

```
input1 = 18;
output1 = addOneIfEvenSubtractOneIfOdd(input1);
disp("Output for " + input1 + ": " + output1);
```

```
Output for 18: 19
```

```
input2 = 17;
output2 = addOneIfEvenSubtractOneIfOdd(input2);
disp("Output for " + input2 + ": " + output2);
```

```
Output for 17: 16
```

```
%For example if the input is 18 then the output should be 19, and if the input is 17 then the output should be 16.
```

P2-19

Write a function that accepts 2 numbers as arguments. Check if they are equal to each other. If they are, then add them. If they are not, then return the absolute value of their difference.

```
num1 = 2;
num2 = 2;
result = compareAndCompute(num1, num2);
disp("Result: " + result);
```

```
Result: 4
```

```
num1 = 2;
num2 = 3;
result = compareAndCompute(num1, num2);
disp("Result: " + result);
```

```
Result: 1
```

```
%For example if the inputs are 2 and 2 then the output should be 4, if the inputs are 2 and 3 then the output should be 1.
```

P2-20

Write a function that accepts 2 numbers as arguments. If their sum is not equal to 10 then return false. If it is equal to 10, return true.

```
num1 = 6;
num2 = 4;
result = checkSumEquals10(num1, num2);
disp("Result: " + result);
```

```
Result: true
```

```
num1 = 3;
num2 = 11;
result = checkSumEquals10(num1, num2);
disp("Result: " + result);
```

```
Result: false
```

```
%For example if the inputs are 6 and 4 then the output should be true (or 1), and if the inputs are 3 and 11 then the output should be false (or 0).
```

P2-21

Write a function that accepts 3 numbers as arguments. Add the largest 2 numbers together. Return the result.

```
num1 = 1;
num2 = -2;
num3 = 3;
result = addLargestTwo(num1, num2, num3);
disp("Result: " + result);
```

```
Result: 4
```

```
%For example if the inputs are 1,-2 and 3 then the output should be 4.
```

P2-22

Write a function that accepts 3 numbers as arguments. Subtract the smallest number from the largest. Return the result.

```
num1 = 1;
num2 = -2;
num3 = 3;
result = subtractSmallestFromLargest(num1, num2, num3);
disp("Result: " + result);
```

```
Result: 5
```

```
%For example if the inputs are 1,-2 and 3 then the output should be 5.
```

P2-23

Write a function that accepts a vector as an input argument. The function should print the number of elements of the input vector as well as its smallest 2 values. If the vector has less than 2 elements the function should tell the user it requires more inputs.

```
vector1 = [-4];
vector2 = [-4, -2, 0, 2, 4];

printVectorInfo(vector1);
```

```
Requires at least 2 inputs
```

```
printVectorInfo(vector2);
```

```
No. of elements is 5 and the smallest two values are -4 and -2
```

```
%For example if the input vector is [-4] then the function should display "Requires at least 2 inputs"
%but if the input vector is [-4 -2 0 2 4] then the function should display "No. of elements is 5 and the smallest two values are -4 and -2".
```

P2-24

Write a script that creates a random vector of 1's and 0's then finds the indexes of the non-zero elements (read about the find function).

```
randomVector = randi([0, 1], 1, 9);
%randomVector = [1 0 0 1 1 0 0 1 0];

nonZeroIndexes = find(randomVector);

disp('Random Vector:');
```

```
Random Vector:
```

```
disp(randomVector);
```

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

```
disp('Indexes of Non-Zero Elements:');
```

```
Indexes of Non-Zero Elements:
```

```
disp(nonZeroIndexes);
```

```
     1     5     7     8     9
```

```
%For example if the random vector is [1 0 0 1 1 0 0 1 0] then the output should be the vector [1 4 5 8].
```

P2-25

Write a script which creates a random 4 x 4 matrix of integers, then finds the even numbers and sets them equal to 0. The final matrix should be only odd numbers and 0 values.

```
randomMatrix = randi([1, 9], 4, 4);
%randomMatrix = [1 3 4 9; 2 2 7 7; 3 1 4 5; 7 9 8 1];


disp('Original Matrix:');
```

```
Original Matrix:
```

```
disp(randomMatrix);
```

```
     1     1     1     4
     4     7     4     9
     1     8     3     2
     9     8     8     3
```

```
evenIndexes = mod(randomMatrix, 2) == 0;
randomMatrix(evenIndexes) = 0;
disp('Modified Matrix:');
```

```
Modified Matrix:
```

```
disp(randomMatrix);
```

```
     1     1     1     0
     0     7     0     9
     1     0     3     0
     9     0     0     3
```

```
%For example if the random matrix is
%1 3 4 9
%2 2 7 7
%3 1 4 5
%7 9 8 1
%then the output should be the matrix
%1 3 0 9
%0 0 7 7
%3 1 0 5
%7 9 0 1
```

P2-26

Perform the element-wise multiplication DE and ED.

```
n = 4;
result = createMatrix(n);
disp('result:');
```

```
result:
```

```
disp(result);
```

```
     0     0     9     0
     0     0     9     7
     0     0     5     1
     3     0     0     3
```

P2-27

Make a new function which takes two input arguments, *n* and *m*, which define an *n* x *m* matrix. Set the odd numbers equal to 0.

```
n = 4;
m = 3;
result = createMatrixWithOddZeros(n, m);
disp('result:');
```

```
result:
```

```
disp(result);
```

```
     0     0    10
    10     0     0
     4    10     4
     2     8     4
```

P2-28

Create a function which takes a value, *x*, and matrix, *A*, as inputs. It will set any entries of the input matrix greater than *x* equal to 0 and return the result.

```
A = [1 3 4 9; 2 2 7 7; 3 1 4 5; 7 9 8 1];
x = 5;
result = setGreaterThanXToZero(x, A);
disp('result:');
```

```
result:
```

```
disp(result);
```

```
     1     3     4     0
     2     2     0     0
     3     1     4     5
     0     0     0     1
```

```
%For example if the random matrix is
%1 3 4 9
%2 2 7 7
%3 1 4 5
%7 9 8 1
%and the value of x is 5 then the output should be the matrix
%1 3 4 0
%2 2 0 0
%3 1 4 5
%0 0 0 1
```

P2-29

The following series converges to 1. Write a function that accepts a number specifying a tolerance, tol, as input argument. Calculate the sum of the series, until the answer is within tol away from 1. Display how many terms are required for the given precision. $S = \sum_{i=1}^{n} \frac{1}{2^i}$

```
calculateSeriesTolerance(0.3);
```

```
Number of terms required for tolerance 0.300000: 2
```

```
calculateSeriesTolerance(0.04);
```

```
Number of terms required for tolerance 0.040000: 5
```

```
calculateSeriesTolerance(0.0005);
```

```
Number of terms required for tolerance 0.000500: 11
```

```
%For example if the tolerance is 0.3 then n should be 2. If the tolerance is 0.04 then n should be 5. If the tolerance is 0.0005 then n should be 11.
```

P2-30

Write a function which accepts a number, *n*, as input argument. Calculate Euler's number up to and including *n* terms *using* the following recursive formula. $e = \sum_{i=0}^{n} \frac{1}{i!}$

```
n = 3;
eApproximation = calculateEuler(n);
fprintf('Approximation of e with n = %d is %.15f\n', n, eApproximation);
```

```
Approximation of e with n = 3 is 2.666666666666667
```

```
n = 7;
eApproximation = calculateEuler(n);
fprintf('Approximation of e with n = %d is %.15f\n', n, eApproximation);
```

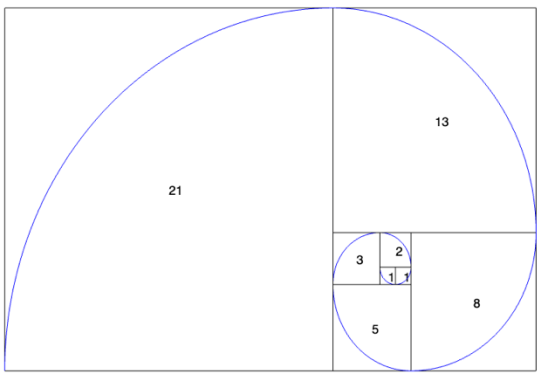
```
Approximation of e with n = 7 is 2.718253968253968
```

```
%For example if n = 3 the approximation is 2.666666666666667. If n = 7 the approximation is 2.718253968253968.
```

P2-31

The golden spiral can be approximated using the Fibonacci sequence:
1, 1, 2, 3, 5, 8, 13, 21, 34…
where each number is the sum of the previous 2 numbers.
Recreate the Fibonacci spiral shown on the next page by plotting squares as shown with lengths equal to the numbers in the Fibonacci sequence, and then adding quarter circular arcs, which join opposite corners of each square, to the plot. The file should be a function or script that specifies the number of terms in the Fibonacci sequence and produces the corresponding plot. Hint: You will probably need to use pen and paper to figure out the coordinates of each of the squares. Also to plot a square you must use 5 points. The first and last coordinate points must be the same!

```
x = 0;
y = 1;
syms v u

axis off
hold on

for n = 1:8
    a = fibonacci(n);
    % Define squares and arcs
    switch mod(n,4)
        case 0
            y = y - fibonacci(n-2);
            x = x - a;
            eqnArc = (u-(x+a))^2 + (v-y)^2 == a^2;
        case 1
            y = y - a;
            eqnArc = (u-(x+a))^2 + (v-(y+a))^2 == a^2;
        case 2
            x = x + fibonacci(n-1);
            eqnArc = (u-x)^2 + (v-(y+a))^2 == a^2;
        case 3
            x = x - fibonacci(n-2);
            y = y + fibonacci(n-1);
            eqnArc = (u-x)^2 + (v-y)^2 == a^2;
    end

    % Draw square
    pos = [x y a a];
    rectangle('Position', pos)
    % Add Fibonacci number
    xText = (x+x+a)/2;
    yText = (y+y+a)/2;
    text(xText, yText, num2str(a))

    % Draw arc
    interval = [x x+a y y+a];
    fimplicit(eqnArc, interval, 'b')
end
```



Problems Week 3

P3-1

Create a cell array called 'myCell' where the first element is a random 3 x 4 matrix of integers, the second element is a vector with 200 elements ranging from -10 to 10, and the third element is a string of your choice.

```
myCell = {randi([-10, 10], 3, 4), linspace(-10, 10, 200), "hearty radish"};
```

P3-2

Request user input for indexes of an element in the 3 x 4 matrix. Your code should check that the user entered a valid pair of numbers and display a warning message if not.

```
validInput = false;
while ~validInput
    rowIndex = input('Enter the row index (1-3): ');
    colIndex = input('Enter the column index (1-4): ');

    if isnumeric(rowIndex) && isnumeric(colIndex) && ...
            numel(rowIndex) == 1 && numel(colIndex) == 1 && ...
            rowIndex >= 1 && rowIndex <= 3 && colIndex >= 1 && colIndex <= 4
        validInput = true;
    else
        disp('Invalid input. Please enter valid row and column indexes.');
    end
end


selectedElement = myCell{1}(rowIndex, colIndex);
disp(['Selected element: ' num2str(selectedElement)]);
```

```
Selected element: -4
```

P3-3

Create a new vector that is the vector in the cell array multiplied by the number in the matrix specified by the user, then insert this new vector as the third element of the cell array and shift the previous third element (string) to the fourth position of the cell array.

```
newVector = myCell{2} * selectedElement;
myCell{3} = newVector;
myCell{4} = myCell{3};
disp('myCell = ');
```

```
myCell =
```

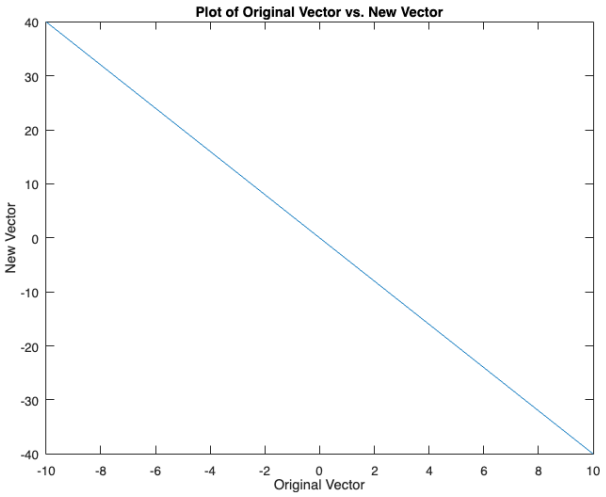
```
disp(myCell);
```

```
    {3×4 double}    {[-10 -9.8995 -9.7990 -9.6985 -9.5980 -9.4975 … 9.5980 9.6985 9.7990 9.8995 10]}    {[40 39.5980 39.1960 38.7940 38.3920 … -38.3920 -38.7940 -39.1960 -39.5980 -40]}    {[40 39.5980 39.1960 38.7940 38.3920 … -38.3920 -38.7940 -39.1960 -39.5980 -40]}
```

P3-4

Plot the 2 vectors from your cell array against each other in one line of code.

```
plot(myCell{2}, myCell{3});
xlabel('Original Vector');
ylabel('New Vector');
title('Plot of Original Vector vs. New Vector');
```



P3-5

Create a structure called myStruct to save the following data:

- Name, Age, Height (cm), Mass (kg)
- Harry, 36, 170, 80
- Georgia, 21, 181, 70
- Elizabeth, 78, 158, 65

```
myStruct = struct('Name', {'Harry', 'Georgia', 'Elizabeth'}, ...
                  'Age', {36, 21, 78}, ...
                  'Height', {170, 181, 158}, ...
                  'Mass', {80, 70, 65});
myStruct
```

myStruct = 1×3 struct

Fields	Name	Age	Height	Mass
1	'Harry'	36	170	80
2	'Georgia'	21	181	70
3	'Elizabeth'	78	158	65

P3-6

Change Georgia's mass to be 68 kg.

```
myStruct(2).Mass = 68;
myStruct
```

myStruct = 1×3 struct

Fields	Name	Age	Height	Mass
1	'Harry'	36	170	80
2	'Georgia'	21	181	68
3	'Elizabeth'	78	158	65

P3-7

Add an extra person to the data with the following details:

- Name, Lily; Age, 24; Height, 162 cm; Mass, 60 kg.

```
newPerson = struct('Name', 'Lily', 'Age', 24, 'Height', 162, 'Mass', 60);
myStruct = [myStruct, newPerson];
myStruct
```

myStruct = 1×4 struct

Fields	Name	Age	Height	Mass
1	'Harry'	36	170	80
2	'Georgia'	21	181	68
3	'Elizabeth'	78	158	65
4	'Lily'	24	162	60

P3-8

Using one line of code, calculate the mean height of the group.

```
meanHeight = mean([myStruct.Height]);
disp('meanHeight: ')
```

meanHeight:

```
disp(meanHeight)
```

```
  167.7500
```

P3-9

BMI is given by Mass (kg) divided by Height (metres) squared. Calculate the BMI of each person in the group and make a new field in your structure that saves this data for each person.

```
for i = 1:length(myStruct)
    heightInMeters = myStruct(i).Height / 100;
    bmi = myStruct(i).Mass / (heightInMeters^2);
    myStruct(i).BMI = bmi;
end
myStruct
```

myStruct = 1×4 struct

Fields	Name	Age	Height	Mass	BMI
1	'Harry'	36	170	80	27.6817
2	'Georgia'	21	181	68	20.7564
3	'Elizabeth'	78	158	65	26.0375
4	'Lily'	24	162	60	22.8624

P3-10

Use the sprintf function to display the names of the people in the group along with their age, height, mass and BMI displaying each value to 2 decimal places).

```
for i = 1:length(myStruct)
    sprintf('Name: %s, Age: %d, Height: %.2f cm, Mass: %.2f kg, BMI: %.2f\n', myStruct(i).Name, myStruct(i).Age, myStruct(i).Height, myStruct(i).Mass, myStruct(i).BMI)
end
```

```
ans =
    'Name: Harry, Age: 36, Height: 170.00 cm, Mass: 80.00 kg, BMI: 27.68
     '

ans =
    'Name: Georgia, Age: 21, Height: 181.00 cm, Mass: 68.00 kg, BMI: 20.76
     '

ans =
    'Name: Elizabeth, Age: 78, Height: 158.00 cm, Mass: 65.00 kg, BMI: 26.04
     '

ans =
    'Name: Lily, Age: 24, Height: 162.00 cm, Mass: 60.00 kg, BMI: 22.86
```

P3-11

Write a script that displays the number 12345.987654321 in the following formats (include leading 0's if the width is longer than the number):

- (a) Width 10 precision 3.
- (b) Width 10 precision 6.
- (c) Width 8 precision 6.
- (d) Width 14 precision 6.
- (e) What happens if you use a higher precision than there are decimal places (e.g. precision 12)?

```
num = 12345.987654321;

fprintf('(a) Width 10 precision 3: %10.3f\n', num);
```

```
(a) Width 10 precision 3:  12345.988
```

```
fprintf('(b) Width 10 precision 6: %10.6f\n', num);
```

```
(b) Width 10 precision 6: 12345.987654
```

```
fprintf('(c) Width 8 precision 6: %8.6f\n', num);
```

```
(c) Width 8 precision 6: 12345.987654
```

```
fprintf('(d) Width 14 precision 6: %14.6f\n', num);
```

```
(d) Width 14 precision 6:    12345.987654
```

```
fprintf('(e) Precision 12: %12.12f\n', num);
```

```
(e) Precision 12: 12345.987654320999
```

P3-12

Write a script that finds and displays the largest *x* that can be input to the exponential function (*ex*) before an infinite result is reached.

```
threshold = realmax;

x = 0;
result = 1;

while result < threshold
    x = x + 1;
    result = exp(x);
end

x = x - 1;

fprintf('The largest input to the exponential function is: %d\n', x);
```

```
The largest input to the exponential function is: 709
```

P3-13

Write a script that asks the user for 2 inputs. Prompt the user for a random sentence. Then prompt the user for a letter of the alphabet. Your script should then display how many times that letter appears in their sentence and lists the words in which it appears.

```
sentence = input('Write a random sentence: ', 's');

letter = input('Pick a letter of the alphabet: ', 's');

count = 0;
words = {};

sentence = lower(sentence);

wordArray = strsplit(sentence);

for i = 1:length(wordArray)
    word = wordArray{i};
    if contains(word, letter)
        count = count + 1;
        words{end+1} = word;
    end
end

fprintf('The number of matching words is %d.\n', count);
```

```
The number of matching words is 1.
```

```
if count > 0
    disp('The maching words are: ');
    disp(words);
end
```

```
The maching words are:
    {'tete'}
```

P3-14

Load the datafile "DOB.mat" into the Matlab workspace then create a categorical array of the generations defined as follows:

- "Boomer" born in 1946-1964
- "Gen X" born in 1965-1980
- "Millennials" born in 1981-1996
- "Gen Z" born in 1997-1997-2012
- "Gen A" born in 2012-present

(Hint: Use the discretize() function)

Display how many of each generation are in the data set then create a table that contains columns for each day, month, date, year and generation. Display the first 10 table entries. Now display the date of births of all Gen A. Lastly, locate all people born in July and display the corresponding year of their births, the day on which they were born, and their generation.

```
% Load the data from the "DOB.mat" file
load('DOB.mat');

% Define generation boundaries
generation_edges = [1946, 1965, 1981, 1997, 2012, inf];

% Define generation names
generation_names = {'Boomer', 'Gen X', 'Millennials', 'Gen Z', 'Gen A'};

% Initialize arrays to store the parsed date components
years = zeros(size(DOB, 1), 1);
months = cell(length(DOB), 1);
dates = zeros(size(DOB, 1), 1);
days = cell(length(DOB), 1);

% Convert cell array of DOB strings to datetime and extract date components
for i = 1:length(DOB)
    dob_datetime = datetime(DOB{i}, 'InputFormat', 'eeee, MMMM d, yyyy');
    years(i) = year(dob_datetime);
    monthes_name = char(month(dob_datetime, 'name'));
    months{i} = monthes_name;
    dates(i) = day(dob_datetime);
    day_of_week = char(day(dob_datetime, 'name'));
    days{i} = day_of_week;
end

% Discretize the years of birth into generations
generations = discretize(years, generation_edges, generation_names);
```

```
% Convert categorical generations to cell array of strings
generations = cellstr(generations);

% Display the count of each generation
generation_counts = histcounts(years, generation_edges);
disp('Summary:');
```

```
Summary:
```

```
for i = 1:length(generation_names)
    fprintf('%s: %d\n', generation_names{i}, generation_counts(i));
end
```

```
Boomer: 137
Gen X: 99
Millennials: 102
Gen Z: 96
Gen A: 66
```

```
% Create a table with required columns
DOBTable = table(days, months, dates, years, generations, 'VariableNames', {'Day', 'Month', 'Date', 'Year', 'Generation'});

% Display the first 10 entries in the table
disp('First 10 Entries:');
```

```
First 10 Entries:
```

```
disp(DOBTable(1:10, :));
```

```
    Day           Month       Date    Year      Generation
    ___________   _____________   ____   ____   ______________

    {'Thursday' }    {'July'     }      7     1955    {'Boomer'     }
    {'Monday'   }    {'October'  }     22     1951    {'Boomer'     }
    {'Wednesday'}    {'February' }     14     2018    {'Gen A'      }
    {'Saturday' }    {'November' }     30     2002    {'Gen Z'      }
    {'Tuesday'  }    {'June'     }     21     1994    {'Millennials'}
    {'Saturday' }    {'August'   }     23     1997    {'Gen Z'      }
    {'Friday'   }    {'January'  }     10     1958    {'Boomer'     }
    {'Friday'   }    {'April'    }      3     1953    {'Boomer'     }
    {'Monday'   }    {'November' }     11     1946    {'Boomer'     }
    {'Friday'   }    {'September'}     13     2013    {'Gen A'      }
```

```
% Display the date of birth of all Gen A individuals
disp('Gen A:');
```

```
Gen A:
```

```
genA_indices = find(strcmp(generations, 'Gen A'));
disp(DOB(genA_indices));
```

```
    {["Wednesday, February 14, 2018" ]}
    {["Friday, September 13, 2013"   ]}
    {["Monday, January 9, 2012"      ]}
    ...

    {["Wednesday, October 21, 2020"  ]}
    {["Monday, October 26, 2020"     ]}
    {["Wednesday, January 16, 2019"  ]}
```

```
% Locate all people born in July and display their birth year, day, and generation
july_indices = find(months == "July");
july_table = DOBTable(july_indices, {'Year', 'Day', 'Generation'});
disp('People Born in July:');
```

```
People Born in July:
```

```
disp(july_table);
```

```
    Year        Day           Generation
    ____    ____________    _______________

    1955    {'Thursday' }    {'Boomer'     }
    2020    {'Friday'   }    {'Gen A'      }
    1964    {'Saturday' }    {'Boomer'     }
    ...
    1966    {'Wednesday'}    {'Gen X'      }
    1992    {'Wednesday'}    {'Millennials'}
    1971    {'Monday'   }    {'Gen X'      }
```

P3-15

The following code performs some tasks in Matlab using a for loop. Vectorise the code to produce the same result without using any loops.

```
tstart=0; tend=20; ni=8;
t(1)=tstart;
y(1)=12 + 6*cos(2*pi*t(1)/(tend-tstart));
for i=2:ni+1
  t(i)=t(i-1)+(tend-tstart)/ni;
  y(i)=12 + 6*cos(2*pi*t(i)/ ...
       (tend-tstart));
end
```

```
tstart = 0;
tend = 20;
ni = 8;

% Create a vector of time values
t = linspace(tstart, tend, ni);

% Calculate y using vectorized operations
y = 12 + 6 * cos(2 * pi * t / (tend - tstart));
```

P3-16

Create a random 4 x 4 matrix, ***A***, with values between 0 and 1 then swap the 2nd and 4th row using one line of code.

```
A = rand(4, 4); A([2, 4], :) = A([4, 4], :)
```

```
A = 4×4
    0.8176    0.8116    0.8759    0.2077
    0.3786    0.9390    0.5870    0.2305
    0.6443    0.3507    0.6225    0.4709
    0.7948    0.5328    0.5502    0.3012
```

P3-17

Using the matrix, ***A***, from Q16 now swap the 1st and 4th columns using one line of code.

```
A(:, [1, 4]) = A(:, [4, 1]);
disp('A = ');
```

```
A =
```

```
disp(A);
```

```
    0.2077    0.8116    0.8759    0.8176
    0.2305    0.9390    0.5870    0.3786
    0.4709    0.3507    0.6225    0.6443
    0.3012    0.5328    0.5502    0.7948
```

P3-18

Using one line of code, set every value of A that is greater than 0.5 equal to 7.

```
A(A > 0.5) = 7;
disp('A = ');
```

```
A =
```

```
disp(A);
```

```
    0.2077    7.0000    7.0000    7.0000
    0.2305    7.0000    7.0000    0.3786
    0.4709    0.3507    7.0000    7.0000
    0.3012    7.0000    7.0000    7.0000
```

P3-19

Create a 4 x 7 matrix, B, of random integers between 10 and 50 then using one line of code set every value of B that is both greater than 20 and less than 40 equal to 0 (Hint: Use the element-wise logical operators, '&' and '|').

```
B = randi([10, 50], 4, 7); B((B > 20) & (B < 40)) = 0;
disp('B = ');
```

```
B =
```

```
disp(B);
```

```
    44    19     0     0     0    19     0
    17     0    17    14    20    14     0
    19     0    47    20     0     0    13
    16    47    50     0     0     0    20
```

P3-20

Using one line of code insert the column vector $[3\ -2\ 4\ 8]^T$ in between the 2nd and 3rd columns of ***B***.

```
B = [B(:, 1:2), [3; -2; 4; 8], B(:, 3:end)];
disp('B = ');
```

```
B =
```

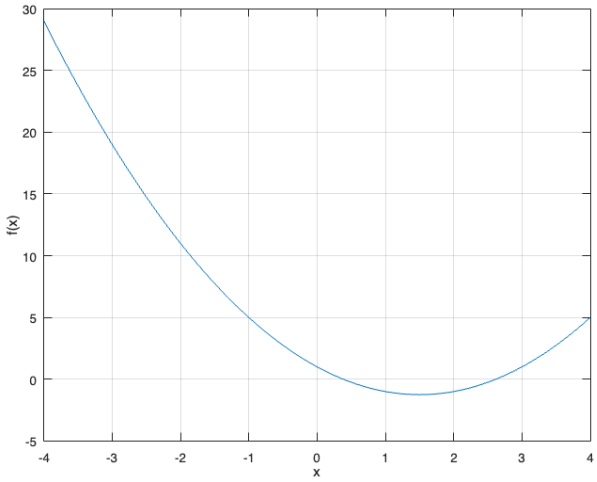
```
disp(B);
```

```
    44    19     3     0     0     0    19     0
    17     0    -2    17    14    20    14     0
    19     0     4    47    20     0     0    13
    16    47     8    50     0     0     0    20
```

P3-21

Write a script that creates an anonymous function for the function $f(x) = x^2 - 3x + 1$ then plots it between -4 and 4 with a grid.

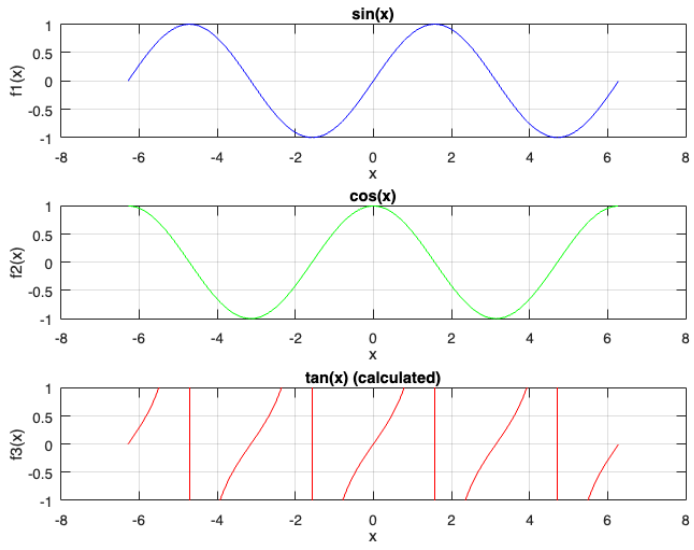
```
% Create an anonymous function for f(x)
f = @(x) x.^2 - 3*x + 1;

% Define the range of x values
x = -4:0.1:4;

% Calculate the corresponding y values using the anonymous function
y = f(x);

figure;

% Plot the function
plot(x, y);
grid on;

xlim([-4, 4]);

% Label the axes
xlabel('x');
ylabel('f(x)');
```



P3-22

Write a script that creates 2 anonymous functions, f1 and f2, for sin(x) and cos(x), then a 3rd anonymous function, f3, equal to the first function divided by the second function (f1/f2 = sin(x)/cos(x) = tan(x)). The script should then plot all 3 functions in different colours using subplots over the domain $-2\pi < x < 2\pi$. Set the axes on the graph to have y-limits between -1 and 1.

```
% Define the range of x values
x = -2*pi:0.01:2*pi;

% Create anonymous functions for sin(x) and cos(x)
f1 = @(x) sin(x);
f2 = @(x) cos(x);

% Calculate f3 = sin(x)/cos(x) without explicitly defining it
f3 = @(x) f1(x) ./ f2(x);

figure;

% Create subplots
subplot(3,1,1);
plot(x, f1(x), 'b');
grid on;
title('sin(x)');
ylabel('f1(x)');
xlabel('x');
ylim([-1, 1]);

subplot(3,1,2);
plot(x, f2(x), 'g');
grid on;
title('cos(x)');
ylabel('f2(x)');
xlabel('x');
ylim([-1, 1]);

subplot(3,1,3);
plot(x, f3(x), 'r');
grid on;
title('tan(x) (calculated)');
ylabel('f3(x)');
xlabel('x');
ylim([-1, 1]);
```



P3-23

Create an anonymous function, f4, that takes 2 inputs, *x* and *y*, then calculates *xy*. Test your function with inputs x = 3 and y = 7.

```
% Create an anonymous function that multiplies x and y
f4 = @(x, y) x.^y;

% Test the function with inputs x = 3 and y = 7
x = 3;
y = 7;
result = f4(x, y);

% Display the result
disp('ans = ')
```

```
ans =
```

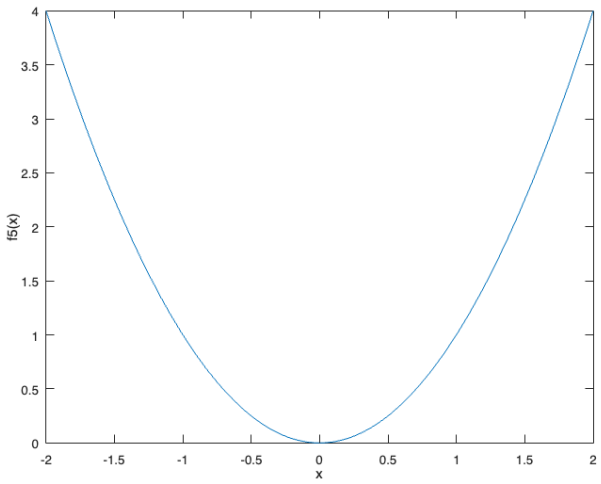
```
disp(result);
```

```
        2187
```

P3-24

Create an anonymous function, f5, that takes 3 inputs, *f*, *a* and *b*, where *f* is another anonymous function, and plots *f* between *a* and *b*. Test f5 by using any test function over whatever domain you like.

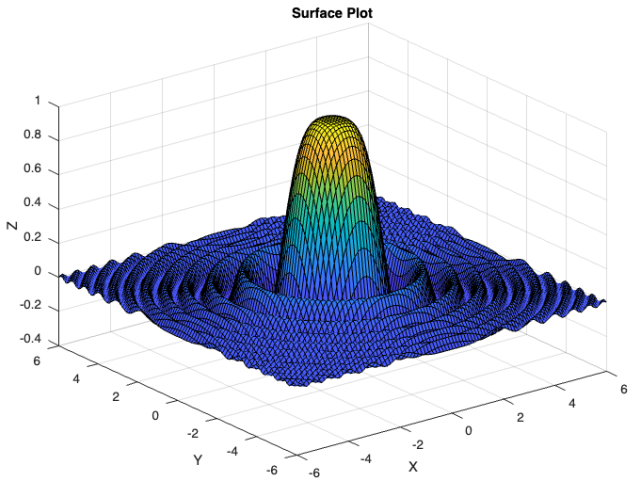
```
% Define the anonymous function f5
f5 = @(f, a, b) fplot(f, [a, b]);

figure;

% Test f5 with an example function f(x) = x^2 and the domain -2 to 2
f5(@(x) x.^2, -2, 2);

% Add labels and title to the plot
xlabel('x');
ylabel('f5(x)');
```



P3-25

Create a function file (not anonymous) that takes 3 inputs and produces a surface plot. The first input should be a function that accepts 2 arguments (x and y), the second and third inputs should be vectors containing the desired values of x and y over which to plot. Test your function using any function and domain. (example solution) I tested my function file with an input function sin(x^2+y^2)/(x^2+y^2) between -6 and 6 on both the x- and y-axes.

```
% Define the function to be plotted
func = @(x, y) sin(x.^2 + y.^2) ./ (x.^2 + y.^2);

% Define the range for x and y
x_values = linspace(-6, 6, 100); % Adjust the number of points as needed
y_values = linspace(-6, 6, 100); % Adjust the number of points as needed

% Call the plot_surface function to create the surface plot
plotSurface(func, x_values, y_values);
```



P3-26

Write a function file that takes between 3 and 6 input arguments (all scalar values) and returns a vector containing all the inputs sorted from lowest to highest. Warning messages should be displayed in the command window if the user enters the wrong number of inputs, or a vector/string instead of a scalar.

```
try
   result = customSort(1, 2);
   disp(result);
end
try
    result = customSort(1, 2, 3);
    disp(result);
end
    1    2    3
try
    result = customSort(1, 2, 3, 4);
    disp(result);
end
    1    2    3    4
try
   result = customSort(1, 2, 3, 4, 5);
   disp(result);
end
    1    2    3    4    5
try
    result = customSort(1, 2, 3, 4, 5, 6);
    disp(result);
end
    1    2    3    4    5    6
try
    result = customSort(1, 2, 3, 4, 5, 6, 7);
    disp(result);
end
try
    result = customSort(1, 2, 3, 4, 'test');
    disp(result);
end
```

P3-27

Challenge Problem

Write a function file that requires 3 inputs, *F, a* and *b*, where *F* is a function of one variable, and plots the function between *a* and *b*, but also accepts additional optional input arguments that specify other intervals to plot the function over.

The plots should appear in the same figure (subplots) with the pattern as shown below:

Note that the additional optional arguments should come in pairs (*c* and *d*, *e* and *f etc.*). Display warning messages if the user enters the wrong number or type of input. Finally accept up to 100 subplots.

Hint: To determine if the first argument is a function read about **isa()** in the documentation.

Hint: To determine the layout of the subplots it will be helpful to think about the relationship between square numbers, number of plots requested (n), and the number of rows and columns.

Write a function file that requires 3 inputs, *F, a* and *b*, where *F* is a function of one variable, and plots the function between *a* and *b*, but also accepts additional optional input arguments that specify other intervals to plot the function over. The plots should appear in the same figure (subplots). Note that the additional optional arguments should come in pairs (*c* and *d*, *e* and *f etc.*). Display warning messages if the user enters the wrong number or type of input. Finally accept up to 100 subplots.

Hint: To determine if the first argument is a function read about isa() in the documentation.

Hint: To make the layout of the plots dynamic (responds to how many subplots you request), try organising them into the following pattern:

```
F = @(x) sin(x);

% Example 1: Plot 1 interval function
plotMultipleFunctions(F, 0, pi);
```

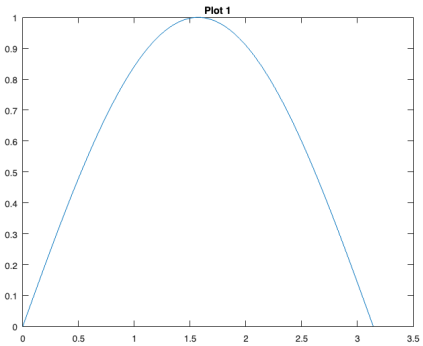
```
No. of plots requested: 1
```

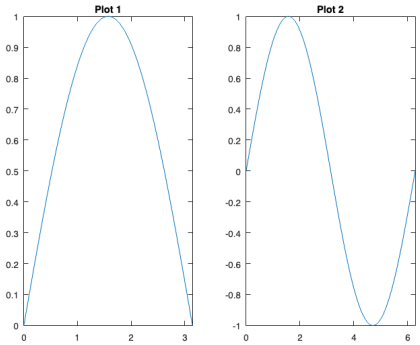
```
% Example 2: Plot 2 intervals functions
plotMultipleFunctions(F, 0, pi, 0, 2*pi);
```

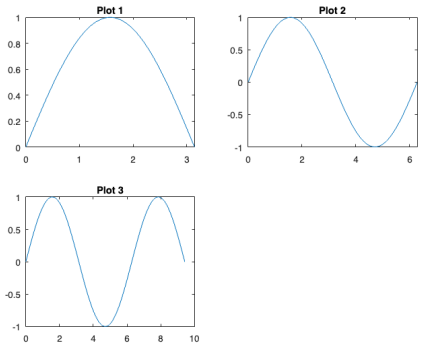
```
No. of plots requested: 2
```

```
% Example 3: Plot 3 intervals functions
plotMultipleFunctions(F, 0, pi, 0, 2*pi, 0, 3*pi);
```

```
No. of plots requested: 3
```







```
% Example 4: Plot 5 intervals functions
plotMultipleFunctions(F, 0, pi, 0, 2*pi, 0, 3*pi, 0, 4*pi, 0, 5*pi);
```

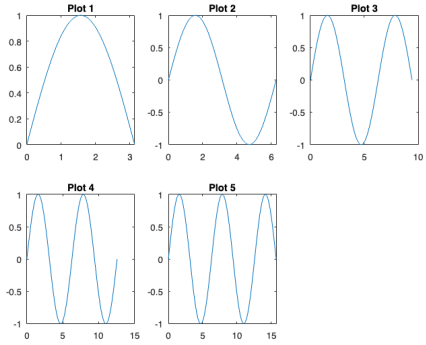
No. of plots requested: 5

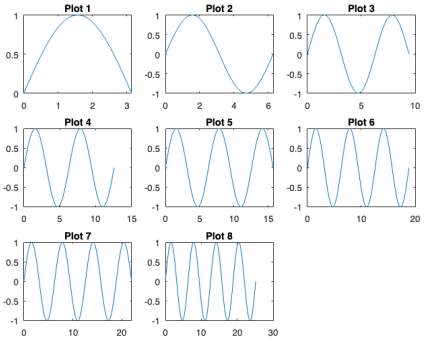
```
% Example 5: Plot 8 intervals functions
plotMultipleFunctions(F, 0, pi, 0, 2*pi, 0, 3*pi, 0, 4*pi, 0, 5*pi, 0, 6*pi, 0, 7*pi, 0, 8*pi);
```

No. of plots requested: 8

```
% Example 6: Plot 96 intervals functions
intervals = cell(1, 96);
for i = 1:96
    intervals{2 * i - 1} = (i - 1) * pi;
    intervals{2 * i} = i * pi;
end

plotMultipleFunctions(F, intervals{:});
```

No. of plots requested: 96





Testing that input errors are handled:

- Too many inputs

```
intervals = cell(1, 101);


for i = 1:101
    intervals{2 * i - 1} = (i - 1) * pi;
    intervals{2 * i} = i * pi;
end


%plotMultipleFunctions(F, intervals{:});
```

- Odd number of interval inputs

```
%plotMultipleFunctions(F, 0, pi, 0);
```

- First argument not a function

```
%plotMultipleFunctions('test', 0, pi);
```

- An interval argument is not a number

```
%plotMultipleFunctions(F, 0, 'test');
```

Problems Week 4

P4-1

(Overflow Error) Matlab returns **Inf** when an overflow error in double precision is encountered. Write a script that starts with $n = 1$, calculates $n!$ and increases n until overflow occurs (number is too big that Matlab can't handle it). Display the value of n that represents the largest factorial that Matlab can handle.

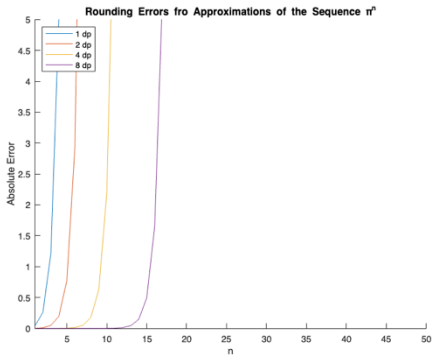
```
n = 1;
maxValue = realmax('double');

while true
    try
        factorial_n = factorial(n);
        if factorial_n > maxValue
            error('Overflow error');
        end
        n = n + 1;
    catch
        break;
    end
end

fprintf('Highest factorial is: %d\n', n - 1);
```

```
Highest factorial is: 170
```

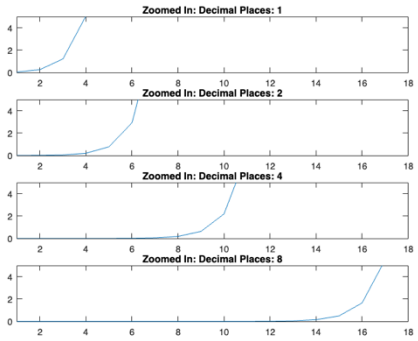
P4-2

[Rounding Error] Approximate the sequence $\{\pi^n\}$ for n = 1 to 18 by rounding π to 1, 2, 4, and 8 decimal places. Calculate the absolute error for each sequence approximation from the sequence that uses π with double precision. Plot the errors for each approximate sequence on the same plot with y-axis between 0 and 5. See if you can figure out how to embed smaller plots within the main plot that zoom in on each of the sequences.

```
% Define the values of n
n = 1:18;
% Calculate the exact sequence using double precision
exact_sequence = pi.^n;
% Define the decimal places for rounding π
decimal_places = [1, 2, 4, 8];

% Initialize arrays to store the errors for each approximation
errors = zeros(length(decimal_places), length(n));

% Loop through each decimal place
for i = 1:length(decimal_places)
    % Round π to the specified decimal place
    rounded_pi = round(pi, decimal_places(i));

    % Calculate the approximate sequence
    approx_sequence = rounded_pi.^n;

    % Calculate the absolute error for each term in the sequence
    errors(i, :) = abs(approx_sequence - exact_sequence);
end
% Create a main plot
figure;
hold on;
title('Rounding Errors fro Approximations of the Sequence π^n');
xlabel('n');
ylabel('Absolute Error');
ylim([0, 5])
;xlim([1, 50]);

% Plot the errors for each decimal place
for i = 1:length(decimal_places)
    plot(n, errors(i, :), 'DisplayName', [num2str(decimal_places(i)) ' dp']);
end

% Add a legend to the main plot
legend('Location', 'Northwest');
hold off;
```



```
% Create embedded plots for zooming in on errors
figure;
for i = 1:length(decimal_places)
    subplot(length(decimal_places), 1, i);
    plot(n, errors(i, :));
    title(['Zoomed In: Decimal Places: ' num2str(decimal_places(i))]);
    xlim([1, 18]);
    ylim([0, 5]);
end
```



P4-3

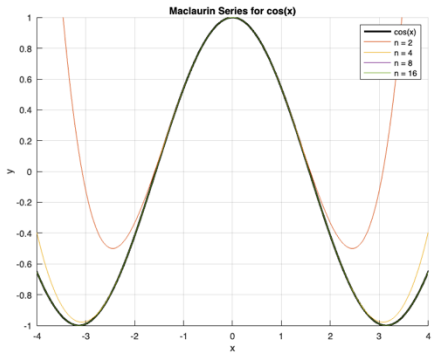
[Truncation Error] The Maclaurin series for cos(x) is

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1^n)^n x^{2n}}{(2n)!}$$

Plot the cosine function in a thick black line between -2π and 2π. On the same graph plot the Maclaurin series approximation for n equal to 2, 4, 8 and 16.

```
% Define the range for x values
x = linspace(-2*pi, 2*pi, 1000);

% Compute the true cosine values
cos_x = cos(x);

% Initialize variables to store Maclaurin series approximations
n_values = [2, 4, 8, 16];
approximations = zeros(length(n_values), length(x));

% Compute Maclaurin series approximations for different n values
for i = 1:length(n_values)
    n = n_values(i);
    approximation = 0;
    for k = 0:n
        approximation = approximation + ((-1)^k * x.^(2*k)) / factorial(2*k);
    end
    approximations(i, :) = approximation;
end

% Create the plot
figure;
hold on;

% Plot the true cosine function in a thick black line and add a legend entry
plot(x, cos_x, 'k', 'LineWidth', 2);
legend('cos(x)');

% Plot Maclaurin series approximations for n values and add legend entries
for i = 1:length(n_values)
    plot(x, approximations(i, :));
end
legend('cos(x)', 'n = 2', 'n = 4', 'n = 8', 'n = 16');
```

```
% Add labels and title
xlabel('x');
ylabel('y');
title('Maclaurin Series for cos(x)');

% Set axis limits
xlim([-4, 4]);
ylim([-1, 1]);

% Turn on the grid
grid on;

% Hold off to end the plotting
hold off;
```



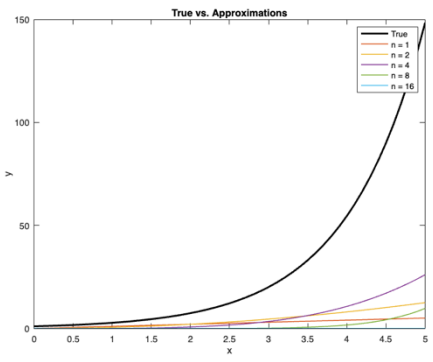
P4-5

[True vs. Relative Error] The Maclaurin series for e^x is

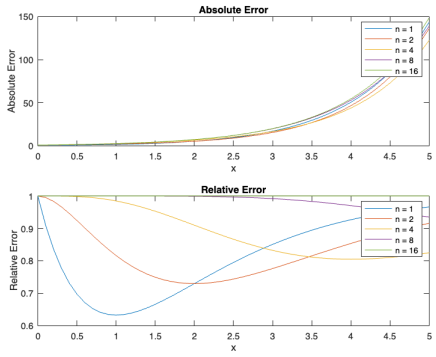
$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Plot the true function from 0 to 5 then on the same graph plot the Maclaurin approximations for *n* equal to 1, 2, 4, 8, and 16. Calculate the true absolute and true relative errors then plot these together in a separate graph.

```
% Define the range of x values
x = 0:0.1:5;
% Calculate true values of e^x
true_values = exp(x);

% Initialize variables to store Maclaurin approximations
n_values = [1, 2, 4, 8, 16];
maclaurin_values = zeros(length(n_values), length(x));

% Calculate Maclaurin approximations for different n values
for i = 1:length(n_values)
    n = n_values(i);
    maclaurin_values(i, :) = sum((x.^n) / factorial(n), 1);
end

% Calculate true absolute and relative errors
true_absolute_errors = abs(true_values - maclaurin_values);
true_relative_errors = true_absolute_errors ./ true_values;

% Create a plot for the true function and Maclaurin approximations
figure;
plot(x, true_values, 'k-', 'LineWidth', 2); % True function
hold on;

for i = 1:length(n_values)
    plot(x, maclaurin_values(i, :), 'LineWidth', 1);
end

legend('True', 'n = 1', 'n = 2', 'n = 4', 'n = 8', 'n = 16');
title('True vs. Approximations');
xlabel('x');
ylabel('y');
hold off;
```



```
figure;

subplot(2, 1, 1);
plot(x, true_absolute_errors, 'LineWidth', 0.5);
legend('n = 1', 'n = 2', 'n = 4', 'n = 8', 'n = 16');
title('Absolute Error');
xlabel('x');
ylabel('Absolute Error');

subplot(2, 1, 2);
plot(x, true_relative_errors, 'LineWidth', 0.5);
legend('n = 1', 'n = 2', 'n = 4', 'n = 8', 'n = 16');
title('Relative Error');
xlabel('x');
ylabel('Relative Error');
```



P4-6

(Convergence with Absolute Error Criterion) The series

$$\sum_{n=0}^{\infty} \frac{4^{n+1}}{(3n+1)!}$$

is convergent. Write a script that finds the value the series converges to by using a loop and the absolute approximate error with a tolerance of 0.0001 (stopping criterion).

```
% Initialize variables
n = 0; % Initialize n
sum_series = 0; % Initialize the sum of the series
tolerance = 0.0001; % Tolerance for absolute error
absolute_error = 1; % Initialize the absolute error
term = 1; % Initialize the current term

% Loop until the absolute error is less than the tolerance
while absolute_error >= tolerance
    % Calculate the current term of the series
    term = (4^(n+1)) / factorial(3*n+1);

    % Add the current term to the sum
    sum_series = sum_series + term;

    % Calculate the absolute error
    absolute_error = abs(term);

    % Increment n for the next iteration
    n = n + 1;

    % Display the current absolute error
    fprintf('Absolute error is: %.5f\n', absolute_error);
end
```

```
Absolute error is: 4.00000
Absolute error is: 0.66667
Absolute error is: 0.01270
Absolute error is: 0.00007
```

```
n = n - 1;

% Display the final result
fprintf('The sum has converged to %.4f\n', sum_series);
```

```
The sum has converged to 4.6794
```

```
fprintf('The number of terms required is %d\n', n);
```

```
The number of terms required is 3
```

P4-7

(Convergence with Relative Error Criterion) Write a script that finds the value that the series in question 6 converges to, by using a loop and the relative approximate error with a tolerance of 0.0001 (stopping criterion). Compare the results of this question with question 6.

```
% Initialize variables
n = 0; % Initialize n
sum_series = 0; % Initialize the sum of the series
tolerance = 0.0001; % Tolerance for relative error
relative_error = 1; % Initialize the relative error
term = 1; % Initialize the current term

% Loop until the relative error is less than the tolerance
while relative_error >= tolerance
    % Calculate the current term of the series
    term = (4^(n+1)) / factorial(3*n+1);

    % Add the current term to the sum
    sum_series = sum_series + term;

    % Calculate the relative error
    if sum_series ~= 0
        relative_error = abs(term / sum_series);
    else
        relative_error = abs(term);
    end

    % Increment n for the next iteration
    n = n + 1;

    % Display the current relative error
    fprintf('Relative error is: %.5f\n', relative_error);
end
```

```
Relative error is: 1.00000
Relative error is: 0.14286
Relative error is: 0.00271
Relative error is: 0.00002
```

```
n = n - 1;

% Display the final result
fprintf('The sum has converged to %.4f\n', sum_series);
```

```
The sum has converged to 4.6794
```

```
fprintf('The number of terms required is %d\n', n);
```

```
The number of terms required is 3
```

P4-8

(Open Method Root Finding) Write a script that implements the Incremental Search Method, with a step size of 0.1, to approximate the 3 roots to the following equation:

$$y = x^3 - 6x^2 + 10x - 4$$

Choose a sensible initial guess for your algorithm (plotting the function might help you decide).
Find a way to test how closely you approximated the roots.
Can you see why this method is only good for initial bracketing?

```
% Define the equation
f = @(x) x.^3 - 6*x.^2 + 10*x - 4;

% Define step size
step_size = 0.1;

% Define the range for searching
x_range = -5:step_size:5;

% Initialize variables to store roots and function values
roots = [];
values_at_roots = [];

% Loop through the range to find potential roots
for i = 1:length(x_range)-1
    x1 = x_range(i);
    x2 = x_range(i+1);

    % Check if the signs of the function values change
    if f(x1) * f(x2) <= 0
        % Use linear interpolation to approximate root
        root = x1 - f(x1) * (x2 - x1) / (f(x2) - f(x1));
        roots = [roots, root];
        values_at_roots = [values_at_roots, f(root)];
    end
end

% Round the root approximations to two decimal places
rounded_roots = round(roots, 2);
rounded_values_at_roots = round(values_at_roots, 4);
% Remove duplicate roots and their corresponding values
[rounded_roots, unique_indices] = unique(rounded_roots);
rounded_values_at_roots = rounded_values_at_roots(unique_indices);
```

```
% Display the rounded root approximations
disp('Root approximations:');
```

```
Root approximations:
```

```
disp(rounded_roots);
```

```
    0.5900    2.0000    3.4100
```

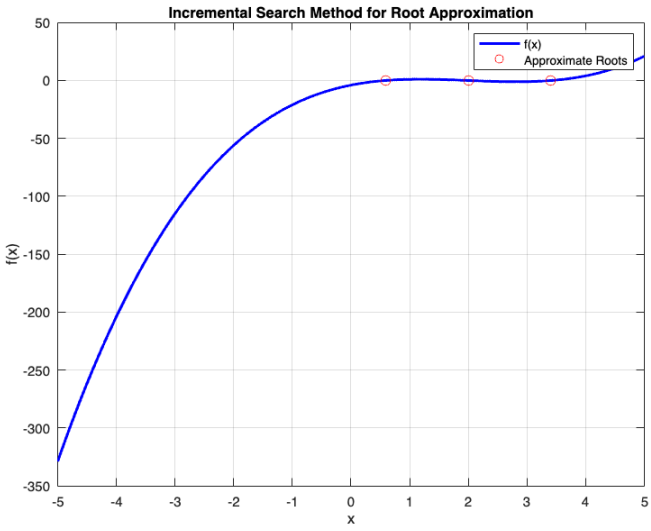
```
% Display the rounded function values at approximate roots
disp('Functions values at approximate roots:');
```

```
Functions values at approximate roots:
```

```
disp(rounded_values_at_roots);
```

```
    0.0049         0   -0.0049
```

```
% Create a plot of the function
figure;
x_values = -5:0.01:5; % Higher resolution for smoother curve
y_values = f(x_values);
plot(x_values, y_values, 'b', 'LineWidth', 2);
hold on;

% Mark the approximate roots on the plot
scatter(rounded_roots, rounded_values_at_roots, 50, 'r');
grid on;

% Labels and title
xlim([-5, 5]);
xlabel('x');
ylabel('f(x)');
title('Incremental Search Method for Root Approximation');
legend('f(x)', 'Approximate Roots');
```



P4-9

(Preparing Open Method for Use With Closed Method) Rewrite your script from Q8 to output 3 intervals where the roots are located between.

```
% Define the equation
f = @(x) x.^3 - 6*x.^2 + 10*x - 4;

% Define step size
step_size = 0.1;

% Define the range for searching
x_range = -10:step_size:10;

% Initialize variables to store intervals
intervals = [];

% Loop through the range to find intervals containing roots
for i = 1:length(x_range)-1
    x1 = x_range(i);
    x2 = x_range(i+1);

    % Check if the signs of the function values change
    if f(x1) * f(x2) <= 0
        % Store the interval
        intervals = [intervals; x1, x2];
    end
end

% Display the intervals containing roots
disp('The intervals are:');
```

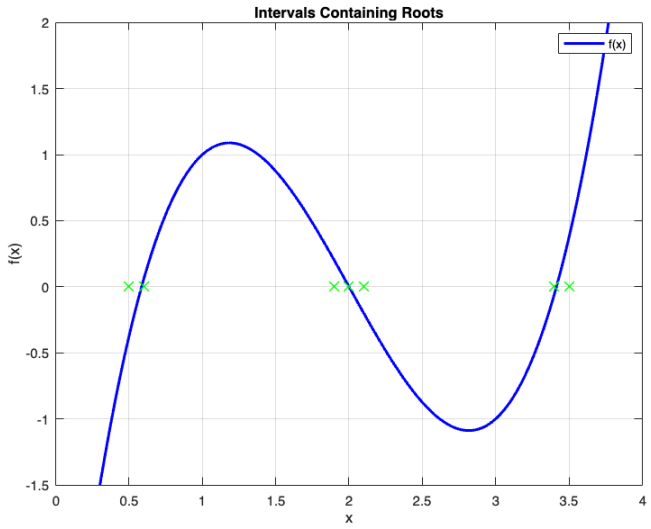
```
The intervals are:
```

```
disp(intervals);
```

```
    0.5000    0.6000
    1.9000    2.0000
    2.0000    2.1000
    3.4000    3.5000
```

```
% Create a plot of the function
figure;
x_values = -10:0.01:10; % Higher resolution for smoother curve
y_values = f(x_values);
plot(x_values, y_values, 'b', 'LineWidth', 2);
hold on;
```

```
% Mark the intervals containing roots on the plot
for i = 1:size(intervals, 1)
    x_interval = intervals(i,:);
    plot(x_interval, 0, 'gx', 'MarkerSize', 10, 'LineWidth', 1);
end

grid on;

% Labels and title
xlim([0, 4]);
ylim([-1.5, 2]);
xlabel('x');
ylabel('f(x)');
title('Intervals Containing Roots');
legend('f(x)');
```



P4-10

(Generalising Method to Any Function) Write a Matlab function file that implements the Incremental Search Method. The user should be able to input a function f(x), step size h, and range of values to search between (x0,xend). The output should be a list of brackets (intervals) containing each root.
Test your function with the function

$$f(x) = \sin(10x) + \cos(3x)$$

on the interval [3, 6] with step sizes of 0.06 and 0.03.

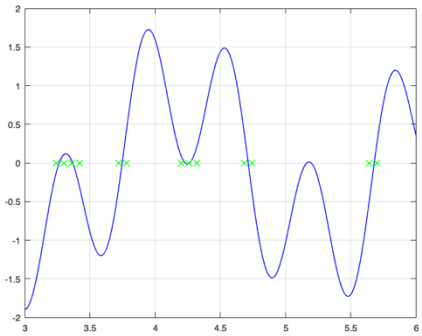
```
% Define the function
f = @(x) sin(10*x) + cos(3*x);
% Define the interval and step sizes
x0 = 3;
xend = 6;
step_size1 = 0.06;
step_size2 = 0.03;
% Find brackets using the Incremental Search Method
brackets1 = incrementalSearch(f, x0, xend, step_size1);
brackets2 = incrementalSearch(f, x0, xend, step_size2);
% Display the brackets
disp('Brackets with step size 0.06:');
```

```
Brackets with step size 0.06:
```

```
disp(brackets1);
```

```
    3.2400    3.3000
    3.3600    3.4200
    3.7200    3.7800
    4.2000    4.2600
    4.2600    4.3200
    4.6800    4.7400
    5.6400    5.7000
```

```
figure;
x_values = 0:0.01:10;
y_values = f(x_values);
plot(x_values, y_values, 'b', 'LineWidth', 1);
hold on;

for i = 1:size(brackets1, 1)
    x_interval = brackets1(i,:);
    plot(x_interval, 0, 'gx', 'MarkerSize', 10, 'LineWidth', 1);
end

grid on;
xlim([3, 6]);
ylim([-2, 2]);
```



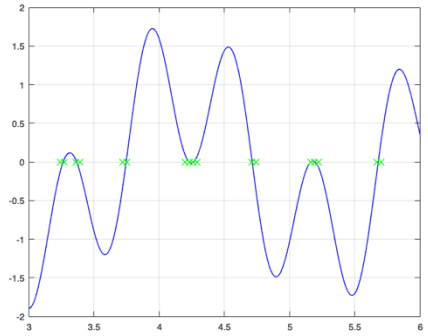
```
disp('Brackets with step size 0.03:');
```

```
Brackets with step size 0.03:
```

```
disp(brackets2);
```

```
    3.2400    3.2700
    3.3600    3.3900
    3.7200    3.7500
    4.2000    4.2300
    4.2600    4.2900
    4.7100    4.7400
    5.1600    5.1900
    5.1900    5.2200
    5.6700    5.7000
```

```
figure;
x_values = 0:0.01:10;
y_values = f(x_values);
plot(x_values, y_values, 'b', 'LineWidth', 1);
hold on;

for i = 1:size(brackets2, 1)
    x_interval = brackets2(i,:);
    plot(x_interval, 0, 'gx', 'MarkerSize', 10, 'LineWidth', 1);
end

grid on;

xlim([3, 6]);
ylim([-2, 2]);
```




```
function brackets = incrementalSearch(f, x0, xend, h)
    brackets = [];
    x = x0;
    while x <= xend - h
        x1 = x;
        x2 = x + h;
        if f(x1) * f(x2) < 0
            brackets = [brackets; [x1, x2]];
        end
        x = x2;
    end
end
```

Problems Week 5

Plotting the function is a good idea to get an idea of where the roots are and provide initial guesses, then the methods can be used to get an accurate estimation.
You can also substitute your answers back into the equation to see if it is correct (anonymous functions can help with that).

P5-1

Use the roots function to find the roots of the polynomial.

$$p(x) = x^5 - 10x^3 + 21x$$

```
% Define the polynomial
p = @(x) x.^5 - 10*x.^3 + 21*x;

% Find the roots
roots_of_p = roots([1, 0, -10, 0, 21, 0]);
```

P5-2

Create an anonymous function for p(x) and substitute the roots you found in question 1 into it to verify your answer.

```
% Substitute the roots back into p(x)
results = arrayfun(p, roots_of_p);

% Display the results
disp('The roots are:');
```

```
The roots are:
```

```
disp(roots_of_p);
```

```
         0
   -2.6458
   -1.7321
    2.6458
    1.7321
```

```
disp('Substituting the roots back into the function gives:');
```

```
Substituting the roots back into the function gives:
```

```
disp(results);
```

```
   1.0e-12 *

         0
   -0.1705
   -0.0568
   -0.0995
    0.0142
```

P5-3

Use the roots function to find the roots of the polynomial (no template). Display only the real roots.

$$q(x) = -2x^6 - 1.5x^4 + 10x + 2$$

```
% Define the polynomial q(x)
q = @(x) -2*x.^6 - 1.5*x.^4 + 10*x + 2;

% Define the coefficients of the polynomial q(x)
coefficients = [-2, 0, -1.5, 0, 0, 10, 2];

% Find the roots of the polynomial
polynomial_roots = roots(coefficients);

% Display only the real roots
real_roots = polynomial_roots(imag(polynomial_roots) == 0);

% Display the real roots and their evaluations
disp("Real roots are as follows.")
```

```
Real roots are as follows.
```

```
for i = 1:length(real_roots)
    disp("Root " + i + ":")
    disp(real_roots(i))

    % Evaluate the polynomial at the root
    evaluation = polyval(coefficients, real_roots(i));
    disp("Substituting back into the function gives:")
    disp(evaluation)
end
```

```
Root 1:
    1.3213
Substituting back into the function gives:
  -1.6431e-14
Root 2:
   -0.1997
Substituting back into the function gives:
  -4.4409e-16
```

P5-4

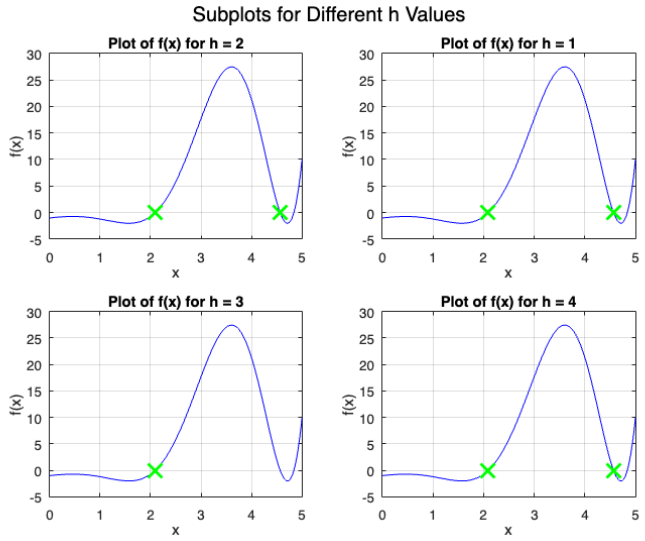
Use the fzero function to locate all roots of

$$f(x) = e^x \cos^2(x) - 2$$

on the interval [0,5]. Plot the function with a grid and mark the roots with large green diamonds. For this question make a function file to hold the function. You will have to divide the interval into pieces using an appropriate step size, h. First try
h = 2, then plot the function and roots to see if it found them all. If not, adjust your value of h and try again.

```
% Define the interval and different step sizes
a = 0;
b = 5;
hs = [2, 1, 3, 4];

% Initialize a cell array to store the roots for each h value
roots_all = cell(1, numel(hs));

% Initialize a cell array to store the x values for each h value
x_values_all = cell(1, numel(hs));

% Create subplots for each h value
for i = 1:numel(hs)
    h = hs(i);

    % Initialize an array to store the roots
    roots = [];

    % Loop through the interval with step size h
    for x = a:h:b
        % Use fzero to find a root in the current subinterval
        root = fzero(@myFunction, x);

        % Check if the root is within the current subinterval
        if root >= x && root <= x + h
            % Add the root to the array
            roots = [roots, root];
        end
    end

    % Store the roots and x values in cell arrays
    roots_all{i} = roots;
    x_values_all{i} = linspace(a, b, 1000);
```

```
    % Plot the function with a grid in the current subplot
    subplot(2, 2, i);
    y_values = myFunction(x_values_all{i});
    plot(x_values_all{i}, y_values, 'b-');
    grid on;
    xlabel('x');
    ylabel('f(x)');
    title(['Plot of f(x) for h = ', num2str(h)]);

    % Mark the roots with large green diamonds
    hold on;
    plot(roots_all{i}, myFunction(roots_all{i}), 'gx', 'MarkerSize', 15, 'MarkerFaceColor', 'g', 'LineWidth', 2);
    hold off;
end

% Adjust subplot layout
sgtitle('Subplots for Different h Values');
```



P5-5

Write a script that solves h(x)=x^2-x-e^-x

$$h(x) = x^2 - x - e^{-x}$$

using Fixed Point Iteration. Choose your own stopping criteria and keep track of the error at each iteration. Keep in mind that you can choose any of the following 4 formulas for your fixed point iteration:

$$g(x) = -e^{-x} + x^2$$

$$g(x) = \sqrt{e^{-x} + x}$$

$$g(x) = -\ln(x^2 - x)$$

$$g(x) = 1 + \frac{e^{-x}}{x}$$

```
% Define the function h(x) and its derivative h'(x)
h = @(x) x.^2 - x - exp(-x);
h_prime = @(x) 2*x - 1 + exp(-x);

% Initial guess and tolerance
x0 = 1;
tolerance = 0.0001;

% Initialize variables to store results
max_iterations = 100;
error_history = zeros(max_iterations, 1);
```

```
% Formula 1: g(x) = -exp(-x) + x^2
g1 = @(x) -exp(-x) + x.^2;

% Initialize variables for Formula 1
x = x0;
iterations1 = 1;
g1list = [];

% Perform Fixed Point Iteration for Formula 1
while iterations1 < max_iterations
    g1list(end+1) = x;
    x_new = g1(x);
    error = abs(x_new - x);

    % Check the stopping criteria (derivative larger than 1)
    if h_prime(x) >= 1
        disp('Formula 1:');
        disp('Derivative larger than 1, stopping.');
        disp(['Number of iterations: ', num2str(iterations1)]);
        break;
    end

    % Check if the error is below the tolerance
    if error <= tolerance
        disp('Formula 1:');
        disp(['Number of iterations: ', num2str(iterations1)]);
        disp(['x = ', g1list]);
        break;
    end

    x = x_new;
    iterations1 = iterations1 + 1;
end
```

```
Formula 1:
Derivative larger than 1, stopping.
Number of iterations: 1
```

```
% Formula 2: g(x) = sqrt(exp(-x) + x)
g2 = @(x) sqrt(exp(-x) + x);

% Initialize variables for Formula 2
x = x0;
iterations2 = 1;
g2list = [];

% Perform Fixed Point Iteration for Formula 2
while iterations2 < max_iterations
    g2list(end+1) = x;
    x_new = g2(x);
    error = abs(x_new - x);

    % Check if the error is below the tolerance
    if error <= tolerance
        disp('Formula 2:');
        disp(['Number of iterations: ', num2str(iterations2)]);
        disp(['x = ', num2str(g2list)]);
        break;
    end

    x = x_new;
    iterations2 = iterations2 + 1;
end
```

```
Formula 2:
Number of iterations: 7
x = 1      1.1696      1.2166      1.23      1.2338      1.2349      1.2352
```

```
% Formula 3: g(x) = -log(x^2 - x)
g3 = @(x) -log(x.^2 - x);

% Initialize variables for Formula 3
x = x0;
iterations3 = 1;
g3list = [];

% Perform Fixed Point Iteration for Formula 3
while iterations3 < max_iterations
    g3list(end+1) = x;
    x_new = g3(x);
    error = abs(x_new - x);

    % Check the stopping criteria (derivative larger than 1)
    if h_prime(x) >= 1
        disp('Formula 3:');
        disp('Derivative larger than 1, stopping.');
        break;
    end

    % Check if the error is below the tolerance
    if error <= tolerance
        disp('Formula 3:');
        disp(['Number of iterations: ', num2str(iterations3)]);
        disp(['x = ', num2str(g3list)]);
        break;
    end

    x = x_new;
    iterations3 = iterations3 + 1;
end
```

```
Formula 3:
Derivative larger than 1, stopping.
```

```
% Formula 4: g(x) = 1 + exp(-x)/x
g4 = @(x) 1 + exp(-x)./x;

% Initialize variables for Formula 4
x = x0;
iterations4 = 1;
g4list = [];

% Perform Fixed Point Iteration for Formula 4
while iterations4 < max_iterations
    g4list(end+1) = x;
    x_new = g4(x);
    error = abs(x_new - x);

    % Check if the error is below the tolerance
    if error <= tolerance
        disp('Formula 4:');
        disp(['Number of iterations: ', num2str(iterations4)]);
        disp(['x = ', num2str(g4list)]);
        break;
    end

    x = x_new;
    iterations4 = iterations4 + 1;
end
```

```
Formula 4:
Number of iterations: 11
x = 1     1.3679     1.1862     1.2575     1.2261     1.2393     1.2337     1.2361     1.235     1.2355     1.2353
```

P5-6

Write a function that takes an initial guess as input then locates a root of

$$r(x) = \sin(x)\cos(2x)$$

using the Newton-Raphson Method. Remember you will have to differentiate the function to get the formula. Choose your own convergence criterion.

```
% Example 1: x0 = 1
[x1, n1] = newtonRaphson(1, 1e-6, 100);
fprintf('Root = %.4f (found after %d iterations)\n', x1, n1);
```

```
Root = 0.7854 (found after 4 iterations)
```

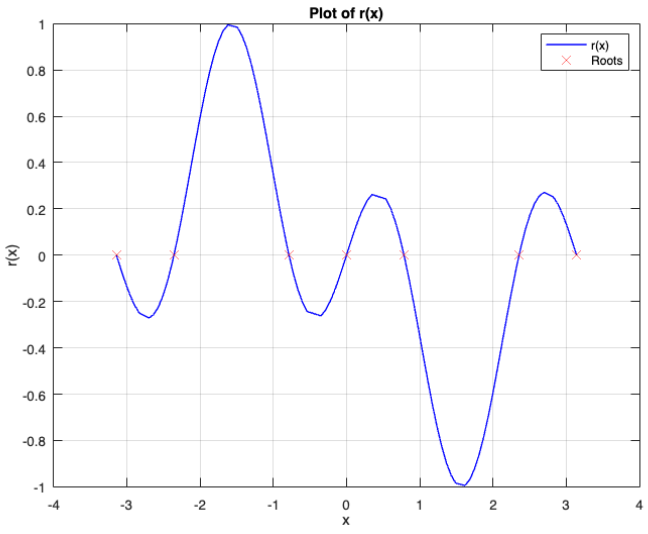
```
% Example 2: x0 = 5
[x2, n2] = newtonRaphson(5, 1e-6, 100);
fprintf('Root = %.4f (found after %d iterations)\n', x2, n2);
```

```
Root = 5.4978 (found after 5 iterations)
```

P5-7

Use your function from question 7 to find all roots of r(x) on the interval $[-\pi, \pi]$.

```
% Define the interval
interval = [-pi, pi];

% Initialize arrays to store roots and corresponding x-values
roots = [];
x_values = [];

% Define tolerance and maximum iterations
tolerance = 1e-6;
maxIterations = 100;

% Iterate through the interval and find roots
for x = linspace(interval(1), interval(2), 100)
    [root, ~] = newtonRaphson(x, tolerance, maxIterations);

    % Check if the root is within the specified interval
    if root >= interval(1) && root <= interval(2)
        roots = [roots, root];
        x_values = [x_values, x];
    end
end

% Define your function for plotting
r = @(x) sin(x) * cos(2 * x);

% Plot the function and roots
figure;
plot(x_values, arrayfun(r, x_values), 'b', 'LineWidth', 1);
title('Plot of r(x)');
xlabel('x');
ylabel('r(x)');

hold on;

% Plot the roots as red dots
plot(roots, arrayfun(r, roots), 'rx', 'MarkerSize', 10);
legend('r(x)', 'Roots');
grid on;
hold off;
```



P5-8

Write a script that locates all roots of

$$w(x) = x^3 - 6x^2 + 10x - 4$$

using the Bisection Method. Locate brackets by using the incremental search method you wrote before.

```
% Define the function w(x)
w = @(x) x^3 - 6*x^2 + 10*x - 4;
% Define the search interval [a, b]
a = -10; % Start with a wide interval
b = 10;
step = 0.01; % Incremental search step size
% Initialize variables
roots = []; % To store the roots
found_exact_root = false;

% Perform incremental search to locate brackets and check for exact roots
while a <= b
    if w(a) == 0
        roots = [roots, a];
        found_exact_root = true;
    elseif w(a) * w(a + step) < 0
        % Bracket found, apply Bisection method
        [root, ~] = bisection(w, a, a + step, 1e-8);
        roots = [roots, root];
    end

    a = a + step;
end

% Display whether an exact root was found
if found_exact_root
    disp('A root was found by the incremental search method.');
else
    disp('No exact root was found by the incremental search method.');
end
```

```
No exact root was found by the incremental search method.
```

```
% Display the roots found by the Bisection method
fprintf('The roots are:\n');
```

```
The roots are:
```

```
fprintf('root = %.4f ', roots);
```

```
root = 0.5858 root = 2.0000 root = 3.4142
```

P5-9-v.1

The volume of liquid V in a hollow horizontal cylinder of radius r and length L is related to the depth of the liquid h by,

$$v = \left[r^2 \cos^{-1}\left(\frac{r-h}{r}\right) - (r-h)\sqrt{2rh - h^2}\right] L$$

Determine h given r = 2m, L = 5m, and V = 8m.

```
% Define a function that represents the equation
fun = @(h) (2^2 * acos((2 - h) / 2) - (2 - h) * sqrt(2 * 2 * h - h^2)) * 5 - 8;

% Use fsolve to find the value of h
h = fsolve(fun, 1); % Start with an initial guess of h = 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>
```

```
% Display the result
fprintf('The depth of the liquid (h) is approximately %.2f meters\n', h);
```

```
The depth of the liquid (h) is approximately 0.74 meters
```

P5-9-v.2

```
% Given values
r = 2;    % radius in meters
L = 5;    % length in meters
V = 8;    % volume in cubic meters

% Define the equation in terms of h
equation = @(h) (r^2 * acos((r - h) / r) - (r - h) * sqrt(2 * r * h - h^2)) * L - V;

% Use numerical solver to find h
initial_guess = 1; % Initial guess for h
h = fzero(equation, initial_guess);

% Display the result
disp('h = ');
```

```
h =
```

```
disp(h);
```

```
    0.7400
```

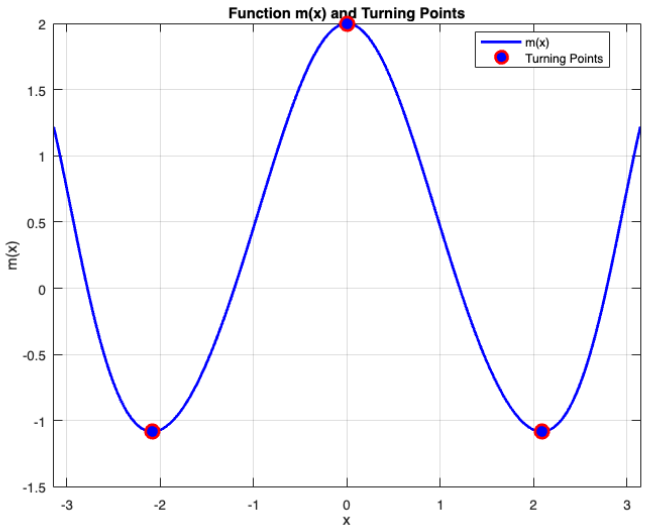
P5-10

Determine the location of the horizontal tangents (turning points) of

$$m(x) = \cos\left(\frac{x^2}{2}\right) + \cos(2x)$$

on the interval $[-\pi,\pi]$. Plot the figure and mark the turning points with thick, large red circles with blue faces

```
% Step 1: Create a range of x-values
x = linspace(-pi, pi, 1000);

% Define the function m(x)
m = cos(x.^2/2) + cos(2*x);

% Step 2: Calculate the derivative of m(x)
dm_dx = diff(m)./diff(x);

% Step 3: Find the x-values where the derivative is approximately zero
x_tangent = x(1:end-1) + diff(x)/2; % Use midpoints of intervals
x_turning_points = x_tangent(abs(dm_dx) < 0.01); % Tolerance can be adjusted

% Step 4: Calculate corresponding y-values
y_turning_points = cos(x_turning_points.^2/2) + cos(2*x_turning_points);

% Step 5: Plot the function and mark turning points
figure;
plot(x, m, 'b', 'LineWidth', 2);
hold on;
plot(x_turning_points, y_turning_points, 'ro', 'MarkerSize', 10, 'MarkerFaceColor', 'b', 'LineWidth', 2);
title('Function m(x) and Turning Points');
xlabel('x');
ylabel('m(x)');
grid on;
xlim([-pi, pi]);

% Display legend
legend({'m(x)', 'Turning Points'}, 'Location', 'Best');

% Show the plot
hold off;
```



```
function y = myFunction(x)
    y = exp(x) .* cos(x).^2 - 2;
end

function [root, n] = bisection(func, a, b, tolerance)
    n = 0;
    while (b - a) / 2 > tolerance
        midpoint = (a + b) / 2;
        if func(midpoint) == 0
            root = midpoint;
            return;
        elseif func(a) * func(midpoint) < 0
            b = midpoint;
        else
            a = midpoint;
        end
        n = n + 1;
    end
    root = (a + b) / 2;
end

function [root, n] = newtonRaphson(x0, tolerance, maxIterations)
    % Define the function and its derivative
    r = @(x) sin(x) * cos(2 * x);
    dr = @(x) cos(x) * cos(2 * x) - 2 * sin(x) * sin(2 * x);
    % Initialize variables
    x = x0;
    n = 0;
    % Iterate using the Newton-Raphson method
    while n < maxIterations
        n = n + 1;
        xNext = x - r(x) / dr(x);
        % Check for convergence
        if abs(xNext - x) < tolerance
            root = xNext;
            return;
        end
        x = xNext;
    end
    % If maxIterations reached, return the current estimate
    root = x;
end
```

Problems Week 6

P6-1

Use the **eye** function to create a 4 x 5 diagonal matrix called ***A*** with 7's on the leading diagonal.

```
% Create a 4x5 diagonal matrix A with 7s on the leading diagonal
A = 7 * eye(4, 5);
disp('A = ');
```

```
A =
```

```
disp(A);
```

```
     7     0     0     0     0
     0     7     0     0     0
     0     0     7     0     0
     0     0     0     7     0
```

P6-2

Create a 3 x 3 elementary matrix called ***E*** that adds 3 of the 1st row to the 3rd row of a matrix. Test it with a random 3 x 3 matrix of integers.

```
% Define the elementary matrix E
E = eye(3);  % Create a 3x3 identity matrix
E(3, 1) = 3; % Set the element in the 3rd row and 1st column to 3

% Create a random 3x3 matrix of integers
A = randi([-10, 10], 3, 3);

% Apply the elementary matrix E to A
B = E * A;

% Display the original matrix A and the result matrix B
disp('Original Matrix:');
```

```
Original Matrix:
```

```
disp(A);
```

```
   -10    -7    -9
    -2    -7     2
    -4    -2    -1
```

```
disp('After Row Operation:');
```

```
After Row Operation:
```

```
disp(B);
```

```
   -10    -7    -9
    -2    -7     2
   -34   -23   -28
```

P6-3

Augment the test matrix from question 2 with a 3 x 3 identity matrix then use Matlab's **rref** function to find it's reduced echelon form.

```
% Augment A with a 3x3 identity matrix
B = [A, eye(3)];

% Find the reduced row echelon form (RREF) using rref
RREF_B = rref(B);

% Display the augmented matrix B and its RREF
disp('Augmented Matrix:');
```

```
Augmented Matrix:
```

```
disp(B);
```

```
   -10    -7    -9     1     0     0
    -2    -7     2     0     1     0
    -4    -2    -1     0     0     1
```

```
disp('RREF:');
```

```
RREF:
```

```
disp(RREF_B);
```

```
    1.0000         0         0    0.0625    0.0625   -0.4375
         0    1.0000         0   -0.0568   -0.1477    0.2159
         0         0    1.0000   -0.1364    0.0455    0.3182
```

P6-4

Create the coefficient matrix for the following linear system, check that its determinant is not 0, then calculate the solution using the inverse matrix method (hint: you can find the inverse of a matrix with the inv function).

$$3x_1 + 4x_2 = 3$$

$$5x_1 + 6x_2 = 7$$

$$x = A^{-1}b$$

```
% Coefficient matrix A
A = [3 4; 5 6];

% Constant vector b
b = [3; 7];

% Step 1: Check if the determinant of A is nonzero
detA = det(A);
if detA == 0
    disp("The determinant of A is zero. The system may not have a unique solution.");
else
    % Step 2: Calculate the inverse of A
    A_inv = inv(A);

    % Step 3: Solve for x using the inverse matrix method
    x = A_inv * b;

    % Display the solution
    disp("The solution is:");
    for i = 1:length(x)
        fprintf("x%d = %.2f\n", i, x(i));
    end
end
```

The solution is:

x1 = 5.00

x2 = -3.00

P6-5

Solve the system above but using Matlab's left-division operator.

```
% Coefficient matrix A
A = [3 4; 5 6];

% Constant vector b
b = [3; 7];

% Solve for x using the left-division operator
x = A \ b;

% Display the solution
disp("The solution is:");
```

```
The solution is:
```

```
fprintf("x1 = %.2f\n", x(1));
```

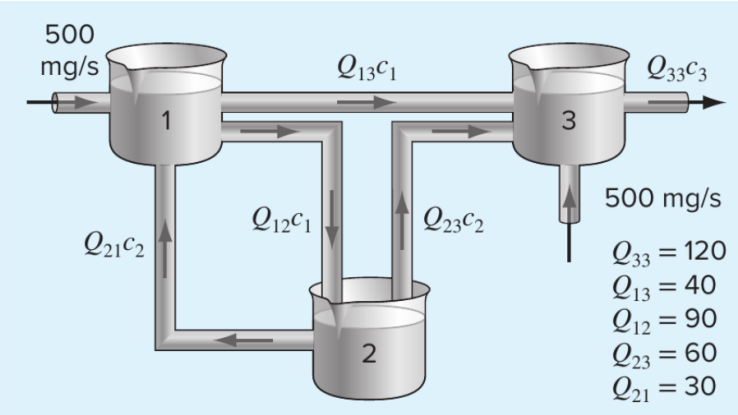
```
x1 = 5.00
```

```
fprintf("x2 = %.2f\n", x(2));
```

```
x2 = -3.00
```

P6-6

There are 3 reactors linked by pipes as shown below. The rate of transfer of chemicals through each pipe is equal to a flow rate (Q, with units of cubic meters per second) multiplied by the concentration of the reactor from which the flow originates (c, with units of milligrams per cubic meter). If the system is at a steady state, the transfer into each reactor will balance the transfer out. Develop mass-balance equations for the reactors and solve the three simultaneous linear algebraic equations for their concentrations.




```
% Given flow rates
Q33 = 120;
Q13 = 40;
Q12 = 90;
Q23 = 60;
Q21 = 30;

% Coefficients of the equations
A = [-Q13-Q12, Q21, -Q23; Q12, -Q21-Q23, 0; Q13, Q23, -Q33];
B = [-500; 0; 500];

% Solve the system of equations
C = A \ B;

% Extract concentrations
C1 = C(1);
C2 = C(2);
C3 = C(3);

% Display the results
fprintf('The value of C1 is %.4f mg/m^3\n', C1);
```

```
The value of C1 is 5.0000 mg/m^3
```

```
fprintf('The value of C2 is %.4f mg/m^3\n', C2);
```

```
The value of C2 is 5.0000 mg/m^3
```

```
fprintf('The value of C3 is %.4f mg/m^3\n', C3);
```

```
The value of C3 is -0.0000 mg/m^3
```

P6-7

Use Matlab's **fsolve** function to find the roots of the system. Since there are infinite solutions you do not need to find every root. Just locate and plot those given in the default window. Plotting is always a good idea to give you an idea of how many roots you are looking for and where to make the initial guesses. Use the **fimplicit** function to plot these implicit functions on the same graph. Define a function at the end of your script closed with keyword **end** that holds the system of equations. Advanced users can also use the ginput() function to pick starting initial guesses.

$$e^{xy} - y^2 = 0$$

$$\cos(x + y) = 0$$

```
% Define the system of equations
% Use fsolve to find the roots
initialGuesses = [0.5, 5; 0.5, 0.5; -0.5, 0.5; -0.5, -5; -0.5, -0.5; 2, -0.5];
roots = [];

for i = 1:size(initialGuesses, 1)
    root = fsolve(@equations, initialGuesses(i, :));
    roots = [roots; root];
end

% Display the roots
disp('The roots are:');
```

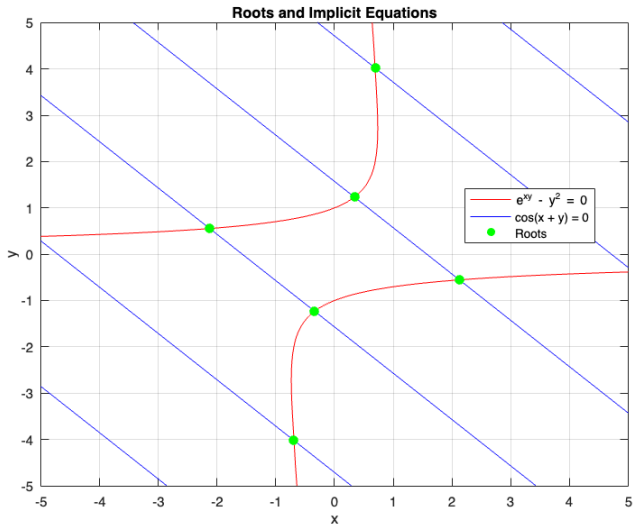
```
The roots are:
```

```
disp(roots);
```

```
    0.6922    4.0202
    0.3387    1.2320
   -2.1254    0.5546
   -0.6922   -4.0202
   -0.3387   -1.2320
    2.1254   -0.5546
```

```
% Plotting using fimplicit
figure;

% Define the implicit functions using function handles
f1 = @(x, y) exp(x .* y) - y.^2;
f2 = @(x, y) cos(x + y);

% Plot the implicit functions
fimplicit(f1, [-5 5 -5 5], 'r');
hold on;
fimplicit(f2, [-5 5 -5 5], 'b');
scatter(roots(:, 1), roots(:, 2), 50, 'go', 'filled');
```

```
title('Roots and Implicit Equations');
xlabel('x');
ylabel('y');
legend('e^{xy} - y^2 = 0', 'cos(x + y) = 0', 'Roots', 'Location', 'best');
grid on;
hold off;
```



P6-8

Find a root of the following system. Check your solution.

$$3x - \cos(yz) = \frac{3}{2} \quad 4x^2 - 625y^2 + 2z = 1 \quad 20z + e^{-xy} = -9$$

```
% Use fsolve to find a root
initialGuess = [0, 0, 0];
[x_sol, fval, exitflag] = fsolve(@myEquations, initialGuess);

%x_sol is the output variable where the solution to the system of equations is stored. x_sol contains the values of x, y, and z.
%fval is the output variable that contains the values of the equations evaluated at the solution x_sol.
%exitflag is the output variable that indicates the exit condition of the solver.
% A positive exitflag typically indicates a successful convergence to a solution, while a negative value may indicate issues or failure to find a solution.

x = x_sol(1);
y = x_sol(2);
z = x_sol(3);

% Check the solution
if exitflag > 0
    fprintf('Root found:\n');
    fprintf('x = %.6f\n', x);
    fprintf('y = %.6f\n', y);
    fprintf('z = %.6f\n', z);
else
    fprintf('No root found or an error occurred.\n');
end
```

```
Root found:
x = 0.833281
y = -0.035201
z = -0.501488
```

```
eq1 = 3 * x - cos(y * z) - 3/2;
eq2 = 4 * x^2 - 625 * y^2 + 2 * z - 1;
eq3 = 20 * z + exp(-x * y) + 9;
fprintf('Substituting into the first equation gives: %e\n', eq1);
```

```
Substituting into the first equation gives: 0.000000e+00
```

```
fprintf('Substituting into the second equation gives: %e\n', eq2);
```

```
Substituting into the second equation gives: -1.358913e-13
```

```
fprintf('Substituting into the third equation gives: %e\n', eq3);
```

```
Substituting into the third equation gives: 0.000000e+00
```

P6-9

Write a script that uses the Newton Method for Nonlinear Systems to solve the following nonlinear system. Try to find the roots of the system with your script. Remember you'll have to compute the Jacobian yourself by hand for now in order to put it into your script. 3x^2+2y^2=25 2x^2-y=15

$$3x^2 + 2y^2 = 25$$

$$2x^2 - y = 15$$

```
syms x y;
eq1 = 3*x^2 + 2*y^2 - 25;
eq2 = 2*x^2 - y - 15;
% Create the vector-valued function F
F = [eq1; eq2];
% Initialize the roots array
roots = [];
% Iterate over different initial guesses
initial_guesses = [1, 1; -1, 1; -1, -1; 1, -1];
for i = 1:size(initial_guesses, 1)
    x0 = initial_guesses(i, :)';
    % Tolerance for convergence
    tolerance = 1e-6;
    % Maximum number of iterations
    maxIterations = 100;
    % Initialize the Jacobian matrix
    Jacobian = jacobian(F, [x, y]);
    % Start the Newton-Raphson iteration
    for iter = 1:maxIterations
        % Calculate the value of F and Jacobian at the current point
        F_value = double(subs(F, [x; y], x0));
        J_value = double(subs(Jacobian, [x; y], x0));
        % Solve the linear system to get the update
        delta = -J_value \ F_value;
        % Update the current point
        x0 = x0 + delta;
        % Check for convergence
        if norm(delta) < tolerance
            roots = [roots; x0'];
            break;
        end
    end
end
```

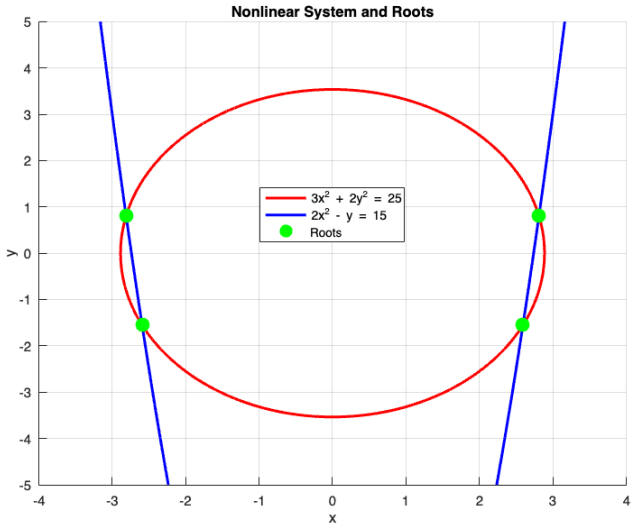
```
% Display the roots
fprintf('The roots are:\n');
```

```
The roots are:
```

```
disp(roots);
```

```
    2.8111    0.8042
   -2.8111    0.8042
   -2.5929   -1.5542
    2.5929   -1.5542
```

```
% Create a plot
figure;
hold on;

% Plot the equations
fimplicit(eq1, 'r', 'LineWidth', 2);
fimplicit(eq2, 'b', 'LineWidth', 2);

% Plot the roots
plot(roots(:, 1), roots(:, 2), 'go', 'MarkerSize', 10, 'MarkerFaceColor', 'g');

xlabel('x');
ylabel('y');
title('Nonlinear System and Roots');
legend('3x^2 + 2y^2 = 25', '2x^2 - y = 15', 'Roots', 'Location', 'Best');
grid on;

hold off;
```



P6-10

Write a script that reduces a 3 x 3 matrix to an echelon form (don't worry about pivoting). Test it on the following matrix:

$$A = \begin{bmatrix} 1 & -3 & 0 \\ 0 & 1 & 3 \\ 2 & -10 & 2 \end{bmatrix}$$

```
% Define the input matrix A
A = [1, -3, 0; 0, 1, 3; 2, -10, 2];

% Display the original matrix A
fprintf('Original Matrix A:\n');
```

```
Original Matrix A:
```

```
disp(A);
```

```
     1    -3     0
     0     1     3
     2   -10     2
```

```
% Subtract row 1 multiplied by 2 from row 3: R3 = R3 - 2 * R1
A(3, :) = A(3, :) - 2 * A(1, :);

% Add row 2 multiplied by 3 to row 1: R1 = R1 + 3 * R2
A(1, :) = A(1, :) + 3 * A(2, :);

% Add row 2 multiplied by 4 to row 3: R3 = R3 + 4 * R2
A(3, :) = A(3, :) + 4 * A(2, :);

% Display the final echelon form matrix
fprintf('Echelon Form Matrix A:\n');
```

```
Echelon Form Matrix A:
```

```
disp(A);
```

```
     1     0     9
     0     1     3
     0     0    14
```

P6-11

Write a script that reduces the above matrix to reduced echelon form (RREF).

```
% Compute the RREF of matrix A
RREF_A = rref(A);

% Display the RREF of matrix A
fprintf('Reduced Row Echelon Form (RREF) of Matrix A:\n');
```

```
Reduced Row Echelon Form (RREF) of Matrix A:
```

```
disp(RREF_A);
```

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

P6-12

Write a script that finds the LU factorisation of the matrix A in question 10 using Gaussian elimination to find U and the formulae given in lecture notes 6 to find L.

```
A = [1, -3, 0; 0, 1, 3; 2, -10, 2];
[n, ~] = size(A);

% Initialize L as an identity matrix
L = eye(n);

% Initialize U as a copy of A
U = A;

% Perform Gaussian elimination to find U and the elements of L
for k = 1:n-1
    for i = k+1:n
        % Calculate the multiplier for the elimination
        L(i, k) = U(i, k) / U(k, k);

        % Update the elements of U and L
        U(i, k:n) = U(i, k:n) - L(i, k) * U(k, k:n);
    end
end

% Display the matrices L and U
fprintf('Matrix L:\n');
```

```
Matrix L:
```

```
disp(L);
```

```
     1     0     0
     0     1     0
     2    -4     1
```

```
fprintf('Matrix U:\n');
```

```
Matrix U:
```

```
disp(U);
```

```
     1    -3     0
     0     1     3
     0     0    14
```

P6-13

Solve the above system using LU factorisation but use Matlab's built in **lu** function. Note you must account for the permutation matrix given by Matlab.

```
% Define the right-hand side vector
b = [-5; -1; -20];

% Solve Ly = b using forward substitution
n = length(b);
y = zeros(n, 1);

y(1) = b(1) / L(1, 1);
for i = 2:n
    y(i) = (b(i) - L(i, 1:i-1) * y(1:i-1)) / L(i, i);
end

% Solve Ux = y using backward substitution
x = zeros(n, 1);

x(n) = y(n) / U(n, n);
for i = n-1:-1:1
    x(i) = (y(i) - U(i, i+1:n) * x(i+1:n)) / U(i, i);
end

% Display the solution
fprintf('Solution:\n');
```

```
Solution:
```

```
disp(x);
```

```
     1
     2
    -1
```

P6-14

Solve the above system using LU factorisation but use Matlab's built in **lu** function. Note you must account for the permutation matrix given by Matlab.

```
A = [10 6 10 4 -7; -7 -8 3 5 4; 10 -2 -10 5 -10; 10 9 7 -2 -5; 0 6 9 3 -10];
disp("Matrix A");
```

```
Matrix A
```

```
A
```

```
A = 5×5
    10     6    10     4    -7
    -7    -8     3     5     4
    10    -2   -10     5   -10
    10     9     7    -2    -5
     0     6     9     3   -10
```

```
A_inverse = calculate_inverse(A);
disp("The invers is A^(-1)");
```

```
The invers is A^(-1)
```

```
A_inverse
```

```
A_inverse = 5×5
   -0.0637    0.1103    0.0509    0.1900   -0.0572
    0.6069   -0.5884   -0.1946   -0.8241   -0.0535
   -0.2148    0.2518    0.0393    0.3518    0.0359
    0.7049   -0.5136   -0.1601   -0.9043   -0.0866
    0.3822   -0.2805   -0.1295   -0.4491   -0.1257
```

```
check = A * A_inverse;
disp("Check A*A_inverse");
```

```
Check A*A_inverse
```

```
check
```

```
check = 5×5
    1.0000    0.0000   -0.0000         0   -0.0000
   -0.0000    1.0000    0.0000   -0.0000   -0.0000
   -0.0000    0.0000    1.0000    0.0000    0.0000
    0.0000    0.0000   -0.0000    1.0000         0
    0.0000   -0.0000   -0.0000   -0.0000    1.0000
```

```
function F = equations(x)
    F = [exp(x(1) * x(2)) - x(2)^2;
         cos(x(1) + x(2))];
end


function inverse_matrix = calculate_inverse(matrix)
    % Check if the input matrix is square
    [m, n] = size(matrix);
    if m ~= n
        error('Input matrix must be square.');
    end

    % Check if the determinant of the matrix is zero
    if abs(det(matrix)) < eps
        error('Matrix is not invertible (determinant is zero).');
    end

    % Calculate the inverse using MATLAB's built-in function
    inverse_matrix = inv(matrix);
end


function F = myEquations(X)
    x = X(1);
    y = X(2);
    z = X(3);


    F(1) = 3 * x - cos(y * z) - 3/2;
    F(2) = 4 * x^2 - 625 * y^2 + 2 * z - 1;
    F(3) = 20 * z + exp(-x * y) + 9;
end
```

Numerical Methods Midterm Exam 2022 – Solutions

a. Find and display all **real roots** of the following polynomial to 8 decimal places accuracy.

$$p(x) = 2x - 5x^2 - x^3 - 2x^7$$

```
% Enter solution here
% Define a list of coefficients for a polynomial in descending order of powers of x
coefficients = [-2, 0, 0, 0, -1, -5, 2, 0];

% Find the roots of the polynomial.
roots_of_poly = roots(coefficients);

% Extract the imaginary parts of the computed roots.
% Y = imag(Z) returns the imaginary part of each element in array Z.
imag(roots_of_poly)
```

```
ans = 7×1
         0
         0
    1.1348
   -1.1348
    0.7779
   -0.7779
         0
```

```
% Filter out the real roots by selecting roots with zero imaginary part.
real_roots = roots_of_poly(imag(roots_of_poly) == 0);

fprintf('Real Roots of the Polynomial:\n');
```

```
Real Roots of the Polynomial:
```

```
% Print the real roots with 8 decimal places accuracy.
fprintf('%.8f\n', real_roots);
```

```
0.00000000
-1.22056809
0.37136791
```

- Constant Polynomial (Degree 0): Equation: f(x) = 5 Real Root: There are no real roots for this constant polynomial.
- Linear Polynomial (Degree 1): Equation: f(x) = 2x + 3 Real Root: x = -3/2
- Quadratic Polynomial (Degree 2): Equation: f(x) = x^2 - 4x + 4 Real Roots: x = 2 (repeated root)
- Cubic Polynomial (Degree 3): Equation: f(x) = 3x^3 - 6x^2 + 3x Real Roots: x = 0 (repeated root), x = 2
- Quartic Polynomial (Degree 4): Equation: f(x) = 4x^4 - 12x^3 + 10x^2 - 2x + 1 Real Roots: x ≈ 0.5 (repeated root), x ≈ 1.0 (repeated root)
- Quintic Polynomial (Degree 5): Equation: f(x) = x^5 - 5x^4 + 10x^3 - 10x^2 + 5x - 1 Real Roots: Solving this quintic polynomial for exact real roots can be very complex and may require numerical methods or software.

```
linCoefficients = [2, 3];
roots_of_lin = roots(linCoefficients);
lin_real_roots = roots_of_lin(imag(roots_of_lin) == 0);
fprintf('%.8f\n', lin_real_roots);
```

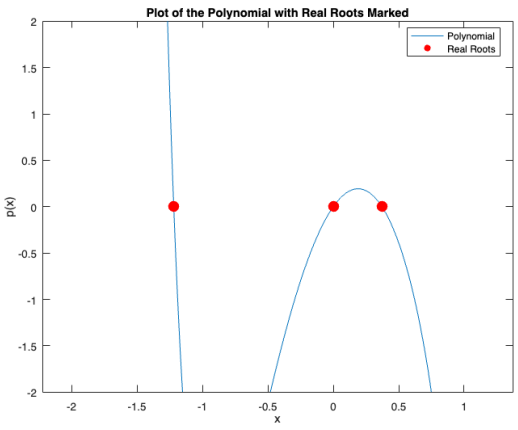
```
-1.50000000
```

```
quadCoefficients = [1, -4, 4];
roots_of_quad = roots(quadCoefficients);
quad_real_roots = roots_of_quad(imag(roots_of_quad) == 0);
fprintf('%.8f\n', quad_real_roots);
```

```
2.00000000
2.00000000
```

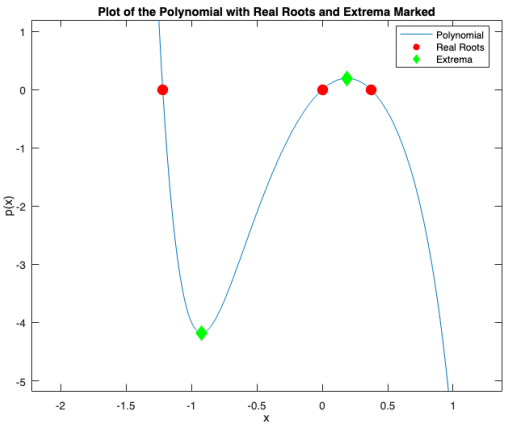
b. Plot the polynomial over an appropriate domain and mark on the real roots with large red circles.

```
% Enter solution here
% Define a range of x values for the plot.
x = linspace(min(real(real_roots)) - 1, max(real(real_roots)) + 1, 1000);

% Evaluate the polynomial at the x values.
y = polyval(coefficients, x);

figure;
plot(x, y);
hold on;

% Mark the real roots with large red circles.
% plot(real(real_roots), zeros(size(real_roots)), 'ro', 'MarkerSize', 10);

% Mark the real roots with red filled circles.
scatter(real(real_roots), zeros(size(real_roots)), 100, 'r', 'filled');
% real(real_roots) =  extracts the real parts of the computed real roots.
% เผื่อว่ามันมีค่าที่ไม่ใช่จำนวนจริงอยู่
% zeros(size(real_roots)) = creates a vector of zeros with the same size as the real_roots
% เพราะ y = 0 ทั้งหมด

xlabel('x');
ylabel('p(x)');
title('Plot of the Polynomial with Real Roots Marked');
legend('Polynomial', 'Real Roots', 'Location', 'best');

x1 = min(real(real_roots)) - 1;
x2 = max(real(real_roots)) + 1;
xlim([x1, x2]);
ylim([-2, 2]);
hold off;
```



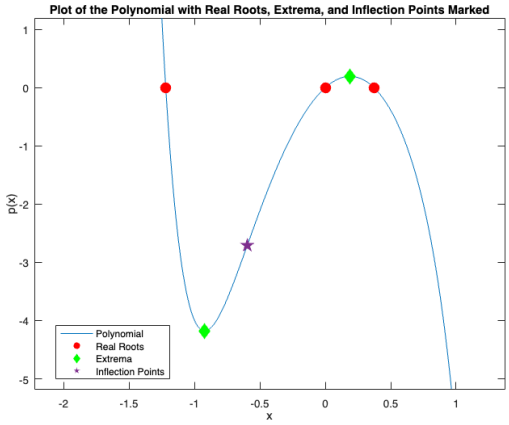
c. Find the locations of any local extrema of $p(x)$ and plot them with large green diamonds.

```
% Enter solution here
% Calculate the derivative of the polynomial
derivative_coefficients = polyder(coefficients);

% Find the roots of the derivative to locate the extrema
extrema_locations = roots(derivative_coefficients);

% Keep only real roots for extrema
real_extrema = extrema_locations(imag(extrema_locations) == 0);

% Calculate the corresponding y values at extrema locations
extrema_values = polyval(coefficients, real_extrema);

figure;
plot(x, y);
hold on;

scatter(real_roots, zeros(size(real_roots)), 100, 'ro', 'filled');  % Real roots as red circles
scatter(real_extrema, extrema_values, 100, 'gd', 'filled');  % Extrema as green diamonds

xlabel('x');
ylabel('p(x)');
title('Plot of the Polynomial with Real Roots and Extrema Marked');
legend('Polynomial', 'Real Roots', 'Extrema', 'Location', 'best');

y1 = min(extrema_values) - 1;
y2 = max(extrema_values) + 1;
xlim([x1, x2]);
ylim([y1, y2]);
hold off;
```



d. Find the locations of any points of inflections and plot them with large purple stars.

```
% Enter solution here
% Calculate the second derivative of the polynomial
second_derivative_coefficients = polyder(derivative_coefficients);

% Find the roots of the second derivative to locate the points of inflection
inflection_locations = roots(second_derivative_coefficients);

% Keep only real roots for inflection points
real_inflection = inflection_locations(imag(inflection_locations) == 0);

% Calculate the corresponding y values at inflection locations
inflection_values = polyval(coefficients, real_inflection);

figure;
plot(x, y);
hold on;

scatter(real_roots, zeros(size(real_roots)), 100, 'ro', 'filled');  % Real roots as red circles
scatter(real_extrema, extrema_values, 100, 'gd', 'filled');  % Extrema as green diamonds
scatter(real_inflection, inflection_values, 200, 'pentagram', 'filled');  % Inflection points as purple stars

xlabel('x');
ylabel('p(x)');
title('Plot of the Polynomial with Real Roots, Extrema, and Inflection Points Marked');
legend('Polynomial', 'Real Roots', 'Extrema', 'Inflection Points', 'Location', 'best');

xlim([x1, x2]);
ylim([y1, y2]);
hold off;
```



2. The speed v of a Saturn V rocket in vertical flight near the surface of earth can be approximated by:

$$v = u \ln \frac{M_0}{M_0 - \dot{m}t} - gt$$

where,

$$\begin{aligned} u &= 2510 \text{ m/s} = \text{velocity of exhaust relative to the rocket} \\ M_0 &= 2.8 \times 10^6 \text{ kg} = \text{mass of rocket at liftoff} \\ \dot{m} &= 13.3 \times 10^3 \text{ kg/s} = \text{rate of fuel consumption} \\ g &= 9.81 \text{ m/s}^2 = \text{gravitational acceleration} \\ t &= \text{time measured from liftoff} \end{aligned}$$

Determine the time when the rocket reaches the speed of sound (335 m/s).

```
% Enter solution here
u = 2510;              % m/s
M0 = 2.8e6;            % kg
m = 13.3e3;            % kg/s
g = 9.81;              % m/s^2
target_speed = 335;    % m/s

% Define the equation as a function of t
equation = @(t) u * log(M0 / (M0 - m * t)) - g * t - target_speed;

% Use the fzero function to find the root (i.e., the time t)
t = fzero(equation, 10);  % Starting with an initial guess of 10 seconds

% Display the result
fprintf('The time when the rocket reaches 335 m/s is %.2f seconds.\n', t);
```

The time when the rocket reaches 335 m/s is 70.88 seconds.

3. Find all real solutions to the following system of equations on the domain $-\pi/2 \le x \le \pi/2, \quad -5 \le x \le 5$ to 7 decimal places then create a plot which displays the solution.

$$\tan(2x - 1) = 1 + \cos(y)$$

$$\cos(x) = 2\sin^2(y)$$

```
% Enter solution here
```

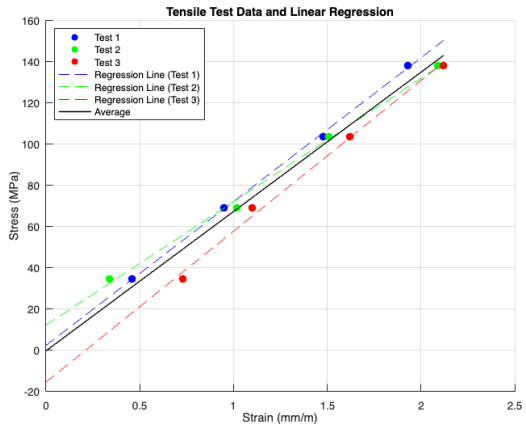
4. Three tensile tests were carried out on an aluminium bar. In each test the strain was measured at the same values of stress.
The results were:

Stress (MPa)	34.5	69.0	103.5	138.0
Strain (Test 1)	0.46	0.95	1.48	1.93
Strain (Test 2)	0.34	1.02	1.51	2.09
Strain (Test 3)	0.73	1.10	1.62	2.12

where the units of strain are mm/m ("millimetres per metre").
Plot the data then use linear regression to estimate the modulus of elasticity of the bar (modulus of elasticity = stress/strain). Plot the regression line on the same figure as the data.

```
% Enter solution here
% Data
stress = [34.5, 69.0, 103.5, 138.0]; % Stress in MPa
strain1 = [0.46, 0.95, 1.48, 1.93]; % Strain for Test 1 in mm/m
strain2 = [0.34, 1.02, 1.51, 2.09]; % Strain for Test 2 in mm/m
strain3 = [0.73, 1.10, 1.62, 2.12]; % Strain for Test 3 in mm/m

% Calculate modulus of elasticity for each test(modulus = stress/strain)
modulus1 = stress ./ strain1;
modulus2 = stress ./ strain2;
modulus3 = stress ./ strain3;

% Perform linear regression for each test, which means we are looking for a linear relationship (a straight line) between the variables strain.. and stress.
p1 = polyfit(strain1, stress, 1); % Linear regression for Test 1
p2 = polyfit(strain2, stress, 1); % Linear regression for Test 2
p3 = polyfit(strain3, stress, 1); % Linear regression for Test 3
% The independent variable (x) is usually the strain because it is the variable you control or measure independently.
% The dependent variable (y) is the stress because it depends on the strain and is what you are trying to predict or understand.

% Create strain values for regression lines
max_strain = max([max(strain1), max(strain2), max(strain3)]);
strain_fit = linspace(0, max_strain, 100);

% Calculate corresponding stress values for regression lines
stress_fit1 = polyval(p1, strain_fit); % Regression line for Test 1
stress_fit2 = polyval(p2, strain_fit); % Regression line for Test 2
stress_fit3 = polyval(p3, strain_fit); % Regression line for Test 3
```

```
% polyval function is used to calculate the predicted or fitted stress values (stress_fit1) based on the coefficients p1 obtained from the linear regression performed earlier with polyfit.

figure;
hold on;
scatter(strain1, stress, 50, 'bo', 'filled', 'DisplayName', 'Test 1');
scatter(strain2, stress, 50, 'go', 'filled', 'DisplayName', 'Test 2');
scatter(strain3, stress, 50, 'ro', 'filled', 'DisplayName', 'Test 3');
plot(strain_fit, stress_fit1, 'b--', 'DisplayName', 'Regression Line (Test 1)');
plot(strain_fit, stress_fit2, 'g--', 'DisplayName', 'Regression Line (Test 2)');
plot(strain_fit, stress_fit3, 'r--', 'DisplayName', 'Regression Line (Test 3)');

% Calculate the average stress and strain
avg_stress = (stress_fit1 + stress_fit2 + stress_fit3) / 3;

% Perform linear regression for the average data
avg_regression = polyfit(strain_fit, avg_stress, 1);

% Calculate corresponding stress values for the average regression line
avg_stress_fit = polyval(avg_regression, strain_fit);

% Plot the average stress-strain relationship
plot(strain_fit, avg_stress_fit, 'k-', 'LineWidth', 1, 'DisplayName', 'Average');

xlabel('Strain (mm/m)');
ylabel('Stress (MPa)');
title('Tensile Test Data and Linear Regression');
legend('Location', 'Northwest');

grid on;
hold off;
```



5. The following data is to be fit to a power equation of the form,

$$w = Ax^B y^C z^D,$$

where A, B, C and D are constants.

Use linearisation and regression to estimate the values of A. B. C and D then use the fitted power equation to estimate the value of w at the point $(x, y, z) = (3.5, 2.3, 7.1)$.

Data:

x	y	z	w
4	6	8	34.68
1	6	12	9.47
2	10	12	25.61
1	10	6	5.97
1	4	6	4.46
5	7	10	58.22
4	6	15	61.07
2	9	8	17.19
5	9	11	68.75
1	10	7	6.86

```
% Enter solution here
```

Linearization

To linearize the equation, take the natural logarithm of both sides:

$$\text{In}(w) = \text{In}(A) + \text{BIn}(x) + \text{CIn}(y) + \text{DIn}(z)$$

Now, our equation is in the form of a linear equation: $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3$, where :

- $y = \text{In}(w)$
- $x_1 = \text{In}(x)$
- $x_2 = \text{In}(y)$
- $x_3 = \text{In}(z)$
- $\beta_0 = \text{In}(A)$
- $\beta_1 = B$
- $\beta_2 = C$
- $\beta_3 = D$

```
% Given data
x = [4; 1; 2; 1; 1; 5; 4; 2; 5; 1];
y = [6; 6; 10; 10; 4; 7; 6; 9; 9; 10];
z = [8; 12; 12; 6; 6; 10; 15; 8; 11; 7];
w = [34.68; 9.47; 25.61; 5.97; 4.46; 58.22; 61.07; 17.19; 68.75; 6.86];
```

```
% Linearize the equation
ln_w = log(w);
ln_x = log(x);
ln_y = log(y);
ln_z = log(z);

% Create the design matrix X for linear regression
X = [ones(size(x)) ln_x ln_y ln_z];

% Perform linear regression to find coefficients of B, C, and D
coefficients = X \ ln_w;

% Extract coefficients for the power-law model: w = A * x^B * y^C * z^D
B = coefficients(2);
C = coefficients(3);
D = coefficients(4);

% Calculate the coefficient A by exponentiating the intercept
A = exp(coefficients(1));

% Display the estimated coefficients
fprintf('Estimated values:\n');
```

```
Estimated values:
```

```
fprintf('A = %.4f\n', A);
```

```
A = 0.5710
```

```
fprintf('B = %.4f\n', B);
```

```
B = 1.1998
```

```
fprintf('C = %.4f\n', C);
```

```
C = 0.3189
```

```
fprintf('D = %.4f\n', D);
```

```
D = 0.9003
```

Estimate w at (x, y, z) = (3.5, 2.3, 7.1)

Now that you have the estimated values of A, B, C, and D, you can calculate w at the point (x, y, z) = (3.5, 2.3, 7.1) using the fitted power equation:

```
% Define values for x, y, and z at a specific point
x_point = 3.5;
y_point = 2.3;
z_point = 7.1;
% Use the estimated coefficients to predict w at the specified point
estimated_w = A * x_point^B * y_point^C * z_point^D;
fprintf('Estimated w at (3.5, 2.3, 7.1) = %.4f\n', estimated_w);
```

```
Estimated w at (3.5, 2.3, 7.1) = 19.5505
```

6. Consider the following multivariable function:

$$f(x, y) = \frac{10 - y}{x - 3}$$

We wish to evaluate $f\left(\frac{1,000,000}{333,333}, \frac{5}{7}\right)$ in Matlab.

a. What is the true relative error when rounding the x and y values to 6 decimal places in the calculation?

```
% Enter solution here
% True value of f
x_true = 1,000,000/333,333;
```

```
x_true = 1
ans = 0
ans = 0
```

```
y_true = 5/7;
f_true = 10 - y_true / (x_true - 3);

% Rounded values of x and y
x_rounded = round(x_true, 6);
y_rounded = round(y_true, 6);

% Calculate f with rounded inputs
f_rounded = 10 - y_rounded / (x_rounded - 3);

% Calculate the true relative error
true_relative_error = abs(f_true - f_rounded) / abs(f_true);

true_relative_error
```

```
true_relative_error = 1.3793e-08
```

b. What is the true absolute error when using double precision in the calculation?

```
% Enter solution here
% Double precision values of x and y
x_double = double(1.000000/3.33333);
y_double = double(0.714286);
% Calculate f with double precision inputs
f_double_precision = 10 - y_double / (x_double - 3);
% Calculate the true absolute error
true_absolute_error = abs(f_true - f_double_precision);
true_absolute_error
```

```
true_absolute_error = 0.0926
```

c. To calculate the relative error in part (a) you had to divide a relatively small number by a relatively large number. Can you trust that value or not? Explain/demonstrate why the relative error you calculated can either be trusted or not.

```
% Enter solution here
```

In part (a), we calculated the true relative error by dividing the absolute difference between the true value f_true and the value calculated with rounded inputs f_rounded by the absolute value of f_true. This method is a valid way to calculate true_relative_error. However, in cases where we are dividing by a relatively small number, we need to be cautious about the relative error because it can potentially be sensitive to small changes in the denominator. This is especially true when the denominator approaches zero, as it can cause the relative error to become very large. In this specific case, the denominator involves the difference between x_true and 3, and x_true is a very large number. When we round x_true to 6 decimal places, you are effectively making it slightly smaller. This can lead to a situation where the denominator becomes very close to zero, which can result in a very large relative error.
To demonstrate this, consider the following MATLAB code:

```
% True value of f
x_true = 1.000000 / 3.33333;
y_true = 5 / 7;
f_true = 10 - y_true / (x_true - 3);
% Slightly perturb x_rounded to be closer to zero
x_perturbed = 1.000001 / 3.33333;
% Calculate f with the perturbed input
f_perturbed = 10 - y_true / (x_perturbed - 3);
% Calculate the true relative error
true_relative_error_perturbed = abs(f_true - f_perturbed) / abs(f_true);
true_relative_error_perturbed
```

```
true_relative_error_perturbed = 2.8637e-09
```

```
true_relative_error
```

```
true_relative_error = 1.3793e-08
```

We find that true_relative_error_perturbed is significantly larger than the original true_relative_error, indicating that the relative error can be sensitive to small changes in the denominator. So, in this case, while the relative error calculation is valid, we should be cautious when interpreting it because it can become very large due to the division by a relatively small number. This means that the relative error value should be considered with care and in the context of the specific problem we are trying to solve.