Project report 1: Scientific Computations With Python

Ryan Holloway (300570174)

Question 1

Introduction

The purpose of this problem is to experiment with various ways of computing solution using a vector. As the problem states, using for loops is a proper strategy to iterate through vectors (or list-like structures) in order to perform computations on each element (value) of the vector. Some helpful functions and operators for these problems are; += takes the left variable and adds the value to the right of the operator to it, ** OR pow(base, exponent) are used to raise a value to an exponential power, max() finds the maximum value in the list, min() finds the minimum value in the list, sum() sums all the values of the list, abs() finds the absolute value of a numerical value, range(start_inclusive, stop_exclusive) generates a range of numerical values between the start value and the stop value.

Procedure

Problem 1.a)

Computing the sum of a sequence of values can be accomplished by iterating through every element (5 elements in this instance) and cumulating the result of every element and storing it in the result (s1). When converting mathematical indexing to computational indexing, i (the iterator) must be converted from the natural mathematical index, starting at i = 1, to the value i = 0, since that is the value computers begin counting at; thus, it will affect the index of the elements in the vector. Additionally, the for loop uses the range function which is inclusive for its first parameter, and is exclusive for its second paramenter. This is why for in range(0, 5) will iterate through i = 0, i = 1, i = 2, i = 3, i = 4. Alternatively, instead of the user counting the elements in the vector, the function len() can be used to determine the length of the vector and can be use in-place of the second parameter, e.g. for i in range(0, len(x)). This is strategy will enable the user to iterate through a vector (or list) of an uknown length.

In [60]:
```
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
s1 = 0.0 # intialize s1 variable to float data type
for i in range(0, len(x)): #iterate through vector x (list)
    s1 += x[i] # cumulatively sum every element in vector x
print(s1)   # print solution
```

```
12.200000000000001
```

Problem 1.b)

This problem implements the same iteration process and summation as 1.a); however, every each element is raised to the power of its natural index. For this reason, starting the index at 0 for computational purposes will cause an error in the calculation of the solution, since n^0 is always equals 1, while n^1 = n. Additionally, the proceeding indices would be off-by-one in their exponential power. To resolve this issue of the list needing to iterate starting at 0, yet needing the value of the power to start at 1, we can simply use i+1 for the exponential power for every iteration. This will ensure the indexing still starts at 0, and every exponential power is accurate.

In [61]:
```
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
s2 = 0.0 # intialize s2 variable to float data type
for i in range(0, len(x)): #iterate through vector x (list)
    s2 += (x[i]**(i+1)) # cumulatively sum every element raised to the
                        # power of i+1 (natural index) in vector x
print(s2)   # print solution
```

```
281528.9834300001
```

Problem 1.c)

To find the maximum absolute of a vector, iteration through every element is required. For every iteration each current value (element) from the vector must be converted to it's absolute value first, then checked whether it's value is greater than all the previous elements' values. It's vital to set the first element as the maximum value, since every subsequent value will have to be compared by each value in the vector. This can be achieved with an if statement that assigns the element to max when i = 0.

In [63]:
```
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
M = 0.0 # intialize M variable to float data type
for i in range(0, len(x)): #iterate through vector x (list)
    if (i == 0):
        M = abs(x[0])
    if (abs(x[i]) > M):
        M = abs(x[i])
print(M)   # print solution
```

```
12.3
```

Problem 1.d)

Finding the minimum absolute value of a vector is found in almost the exact same way as finding the maximum absolute value of a vector (1.c). The only alteration that is needed in the algorithm is to swap the greater than comparator for the less than comparator.

In [65]:
```
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
M = 0.0 # intialize m variable to float data type
for i in range(0, len(x)): #iterate through vector x (list)
    if (i == 0):
        m = abs(x[0])
    if (abs(x[i]) < m):
        m = abs(x[i])
print(m)   # print solution
```

```
0.0
```

Problem 1.e)

Abstracting away the much of the algorithms above is possible by implementing the built-in functions to Python; 1.a) sum(), 1.b) sum(), 1.c) max(), 1.d) min()

1.a).2

In [66]:
```
# solving using sum()
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
s1 = sum(x)
print(s1)   # print solution
```

```
12.200000000000001
```

Problem 1.e) >> 1.b).2

In [67]:
```
# solving using sum()
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
x_pow_i_list = list(range(len(x))) # instantiate a list to store each elements
                                   # ith exponential value
s2 = 0.0 # intialize s2 variable to float data type
for i in range(0, len(x)): #iterate through vector x (list)
    x_pow_i_list[i] = (x[i]**(i+1)) # store the result of power of i+1
                                # (natural index) in vector x  in x_pow_i_list
    s2 = sum(x_pow_i_list) # sume all the exponential values
print(s2)   # print solution
```

```
281528.9834300001
```

Problem 1.e) >> 1.c).2

In [68]:
```
# solving using max() function and Python's built-in abs() function
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
x_abs_i_list = list(range(len(x))) # instantiate a list to store each elements
                                   # absolute value
M = 0.0 # intialize M variable to float data type
for i in range(0, len(x)): #iterate through vector x (list)
    x_abs_i_list[i] = abs(x[i]) # assign absolute values of vector x to
                                # vector_abs_i_list
M = max(x_abs_i_list) # find the maximum absolute value
print(M)   # print solution
```

```
12.3
```

In [69]:
```
# solving using max() function and numpy's abs() function
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
M = max(np.abs(x)) # assign the maximum absolute value of vector x to M
print(M)   # print solution
```

```
12.3
```

Problem 1.e) >> 1.d).2

In [70]:
```
# solving using min() function and Python's built-in abs() function
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
x_abs_i_list = list(range(len(x))) # instantiate a list to store each elements
```

```
                                  # absolute value
m = 0.0 # intialize M variable to float data type
for i in range(0, len(x)): #iterate through vector x (list)
    x_abs_i_list[i] = abs(x[i]) # assign absolute values of vector x to
                               # vector x_abs_i_list
m = min(x_abs_i_list)  # find the min absolute value
print(m)   # print solution
```

```
0.0
```

In [71]:

```
# solving using min() function and numpy's abs() function
import numpy as np
x = np.array([0.1, 1.3, -1.5, 0, 12.3]) # store vector x in a list
m = min(np.abs(x)) # assign the minimum absolute value of vector x to M
print(m)   # print solution
```

```
0.0
```

Observations

My initial observations are that there are many ways to solve one problem, not only in mathematics, but also how it can be represented in a computational algorithm as well as the computing language it is written in. The concept of abstraction becomes very relevant when realizing how much more concise code can be written by implementing certain pre-written functions from various libraries. For instance, it is possible to write an algorithm to convert every value to its absolute value by checking if the value is negative, then multiply the value by -1. Fortunately, the abs() function comes built-in with Python, so this algorithm is not necessary for practical implementation. However, the abs() function is limited to the scope of how it was written. For instance, the abs() function will only operate on one numerical value at a time, and does not work on a list-like structure. This limitation of the abs() function became apparent in solving problem 1.d). Fortunately, the library Numpy provides its own version of the abs() function, e.g. numpy.abs() which iterates through the list-like structure and converts all the values to their respective absolute value. This one function abstracted away multiple lines of code and creates a much more efficient workflow for solving a relatively simple computation. While these functions are very useful in certain situations, it is valuable to understand the algorithm they are invoking to accomplish their output to be certain the result will produce the intended behaviour.

Problem 1.b) demonstrates the issue between the natural ordering of starting a count from 1, as is standard in mathematics, in comparison to the computational way of counting starting at 0. This is the most prevalent difference in converting mathematical equations to computational algorithms, thus it should be closely analysed for every conversion.

Problem 1.e) emphasises the efficiency of the built-in function like max(), min(), and sum(). All three of these functions can iterate through a list-like structure, preventing any need to write a for loop to find the maximum, minimum, or sum of a list.

Conclusions

The less pre-built functions a program uses, the easier it is to see observe and analyse every computational operation. Inversely, the more pre-built function a program uses, the more

concise the code is which may be a more simple and abstract way of viewing the purpose of the program.

Question 2

Introduction

The purpose of this problem is to understand the ways to create various row vectors, and how to manipulate them into larger matrices (arrays) by using Numpy's various functions, as well as the built-in functionality of Python. Some helpful functions to solve these problems are; numpy.ones() generates a list with values of 1, numpy.zeros() generates a list with values of 0, numpy.linspace() generates a list of values between a start and stop value, as well as the number of elements in the list, and numpy.arange() which is a similar function that generates a list with a start and stop value (exclusive), as well as a step value that increments between the start and stop values.

Procedure

Problem 2.a)

The best way to create numerical values that are evenly spaced in a range between two values is to use Numpy's linspace() function. Numpy.linspace() makes it very easy to generate a list of values equally spaced out, while including the start and stop values. However, if the intention is to exclude the start and stop values, there is a bit of a workaround to generate those values. Numpy.linspace() is designed to easily exclude the endpoint value, as the fourth parameter is a boolean, True to include the endpoint and False to exclude the endpoint. Unfortunately, there is not a built-in way to exclude the starting value. This can be resolved by implemented the slicing index operation which can exclude the first value by starting at the second value, e.g. index 1. Then, add 1 to the number of values created, e.g. the 3rd parameter, since the slicing index skips the first value which negates 1 from the original number of values created; meaning only 4 values would be generated. After increasing the number of values parameter to 6, the function was setup to perform the proper value generation for creating a vector of 5 elements equally spaced apart, while excluding the start and stop values of 2 and 3.

In [78]:
```
# create a row vector x with 5 elements between 2 and 3 (inclusive)
import numpy as np
x = np.array(np.linspace(2, 3, 5))
print(x)
```

```
[2.   2.25 2.5  2.75 3.  ]
```

In [79]:
```
# create a row vector x with 5 elements between 2 and 3 (exclusive)
import numpy as np
x = np.linspace(2, 3, 6, False)[1:] # set endpoint to be false to exclude 3
                                    # slice the start point to exclude 2.0 and add 1 to
                                    # the number of values created to counteract the slice
print(x)
```

```
[2.16666667 2.33333333 2.5        2.66666667 2.83333333]
```

Problem 2.b)

To add 1 to the second element in the row vector, accessing the value through indexing is a straightforward approach. Python allows for altering values of a list-like structure through indexing (excluding tuples which are immuteable). Notice that the index for the second element is 1, not 2, since computers count indices starting from 0, not at 1, as is done in mathematics.

In [80]:
```
import numpy as np
x = np.array(np.linspace(2, 3, 6, False)[1:]) # set endpoint to be false to
                # exclude 3 slice the start point to exclude 2.0 and add 1 to
                # the number of values created to counteract the slice
x[1] += 1     #add 1 to the second element
print(x[1])
```

```
3.3333333333333335
```

Problem 2.c)

To create a row vector of 5 elements starting a specific value, 4, then incrementing with only the even numbers, the algorithm needs to generate a row vector consisting of 4, 6, 8, 10, 12. Numpy aranage() function cane be used to create a list of values that start (inclusive), stop (exclusive), step. Manipulating these values with some logic is how to generate an efficient strategy to create the row vector. The start value can be 4, and the stop value should be the starting position + the number of steps multiplied by the size of each step. In this case that value is 14, which is exclusive, meaning the last value that will be included is 12. The step parameter will simply be the size of the step, which is two since the goal is for all the values to be even and 2m = even, where m is any integer. Thus, the values generated are 4, 6, 8, 10, 12. Five integer values that are all even and increment evently, starting at 4.

In [1]:
```
# create a row vector y with 5 elements with incremental even values 4,6,8...
import numpy as np
y = np.array(np.arange(4, 5*2 + 4, 2)) # arange: set start to 4,
                                    # set stop to 5*2 + 4
                                    # set step to 2
print(y) # print solution
```

```
[ 4  6  8 10 12]
```

Problem 2.d)

To genearte the row vector of ones of the same size as x and y, e.g. 5, the numpy.shape() function can be used by simplying using one of the array variable (x or y) as the argument. This creates an array of the same size as x and y. To make all the elements have the value 1, the numpy.ones() function allows for using an array for the argument. This successfully generates a row vector of 5 elements with the value 1. To Create the matrix, A, all that is needed is the numpy.array() function, while using the already created arrays as the argument, within a list, where each element is the each array, x, ones_list, and y, respectively. This generates a 2D matrix (array) with 5 rows and 3 columns with x as the first row, one_row as the second row, and y as the third row.

In [101…
```
# create a row vector x with 5 elements between 2 and 3 (exclusive)
import numpy as np
x = np.linspace(2, 3, 6, False)[1:] # set endpoint to be false to exclude 3
```

```
                        # slice the start point to exclude 2.0 and add 1 to
                        # the number of values created to counteract the slice
x[1] += 1     #add 1 to the second element
ones_row = np.ones(np.shape(x)) # use dimensions of x (shape) to generate
                               # row of ones using numpy.ones()
y = np.array(np.arange(4, 5*2 + 4, 2)) # arange: set start to 4,
                                    # set stop to 5*2 + 4
                                    # set step to 2
A = np.array([x, ones_row, y]) # initalise matrix A; rows:= x, ones_row, y
print(A)
```

```
[[ 2.16666667  3.33333333  2.5         2.66666667  2.83333333]
 [ 1.          1.          1.          1.          1.        ]
 [ 4.          6.          8.         10.         12.        ]]
```

Problem 2.e)

Creating a row vector of the mean of the repective columns of A, requires generating a row vector (array) of the length of the rows of A, i.e, the number of the columns of A. Since A is a 2D array, accessing one of the rows and finding the length of the row will return the number of columns of the matrix, A. Since we are finding the mean of the columns of A, the initial values of the resultant row vector, mean_row, needs to be 0's. So, initialising mean_row with as a numpy.array() of zeros, using numpy.zeros(), with the argument of the lenth of the first for of A, will generate a row vector of 0's, with the proper length, 5. Then iterating through each element is possible by using a nested for loop, where i denotes the row and j denotes the column. With every iteration, we can store the jth (column) value in the row vector, mean_row, which will keep the ordering of the columns with their respective columns. After the first nested loop is finished, mean_row has the sums of each column stored in each of it's elements. Now, every element in mean_row needs to be divided by the number of rows of matrix A, 3. This is achieved by one for loop.

In [104…

```
# create a row vector that is the mean of the respective columns of A
import numpy as np
mean_row = np.array(np.zeros(len(A[0]))) # initalize a row of zeros,
                                        # with length of rows of matrix A
for i in range(len(A)):    # iterate through every row. i denotes row
    for j in range(len(A[0])): # iterate through every column. j denotes column
        mean_row[j] += A[i,j]   # sum each column of A into column and store
                              # values in respective columns of mean_row
for i in range(len(A[0])): # iterate through every column value of mean_row
    mean_row[i] /= len(A) # divide each column value by the number of rows of A

print(mean_row) # print solution
```

```
[2.38888889 3.44444444 3.83333333 4.55555556 5.27777778]
```

Observations

The list generation functions that come built-in with Numpy are very useful in creating specific arrays; however, it is important to understand which functions have parameters with inclusive values and exclusive values. These details can create errors, so they must be monitored closely. Some functions enable the option to choose if the stop value is inclusive or exclusive, such as numpy.linspace has its 4th parameter as a boolean that declares whether the stop value should be inclusive or exclusive. Interestingly, the start value does not have a similar option, so it is always inclusive by default. To get around this, for problem

2.d) I had to implement the slicing index functionality of Python so the list would start at the second value, so the start value would be excluded.

Conclusions

Even though there are many functions to handle different needs and situations, the programmer will still have to understand the details of every function in order to get accurate results. Also, many operations require logic such as using for loops and nested for loops accompanied with indexing to access the elements of the arrays, which is not as intuitive as implementing pre-built functions.

Question 3

Introduction

The purpose of this problem is to experiment with the various multiplication operations to understand how they vary, i.e. matrix multiplication versus element-wise multiplication. Additionaly, experimenting with the how to properly compute exponential operations. Some useful functions for this problem are; numpy.matmul(), abs(), pow(), numpy.linalg.inv(), numpy.linalg.det(), and numpy.linalg.eig().

Procedure

Problem 3.a)

Matrix addition and subtraction is very simply with Numpy's arrays, since the + and - operators perform the element-wise computation.

In [16]:
```
# compute C1 = A + B and C2 = A - B
import numpy as np
A = np.array([[1, 2], [4, -1]]) # create 2D matrix A
B = np.array([[4, -2], [-6, 3]]) # create 2D matrix B

C1 = A + B # matrix addition
print("C1 = ") # print matrix label
print(C1) # print solution
print() # indent

C2 = A - B # matrix subtraction
print("C2 = ") #print matrix label
print(C2) # print solution
```

```
C1 =
[[ 5  0]
 [-2  2]]

C2 =
[[-3  4]
 [10 -4]]
```

Problem 3.b)

In [31]:
```
# compute D1 = A * B and D2 = B * A using elementwise multiplication
import numpy as np
A = np.array([[1, 2], [4, -1]]) # create 2D matrix A
B = np.array([[4, -2], [-6, 3]]) # create 2D matrix B
print("Element-wise Multiplication:")
print("D1 =")
D1 = A * B  # element-wise multiplication
print(D1) #print solution
print()

print("D2 =")
D2 = B * A # element-wise multiplication
print(D2) # print solution
```

```
Element-wise Multiplication:
D1 =
[[  4  -4]
 [-24  -3]]

D2 =
[[  4  -4]
 [-24  -3]]
```

In [32]:
```
# compute D1 = A * B and D2 = B * A using matrix multiplication
import numpy as np
A = np.array([[1, 2], [4, -1]]) # create 2D matrix A
B = np.array([[4, -2], [-6, 3]]) # create 2D matrix B
D1 = np.matmul(A, B)     # matrix multiplications
print("Matrix Multiplication:")
print("D1 =")
print(D1)
print()

print("D2 =")
D2 = np.matmul(B, A)     # matrix multiplications
print(D2)
```

```
Matrix Multiplication:
D1 =
[[ -8   4]
 [ 22 -11]]

D2 =
[[ -4  10]
 [  6 -15]]
```

Problem 3.c)

In [77]:
```
# compute F = B + AB^(1/3) using element by element operations
import numpy as np
import math

A = np.array([[1.0, 2.0], [4.0, -1.0]]) # create 2D matrix A
B = np.array([[4.0, -2.0], [-6.0, 3.0]]) # create 2D matrix B
# B_1_3_test = B**(1.0/3.0) ERROR: negative inputs return nan
# B_1_3_test = math.pow(B, (1.0/3.0)) ERROR: pow() only work on scalars

B_1_3 = np.array(np.zeros_like(B)) # create matrix to store B^1/3 values

for i in range(len(B)):    # iterate through rows of B
    for j in range(len(B[0])):  # itreate through columns of B
        if (B[i,j] < 0):   # if B[i,j] element value is negative
            neg_B_i_j = abs(float(B[i, j])) # convert value to positive
            B_1_3[i,j] = (neg_B_i_j**(1.0/3.0)) * -1.0
```

```
        else:               # if B[i,j] element value is positive
            B_1_3[i,j] = float(B[i,j]**(1.0/3.0))

F = B + A * B_1_3
print("F = ")
print(F)
```

```
F =
[[  5.58740105  -4.5198421 ]
 [-13.26848237   1.55775043]]
```

Problem 3.d)

In [87]:
```
# find if A and are singular. If not, computer their inverse.
import numpy as np

A = np.array([[1.0, 2.0], [4.0, -1.0]]) # create 2D matrix A
B = np.array([[4.0, -2.0], [-6.0, 3.0]]) # create 2D matrix B

det_A = np.linalg.det(A)
if (abs(det_A - 0.0) < 0.0001):  # if determinant of A is 0
    print("The determinant of A is 0, thus it is singular not invertible.")
else:
    print("The determinant of A is not 0, thus it is invertible.")
    print("Inverse of A = ")
    print(np.linalg.inv(A))
    # need to find the inverse of A

det_B = np.linalg.det(B)
if (abs(det_B - 0.0) < 0.0001):  # if determinant of B is 0
    print("The determinant of B is 0, thus it is singular and not invertible.")
else:
    print("The determinant of B is not 0, thus it is invertible.")
    print("Inverse of B = ")
    print(np.linalg.inv(B))
    # need to find the inverse of B
```

```
The determinant of A is not 0, thus it is invertible.
Inverse of A =
[[ 0.11111111  0.22222222]
 [ 0.44444444 -0.11111111]]
The determinant of B is 0, thus it is singular and not invertible.
```

Problem 3.e)

The answer should be: lamda = 7 and lamda = 0

In [88]:
```
# find the eigenvalues of B
import numpy as np

B = np.array([[4.0, -2.0], [-6.0, 3.0]]) # create 2D matrix B
# eig_B = np.linalg.eig(B) # produces incorrect output
# print(eig_B)
```

```
(array([ 7.0000000e+00, -4.4408921e-16]), array([[ 0.5547002 ,  0.4472136 ],
       [-0.83205029,  0.89442719]]))
```

Observations

3.a) No need to initialise dimensions of solution arrays. 3.b) * is element-wise multiplication, not matrix multiplication, need numpy.matmult() 3.c) When hard-coding values for a matrix

(array), should default to typing in float versions of values, so that calculations are not truncated later by being of type integer by accident. Cube root of negative number must be handled manually. 3.a) and 3.b) were not affected by this type conversion issue, because addition and multiplication are both closed under integers.

Conclusions

Since computers have a finite memory, they can only store a certain degree of accuracy with numerical values. For this reason, there can be slight errors in calculations that may become more severe with increased complexity. This is why it is not safe to compare numerical values with the == operator, since it is possible two values should be equal, mathematically, but computationally, they may be very slightly varying in their values. Instead of ==, it is better to compare the values for a margin of error, checking that they are very close to the exact same value.

Question 4

Introduction

The purpose of this problem is to experiment with creating matrices (arrays) with random values between a range, as well as discovering the differences in how a dot product can be calculated using a user-defined function, mydot(), inner(), and dot(). Some helpful functions for this problem is, random.uniform(), .append(), np.shape(), and np.transpose().

Procedure

Problem 4.a)

The module, random, has a function called unifor() which creates random values between a starting and stopping value, where both values are inclusive. This can be used to generate each value, then use the append() function to append each of those random values onto a temporary list. The temporary list can then be used to generate the numpy.array() of the certain desired dimensions (shape). Once the two matrices (arrays) are created, x and y, they are ready for computation. Since the dot product is the sum of all the compents multiplied by its respective correlating index of the other vector, the algorithm can simply loop through every element using a nested for loop, accessing each element by its index, i,j (row, column). The multiplication is performed and the product is cumulatively stored by summing every resepctive product for every iteration until the final result is found.

In [86]:

```
import numpy as np
import random

n = 100          # intialise n to 100; variable used to easily change domain size
temp_list_x = []
temp_list_y = []

for i in range(n):
    temp_list_x.append([random.uniform(-10.0, 10.0)]);
```

```
for i in range(n):
    temp_list_y.append([random.uniform(-10.0, 10.0)]);

x = np.array(temp_list_x)
y = np.array(temp_list_y)

#-----------------------------------------------

x_rows = np.shape(x)[0]
x_columns = np.shape(x)[1]
y_rows = np.shape(y)[0]
y_columns = np.shape(y)[1]

def mydot(x, y):
    result = 0.0
    x_rows = np.shape(x)[0]
    x_columns = np.shape(x)[1]
    y_rows = np.shape(y)[0]
    y_columns = np.shape(y)[1]

    if (x_rows == y_rows and x_columns == y_columns):
        for i in range(x_rows):
            for j in range(x_columns):
                result += x[i,j] * y[i,j]
    return result

print("mydot: ", mydot(x, y)) # print solution
```

```
mydot:  175.74279693381342
```

Problem 4.b) work started in 4.a)

In [ ]:

```
```

Observations

The dot product is (to my knowledge) an operation that results in a scalar value, which is why mydot() is designed to produce a scalar result. However, both dot() and inner() produce a matrix (array) of the element-wise multiplication, which seems odd. Also, inner() and dot() require different arrangements in regards to how the shape of the two vectors (arrays) interact.

Conclusions

Again, this proves the need to understand the mathematival algorithm behind the pre-built functions, since they both seem to produce a result that varies from the expected outcome. While they are still useful, it may be more practical in some instances to write a user-defined function such as mydot().

Question 5

Introduction

The purpose of this problem is to create two column vectors, or more generally sets of

numbers, that are equally spaced by a certain interval for the purpose of plotting mathematical functions on a graph. Then it requires both mathematical knowdledge and programming knowdledge to convert to functions into python code. The matplotlib module provides functionality within pylot that is useful for producing graphs. The two functions I used repeatedly was the matplotlib.pyplot.figure() and the matplotlib.pyplot.plot(). Other useful functions for this problem are, numpy.log10(), math.sqrt(), numpy.arange(), pow(), math.e(), math.acos(), and np.tan()

Procedure

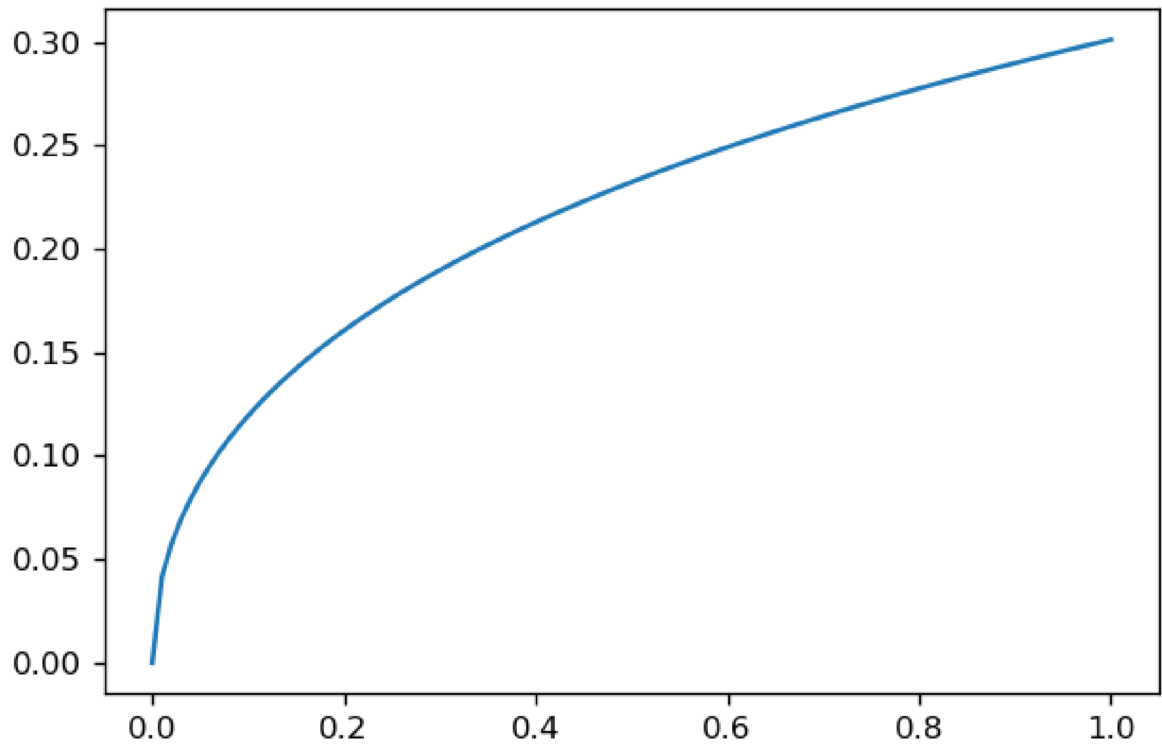
For all of these problems the same process can be applied of using the x vector (array) as the inputs and the y vector (array) as the output. Printing out the values is a good way of checking the exact output to verify if the algorithms are accurate.

Problem 5.a)

In [5]:
```
import numpy as np
import math
import matplotlib.pyplot as plt

x = np.arange(0, 1.01, 0.01)
y_a = np.arange(0, 1.01, 0.01)

def f_a(x, a):
    return np.log10(a + math.sqrt(x))

for i in range(len(x)):
    y_a[i] = f_a(x[i], 1)

print(y_a)

plt.figure(num = 0, dpi = 120)
plt.plot(x, y_a)
```

```
[0.         0.04139269 0.05744599 0.06937394 0.07918125 0.08764188
 0.09515155 0.10194464 0.10817341 0.11394335 0.11933105 0.12439416
 0.12917738 0.13371614 0.13803912 0.14216986 0.14612804 0.14993021
 0.15359052 0.15712114 0.16053263 0.16383424 0.16703409 0.17013938
 0.17315652 0.17609126 0.17894875 0.18173364 0.18445016 0.18710215
 0.18969311 0.19222625 0.19470451 0.1971306  0.19950703 0.20183611
 0.20411998 0.20636065 0.20855997 0.21071966 0.21284136 0.21492657
 0.21697673 0.21899315 0.2209771  0.22292977 0.22485226 0.22674562
 0.22861086 0.23044892 0.23226069 0.23404701 0.23580869 0.2375465
 0.23926116 0.24095337 0.24262378 0.24427302 0.24590171 0.2475104
 0.24909966 0.25067001 0.25222194 0.25375596 0.25527251 0.25677204
 0.25825497 0.25972173 0.2611727  0.26260826 0.26402877 0.2654346
 0.26682607 0.26820351 0.26956723 0.27091755 0.27225475 0.27357912
 0.27489093 0.27619044 0.27747792 0.2787536  0.28001773 0.28127054
 0.28251225 0.28374309 0.28496326 0.28617296 0.28737241 0.28856178
 0.28974126 0.29091105 0.2920713  0.2932222  0.29436392 0.2954966
 0.29662042 0.29773552 0.29884205 0.29994017 0.30103   ]
```

Out[5]:
```
[<matplotlib.lines.Line2D at 0x1fec1912fd0>]
```



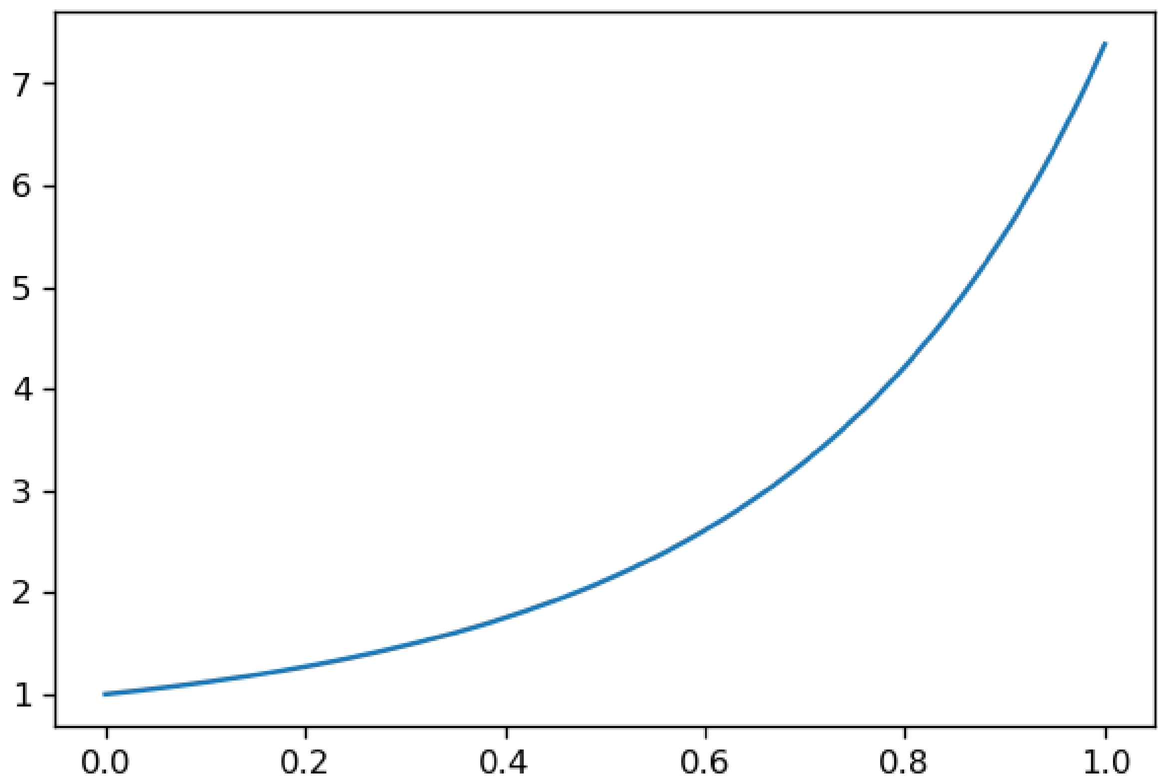
Problem 5.b)

In [6]:

```
import numpy as np
import math
import matplotlib.pyplot as plt

x = np.arange(0, 1.01, 0.01)
y_b = np.arange(0, 1.01, 0.01)

def f_b(x, a):
    return (pow(a, (x + pow(x, 2))))

for i in range(len(x)):
    y_b[i] = f_b(x[i], math.e)

print(y_b)

plt.figure(num = 0, dpi = 120)
plt.plot(x, y_b)
```

```
[1.         1.01015118 1.0206095  1.03138236 1.0424774  1.05390256
 1.06566605 1.07777637 1.09024234 1.10307309 1.11627807 1.12986708
 1.14385027 1.15823813 1.17304156 1.18827182 1.20394059 1.22005995
 1.23664244 1.25370103 1.27124915 1.28930073 1.30787018 1.32697246
 1.34662303 1.36683794 1.38763381 1.40902785 1.43103792 1.45368251
 1.47698079 1.50095264 1.52561865 1.55100018 1.57711937 1.60399918
 1.63166342 1.66013679 1.68944488 1.71961426 1.7506725  1.78264818
 1.81557097 1.84947164 1.88438216 1.92033567 1.9573666  1.99551068
 2.03480502 2.07528813 2.11700002 2.15998224 2.20427796 2.24993201
 2.29699099 2.34550329 2.39551922 2.44709107 2.50027319 2.55512207
 2.61169647 2.67005748 2.73026862 2.79239598 2.85650829 2.92267707
 2.99097673 3.06148472 3.13428162 3.20945134 3.28708121 3.36726216
 3.4500889  3.53566005 3.62407832 3.71545074 3.80988879 3.90750866
 4.00843143 4.11278329 4.22069582 4.33230616 4.44775734 4.56719851
 4.69078525 4.81867985 4.95105161 5.08807721 5.22994102 5.37683548
 5.52896148 5.68652875 5.84975629 6.0188728  6.19411718 6.37573896
 6.56399886 6.75916931 6.96153503 7.17139359 7.3890561 ]
```

Out[6]: [<matplotlib.lines.Line2D at 0x1fec197f790>]



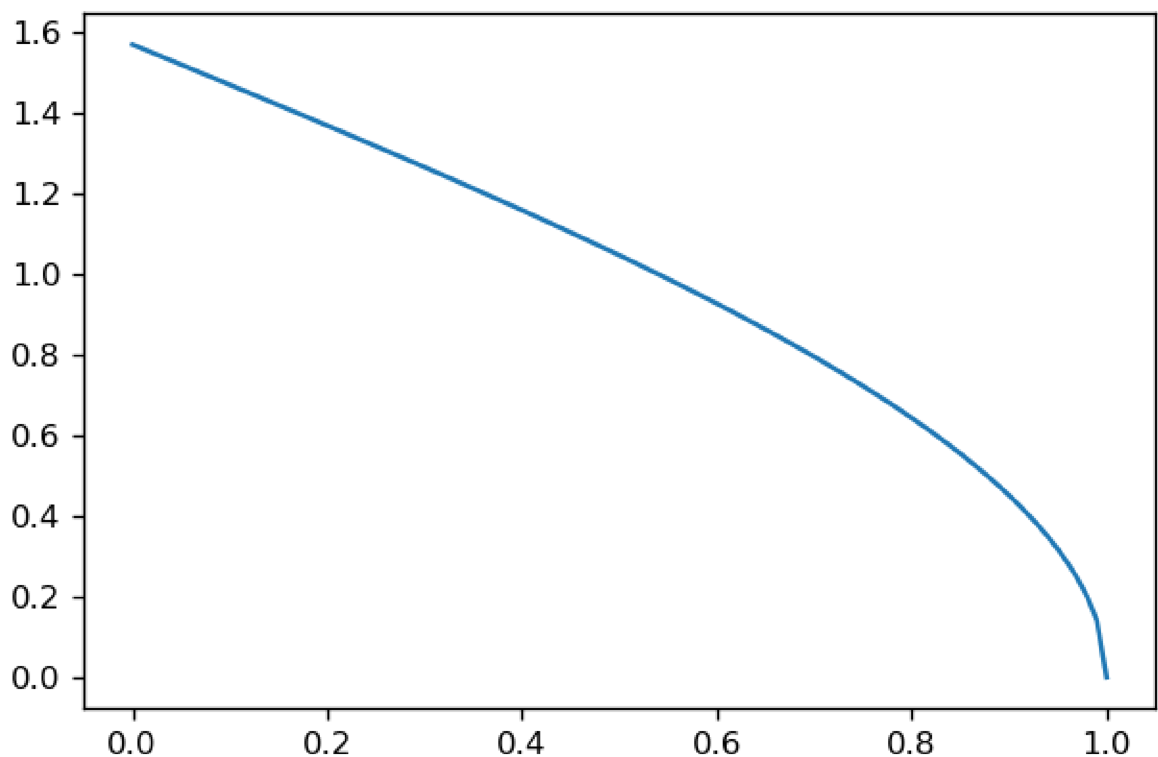
Problem 5.c)

In [7]:
```
import numpy as np
import math
import matplotlib.pyplot as plt

x = np.arange(0, 1.01, 0.01)
y_c = np.arange(0, 1.01, 0.01)

def f_c(x):
    return (math.acos(x))

for i in range(len(x)):
    y_c[i] = f_c(x[i])

print(y_c)

plt.figure(num = 0, dpi = 120)
plt.plot(x, y_c)
```

```
[1.57079633 1.56079616 1.55079499 1.54079182 1.53078565 1.52077547
 1.51076027 1.50073903 1.49071075 1.48067438 1.47062891 1.46057328
 1.45050644 1.44042735 1.43033491 1.42022805 1.41010567 1.39996666
 1.38980988 1.37963418 1.36943841 1.35922137 1.34898186 1.33871864
 1.32843048 1.31811607 1.30777412 1.2974033  1.28700222 1.27656949
 1.26610367 1.25560329 1.24506684 1.23449275 1.22387943 1.21322522
 1.20252843 1.19178731 1.18100003 1.17016473 1.15927948 1.14834226
 1.13735101 1.12630355 1.11519765 1.10403099 1.09280113 1.08150555
 1.07014161 1.05870657 1.04719755 1.03561154 1.02394538 1.01219576
 1.00035922 0.98843209 0.97641053 0.96429047 0.95206764 0.93973749
 0.92729522 0.91473574 0.90205362 0.88924312 0.87629806 0.86321189
 0.84997757 0.83658754 0.82303369 0.80930727 0.79539883 0.78129812
 0.76699401 0.75247438 0.73772597 0.72273425 0.70748321 0.69195518
 0.67613051 0.65998733 0.64350111 0.62664421 0.60938531 0.59168864
 0.5735131  0.55481103 0.53552665 0.51559401 0.49493413 0.47345116
 0.45102681 0.42751226 0.40271584 0.37638348 0.34816602 0.31756043
 0.28379411 0.24556552 0.20033484 0.14153947 0.        ]
```

Out[7]:

```
[<matplotlib.lines.Line2D at 0x1fec19e3d60>]
```



Problem 5.d)

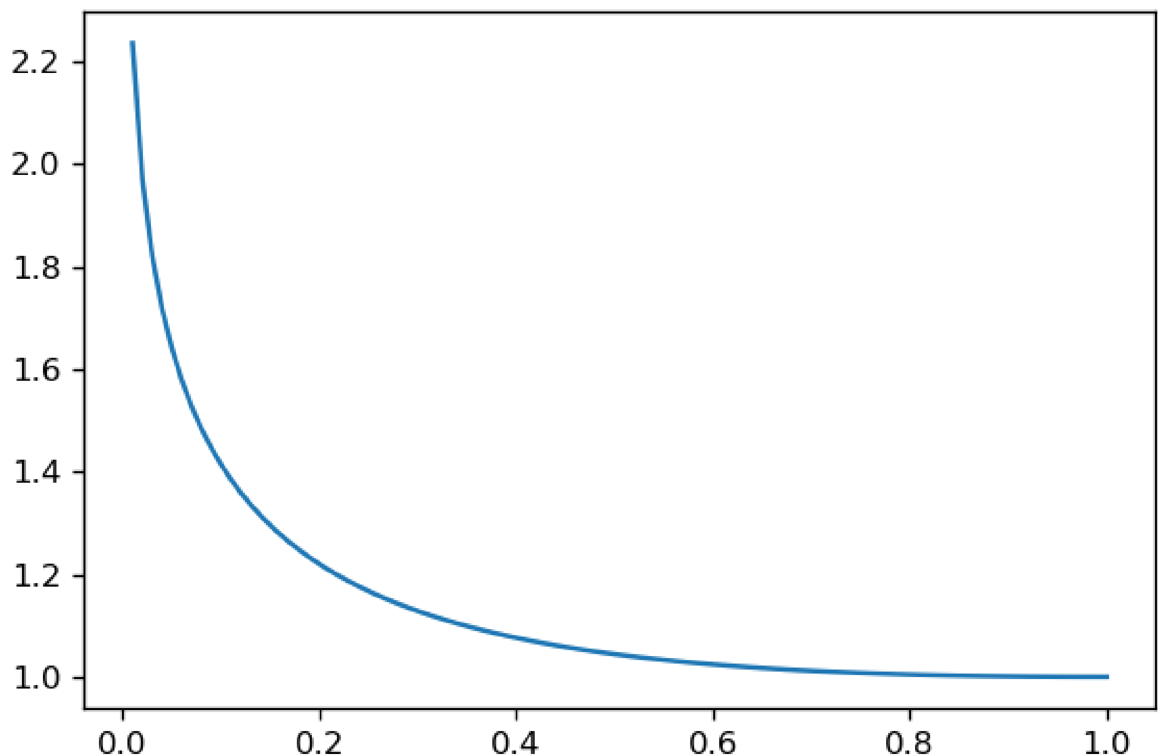
In [8]:

```python
import numpy as np
import math
import matplotlib.pyplot as plt

x = np.arange(0, 1.01, 0.01)
y_d = np.arange(0, 1.01, 0.01)

def f_d(x):
    return (math.sqrt(1 + pow(np.log10(x),2)))

for i in range(len(x)):
    y_d[i] = f_d(x[i])

print(y_d)

plt.figure(num = 0, dpi = 120)
plt.plot(x, y_d)
```

```
[       inf 2.23606798 1.97142057 1.82185611 1.71878919 1.64093847
 1.57889657 1.5276775  1.48432192 1.4469308  1.41421356 1.38525376
 1.3593775  1.33607499 1.31495146 1.29569503 1.27805516 1.26182759
 1.24684363 1.23296244 1.22006519 1.20805078 1.19683246 1.18633526
 1.17649399 1.16725157 1.1585578  1.15036828 1.14264353 1.13534837
 1.12845123 1.12192376 1.11574036 1.10987784 1.10431517 1.09903319
 1.09401438 1.08924274 1.08470357 1.08038334 1.0762696  1.07235085
 1.06861644 1.06505651 1.0616619  1.05842409 1.05533515 1.05238767
 1.04957475 1.0468899  1.04432708 1.04188061 1.03954514 1.03731567
 1.03518747 1.0331561  1.03121737 1.02936732 1.02760222 1.02591852
 1.02431288 1.02278213 1.02132326 1.01993344 1.01860994 1.01735019
 1.01615176 1.0150123  1.01392962 1.01290158 1.01192619 1.01100152
 1.01012574 1.0092971  1.00851393 1.00777462 1.00707765 1.00642156
 1.00580494 1.00522645 1.0046848  1.00417876 1.00370715 1.00326883
 1.00286272 1.00248775 1.00214293 1.00182729 1.0015399  1.00127987
 1.00104633 1.00083845 1.00065545 1.00049654 1.00036099 1.00024809
 1.00015714 1.00008749 1.00003849 1.00000953 1.        ]
C:\Users\User\AppData\Local\Temp\ipykernel_13444\567844951.py:9: RuntimeWarning: d
ivide by zero encountered in log10
  return (math.sqrt(1 + pow(np.log10(x),2)))
```

Out[8]:

```
[<matplotlib.lines.Line2D at 0x1fec1a4dd90>]
```



Problem 5.e)

In [9]:

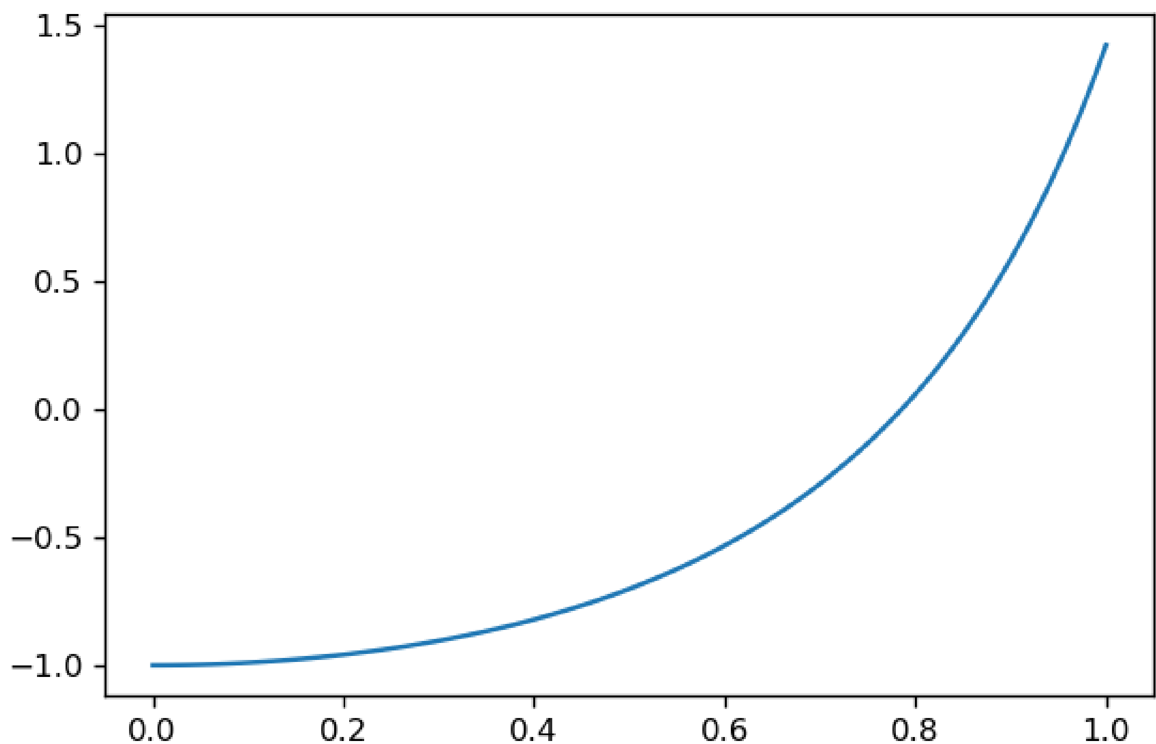
```python
import numpy as np
import math
import matplotlib.pyplot as plt

x = np.arange(0, 1.01, 0.01)
y_e = np.arange(0, 1.01, 0.01)

def f_e(x):
    return (pow(np.tan(x), 2) - 1)

for i in range(len(x)):
    y_e[i] = f_e(x[i])
```

```
print(y_e)

plt.figure(num = 0, dpi = 120)
plt.plot(x, y_e)
```

```
[-1.         -0.99989999 -0.99959989 -0.99909946 -0.99839829 -0.99749583
 -0.99639134 -0.99508395 -0.99357259 -0.99185606 -0.98993295 -0.98780172
 -0.98546062 -0.98290775 -0.98014102 -0.97715815 -0.97395667 -0.97053394
 -0.96688709 -0.96301308 -0.95890864 -0.9545703  -0.94999435 -0.94517689
 -0.94011374 -0.9348005  -0.92923254 -0.92340493 -0.91731251 -0.91094981
 -0.90431108 -0.89739029 -0.89018106 -0.88267672 -0.87487022 -0.8667542
 -0.85832089 -0.84956217 -0.84046948 -0.83103386 -0.82124589 -0.81109572
 -0.80057296 -0.78966676 -0.7783657  -0.7666578  -0.75453052 -0.74197065
 -0.72896435 -0.71549709 -0.70155359 -0.68711783 -0.67217295 -0.65670125
 -0.64068413 -0.624102   -0.60693428 -0.58915932 -0.5707543  -0.55169524
 -0.53195683 -0.51151242 -0.49033393 -0.46839171 -0.4456545  -0.42208927
 -0.39766116 -0.37233332 -0.34606678 -0.31882031 -0.29055028 -0.26121047
 -0.2307519  -0.19912262 -0.1662675  -0.13212804 -0.09664202 -0.05974333
 -0.02136161  0.01857807  0.06015556  0.10345601  0.1485703   0.19559554
  0.24463554  0.29580137  0.34921201  0.40499502  0.46328729  0.52423582
  0.58799873  0.65474618  0.72466154  0.79794263  0.87480306  0.95547387
  1.04020512  1.12926793  1.22295655  1.32159081  1.42551882]
```

Out[9]:

```
[<matplotlib.lines.Line2D at 0x1fec2aae3d0>]
```



Observations

After years of plotting mathematical functions by hand, it is astonishing to witness the speed and relatively accurate computation and graphing of functions. This technique of converting a function to code then graphing it is a useful way to visualise functions graphically, as well as find solutions.

Conclusions

When dealing with a large domain, it would be more efficient to use programming to visualise and solve mathematical functions.

Question 6

Introduction

The purpose of this problem is to experiment with solving linear equations using Numpy's solve() function and Scipy's solve_banded() function, then comparing how they work, as well as their computational processing time.

Procedure

Problem 6.a)

Numpy's solve() function makes it quick and simply to compute find the solutions to an Ax = b problem. All that is needed is the matrices to be input as the two parameters and the function returns the solution matrix.

In [88]:
```
import numpy as np

A = np.array(([[3.0, 2.0, 1.0, 0.0],
    [-3.0, -2.0, 7.0, 1.0],[3.0, 2.0, -1.0, 5.0],[0.0, 1.0, 2.0, 3.0]]))

b = np.array([[6.0],[3.0],[9.0],[6.0]])

x = np.linalg.solve(A, b)
print("x = ")
print(x)
```

```
x =
[[1.]
 [1.]
 [1.]
 [1.]]
```

Problem 6.b)

Matrix A is already in banded form.

Problem 6.c)

In [69]:
```
import numpy as np
from scipy.linalg import solve_banded

A = np.array(([[3.0, 2.0, 1.0, 0.0],
    [-3.0, -2.0, 7.0, 1.0],[3.0, 2.0, -1.0, 5.0],[0.0, 1.0, 2.0, 3.0]]))

b = np.array([[6.0],[3.0],[9.0],[6.0]])

# convert banded matrix to matrix diagonal ordered form
(n, n) = A.shape     # store the dimensions of A in n; always sqaure nxn matrix

lower = 2
upper = 2

Ab_rows = lower + upper + 1
```

```
Ab = np.zeros((Ab_rows, n)) # rows = number of nonzero diagonals in matrix A
                            # rows = lower + upper + 1 (for the main diagonal)
                            # columns = number of columns in matrix A


for j in range(n):        # convert matrix banded form to diagonal ordered form
    M = max(1, j+1 - upper)
    m = min(n, j+1 + lower)
    for i in range(M-1, m):
        Ab[upper + i - j, j] = A[i, j]

x = solve_banded((lower, upper), Ab, b)
print()
print(x)
```

```
[[1.]
 [1.]
 [1.]
 [1.]]
```

Problem 6.d).1 Numpy's solve time

In [81]:

```
# importing the required module
import numpy as np
import timeit

# code snippet to be executed only once
mysetup = '''import numpy as np
import timeit'''

# code snippet whose execution time is to be measured
mycode = '''
def function1():
    A = np.array(([[3.0, 2.0, 1.0, 0.0],
        [-3.0, -2.0, 7.0, 1.0],[3.0, 2.0, -1.0, 5.0],[0.0, 1.0, 2.0, 3.0]]))

    x = np.linalg.solve(A, b)
'''

# timeit statement
print (timeit.timeit(setup = mysetup,
                     stmt = mycode,
                     number = 100000000))
```

```
4.3740741000001435
```

Problem 6.d).2 Scipy's solve_banded time

In [80]:

```
# importing the required module
import numpy as np
from scipy.linalg import solve_banded
import timeit

# code snippet to be executed only once
mysetup = '''import numpy as np
from scipy.linalg import solve_banded
import timeit'''

# code snippet whose execution time is to be measured
mycode = '''
def function1():
    A = np.array(([[3.0, 2.0, 1.0, 0.0],
        [-3.0, -2.0, 7.0, 1.0],[3.0, 2.0, -1.0, 5.0],[0.0, 1.0, 2.0, 3.0]]))
```

```
    b = np.array([[6.0],[3.0],[9.0],[6.0]])

    # convert banded matrix to matrix diagonal ordered form
    (n, n) = A.shape     # store the dimensions of A in n; always sqaure nxn matrix

    lower = 2
    upper = 2

    Ab_rows = lower + upper + 1

    Ab = np.zeros((Ab_rows, n)) # rows = number of nonzero diagonals in matrix A
                                # rows = lower + upper + 1 (for the main diagonal)
                                # columns = number of columns in matrix A


    for j in range(n):      # convert matrix banded form to diagonal ordered form
        M = max(1, j+1 - upper)
        m = min(n, j+1 + lower)
        for i in range(M-1, m):
            Ab[upper + i - j, j] = A[i, j]

    x = solve_banded((lower, upper), Ab, b)
'''

# timeit statement
print (timeit.timeit(setup = mysetup,
                     stmt = mycode,
                     number = 100000000))
```

4.69378130000041

Problem 6.d).3 Results

Scipy's solve_banded() function processing time takes longer than Numpy's solve() function.

Observations

The convenience of numpy.solve() working on the matrix in its original form makes it easier to use. Having to rearrange the matrix into diagonal ordered form is a cumbersome process. For 6.d), I had to set the number of iterations of the timeit algorithm to a high value in order to notice any difference between runtimes of numpy.solve() and scipy.linalg.solve_banded.

Conclusions

Numpy's solve() function is faster to compute and easier to use. The only advantage to Scipy's solve_banded() function is efficiency with memory, so I'm assuming it is the better choice when the data is so large that optimisation of memory and its management is needed.