Project report (1)

Name (300493343)

Question 1

Introduction

The purpose of this project was to use learn the functions from NUMPY. Some of the functions include computing the sum as well as calculating the minimun and maximum values of a vector.

Procedure

In [10]:
```
import numpy
from numpy import array

#question 1a
x = [0.1, 1.3, -1.5, 0, 12.3]
#s = 0; #incrment later
def my_sum(x):
    s = 0;
    for i in range(len(x)):
        #print(x[i])
        s += x[i]
    return s

print("s = ", my_sum(x))
```

```
s =  12.200000000000001
```

In [14]:
```
#question 1b
def my_sum2(x):
    s2 = 0
    for i in range(len(x)):
        #print(x[i])
        s2 += x[i]**i #power
    return s2

print("s2 = ", my_sum2(x))
```

```
s2 =  22893.214100000005
```

In [15]:
```
#question 1c
def my_max(x):
    s3 = 0
    for i in range(len(x)):
        #print(x[i])
        if(abs(x[i]) > s3):
                s3 = abs(x[i])
    return s3

print("s3 = ",my_max(x))
```

```
s3 =  12.3
```

In [16]:
```
#question 1d

def my_min(x):
    s4 = 0;
    for i in range(len(x)):
        #issue with this is that sometimes there won't be a 0 in the array
        #print(x[i])
        if(abs(x[i]) <= s4):
            s4 = abs(x[i])
    return s4

print("s4 = ", my_min(x))
```

```
s4 =  0
```

In [17]:
```
#question 1e
sumdif = abs(numpy.sum(x) - my_sum(x))
maxdif = abs(numpy.max(x) - my_max(x))
mindif = (numpy.min(x) - my_min(x))
print("difference between numpy sum and my sum = ", sumdif)
print("difference between numpy sum and my max = ", maxdif)
print("difference between numpy sum and my min = ", mindif)
```

```
difference between numpy sum and my sum =  0.0
difference between numpy sum and my max =  0.0
difference between numpy sum and my min =  -1.5
```

Observations

here I have noticed that the function I have made for finding the minimum absolute value was different to that of the python's finding minimum value function.

Discussion

I have noticed the reason why my "find minimum aboslute value" was different from python's "finding mimum value". this was because python's min() function does not consider an aboslute value hence it will consider negative values to be the minimum value.

Conclusions

In conclusion, sometimes it's better to have a custom function to carry out a certain algrothim instead of using a built in function since you can tailor it to your needs. for example if you're in charge of a bank and you want to find the member with the lowest deposit but you want to ignore any withdrawal you might consider finding the minimum aboslute value as a more useful alogrithm than using min(listOfMembers).

Question 2

Procedure

In [39]:
```
#question 2a
import numpy
from numpy import array

x = numpy.linspace(2,3,5)
print("row vector x = ",x)
```

```
row vector x =  [2.   2.25 2.5  2.75 3.  ]
```

In [40]:
```
#question 2b
x[1] += 1
print("row vector x[1] + 1 = ", x)
```

```
row vector x[1] + 1 =  [2.   3.25 2.5  2.75 3.  ]
```

In [41]:
```
#question 2c
y = numpy.linspace(4,(4 + 2*5 - 2),5) #must be have 5 elements
print("row vector y = ",y)
```

```
row vector y =  [ 4.  6.  8. 10. 12.]
```

In [50]:
```
#question 2d

X = numpy.array([x])#to make an array into a vector
o = numpy.array([numpy.ones(5)])
Y = numpy.array([y])#needs to be captial or else it will update again and again

#z is temp in order to concate twice
z = numpy.concatenate((X,o))
#print(z)
A = numpy.concatenate((z,Y))#adding y onto the matrix
print("after concatenating all the rows together: \n")
print(A)
```

```
[[1. 1. 1. 1. 1.]]
after concatenating all the rows together:

[[ 2.    3.25  2.5   2.75  3.  ]
 [ 1.    1.    1.    1.    1.  ]
 [ 4.    6.    8.   10.   12.  ]]
```

In [47]:
```
#question 2e

def column_mean(A):
    #have decided to make the loop size depend on the amount of columns in the matrix. this
    ztemp = [None] * len(A[0])
    for i in range(len(A[0])):
        tempVector = A[:,i] #each column into a vector
        tempElement = sum(tempVector)/len(A) #calculating the mean
        ztemp[i] = tempElement #inputting it in
    return ztemp

z = column_mean(A)

print("mean value of the all the columns = \n",z)
```

```
mean value of the all the columns =
 [2.3333333333333335, 3.4166666666666665, 3.8333333333333335, 4.583333333333333, 5.333333
333333333]
```

Observations

I have noticed that I need to put [] around arrays in order to turn them into vectors. Furthermore, I noticed that concatenation of row vectors are required in order to make it into a matrix. Some numpy librarys have made this question easier; numpy.ones(). This built in function allowed a row vector of 1s to be generated.

Discussion

I believe python does not recognise the numpy.linspace() as a vector hence would not be able to concatenate into a matrix. To over come this i have reassigned the arrays into numpy.arrays in order for python to recognise them as vectors. numpy.ones() is a better tool than creating the row of 5 matrixes manually. This is because if you were to manually create a row vector "[[1,1,1,1,1]]" you might miss one of the column which will result in the program failing to execute.

Conclusion

In conclusion, I find linspace very useful to create an evenly spaced out array but the draw back would be turning it into a vector later. On another note, Using built in functions creates a higher chance of not making human errors; the row of ones.

Question 3

Procedure

In [1]:
```
#question 3a
import numpy
from numpy import array
from numpy import linalg as npl

A = numpy.array([[1, 2], [4, -1]])
B = numpy.array([[4, -2], [-6, 3]])
C1 = A + B #addition
print("C1 = A + B\n",C1)
C2 = A - B
print("C2 = A - B\n",C2)
```

```
C1 = A + B
 [[ 5  0]
 [-2  2]]
C2 = A - B
 [[-3  4]
 [10 -4]]
```

In [2]:
```
#question 3b
D1 = numpy.dot(A,B)
print("D1 = A*B\n",D1)
D2 = numpy.dot(B,A)
print("D2 = B*A\n",D2)
```

```
D1 = A*B
 [[ -8   4]
 [ 22 -11]]
D2 = B*A
 [[ -4  10]
 [  6 -15]]
```

In [3]:
```
#question 3c
tempB = numpy.cbrt(B)#cube rooting of B
#built in function, works with complex numbers
z = numpy.dot(A,tempB) #let z = A*B**1/3
#print(z)
f = B + z #matrix f
print("F matrix: \n",f)
```

```
F matrix:
 [[ 1.95315987 -0.37542191]
 [ 2.1667248  -3.48193377]]
```

In [6]:
```
#question 3d
import numpy.linalg as npl
if(npl.det(A) >= 0.00000001 or npl.det(A) <= 0.00000001):
    print("inverse of A:\n")
    print(npl.inv(A))
else:
    print("A is a singular matrix")

print("\n")
if(npl.det(B) >= 0.00000001 or npl.det(B) <= -0.00000001):
    print(npl.det(B))
    print("inverse of B:\n")
    print(npl.inv(B))
else:
    print("B is a singular matrix")
```

```
inverse of A:

[[ 0.11111111  0.22222222]
 [ 0.44444444 -0.11111111]]


B is a singular matrix
```

In [8]:
```
#question 3e
B = numpy.array([[4, -2], [-6, 3]])
print(B)
e = npl.eigvals(B)
print("eigen values of e =", e)
#i believe there's some rounding errors since 10^-16 is a small number
```

```
[[ 4 -2]
 [-6  3]]
eigen values of e = [ 7.0000000e+00 -4.4408921e-16]
```

Observations

Based on my observations I have noticed that multiplying matrices via * is different from numpy.dot(matrixA, matrixB). Further more I have noticed that numpy.cbrt() can deal with complex numbers as a root of a negative number will always be an imaginary number.

Discussion

I believe the reason why mutliplying matrix requires to call np.dot() is because python would normally multiply each element respectively, where as the dot product requires special procedures. Hence using the dot() will actually allow us to find the product of those 2 matrices. Along with this, numpy.cbrt() (cube root) was very useful as it saves me time that i would use to write a function that would deal with complex numbers.

Conclusion

In conclusion, I find that the utilisation of numpy functions to be extremely useful. This is because using algorithm that are already pre-written almost always save time. In terms of versatility, it can deal with complex numbers which can save a lot of time.

Question 4

Procedure

In [7]:
```
#question 4a and 4b
import numpy
from numpy import random

mydotsum = 0;

def mydot(x,y):
    sum = 0;
    for i in range(N):
        sum += (x[i] * y[i])

    return sum

a = -10
b = 10
N = 100

x = a + (b-a)*random.random(N)
y = a + (b-a)*random.random(N)

tempx = numpy.array(x)
tempy = numpy.array(y)

mydotsum = mydot(x,y)
print("my dot product is given by", mydotsum)
print("np.dot gave me", numpy.dot(x,y))
print("np.inner gave me", numpy.inner(x,y))
```

```
my dot product is given by 276.4597079218745
np.dot 276.4597079218744
np.inner 276.4597079218744
```

4b) i have noticed that my function can only work with real values where as numpy's .inner() and .dot() can work with complex numbers

Observations

I have noticed that numpy.dot() only accepts row vectors and not column vectors. Another thing I noticed is that numpy.dot and numpy.inner can have imaginary number inputs.

Discussion

I believe that accepting imaginary number as inputs provides a lot more versatility as it's not only limited to real integers. However, I believe an addition to the numpy.dot() would be to accept column vectors (1,100) instead of only accepting row vectors/ arrays.

Conclusion

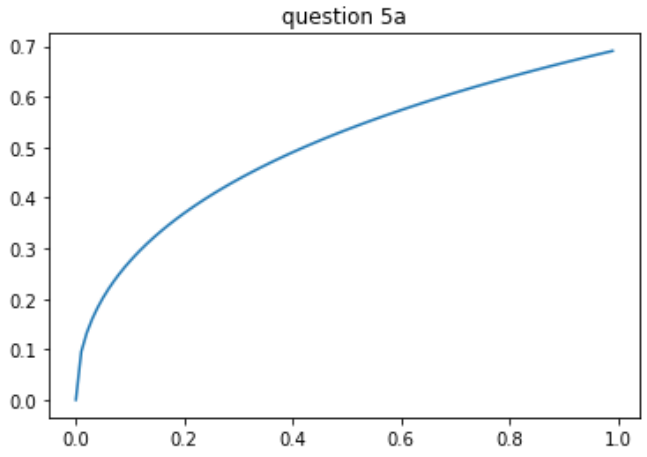
In conclusion, numpy.dot() as well as numpy.inner() are both very useful functions in the numpy library as it can accept a wide range of values.

Question 5

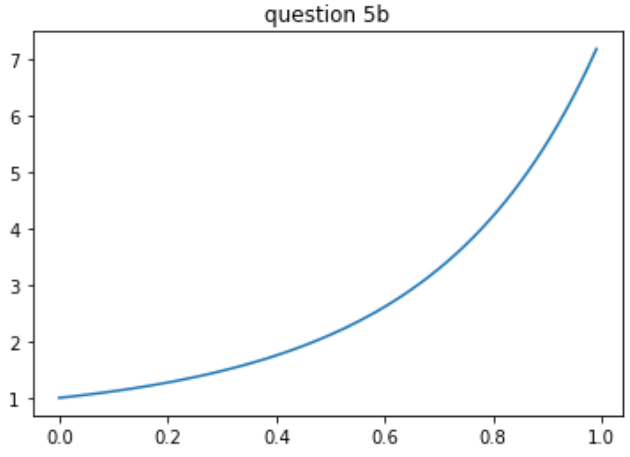
Procedure

In [71]:
```
import numpy as np
from matplotlib import pyplot as plt

#question 5a
x = np.arange(0.0,1.0,0.01)
y = np.log(1 + x**(1/2))
print(x)
plt.plot(x,y)
plt.title("question 5a")
plt.show()
```

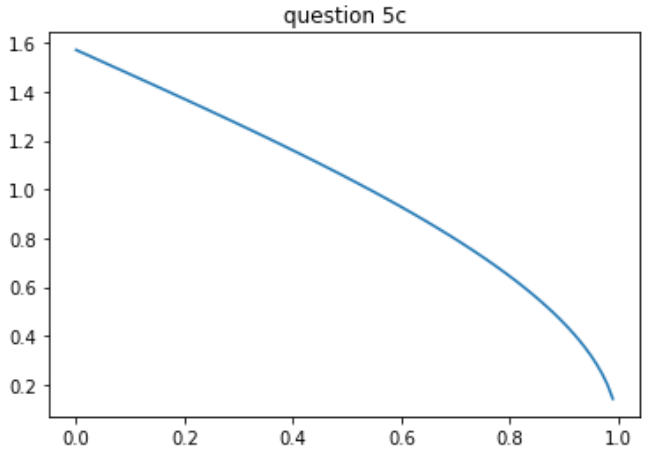
```
[0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1  0.11 0.12 0.13
 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27
 0.28 0.29 0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4  0.41
 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49 0.5  0.51 0.52 0.53 0.54 0.55
 0.56 0.57 0.58 0.59 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69
 0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83
 0.84 0.85 0.86 0.87 0.88 0.89 0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97
 0.98 0.99]
```



In [72]:
```
#question 5b

y1 = np.exp(x + x**2)

plt.plot(x,y1)
plt.title("question 5b")
plt.show()
```

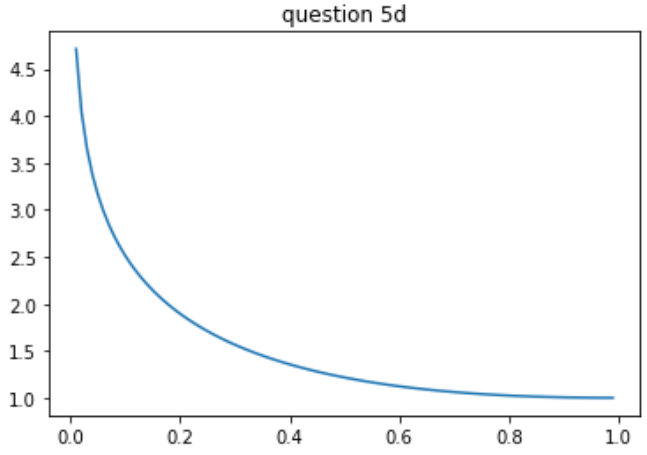


In [73]:
```
#question 5c
y2 = np.arccos(x)

plt.plot(x,y2)
plt.title("question 5c")
plt.show()
```

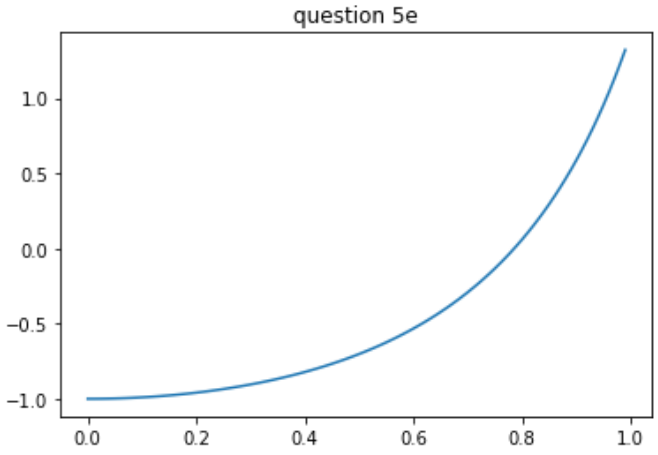


In [74]:
```
#question 5d
y3 = (1+(np.log(x))**2)**(1/2)
plt.plot(x,y3)
plt.title("question 5d")
plt.show()
```

```
C:\Users\hau shian\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: RuntimeWarning: d
ivide by zero encountered in log
```



In [76]:
```
#question 5e
y5 = np.tan(x)**2 - 1

plt.plot(x,y5)
plt.title("question 5e")
plt.show()
```



Observation

I have noticed that np.arrange allows you to increment between 2 values which makes plotting graphs very simple. On another note, I noticed that plt.plot still runs even when there's an undefined spot.

Discussion

I believe that plotting on the graph allows you to have a big picture view of a set of data hence having 1 data point not showing (undefined, question 5d) should not make a huge difference. Which seems to make sense in our case where plt.plot still runs.

Conclusion

In conclusion, the plotting library is very versatile as it doesn't fail the program if it has an undefined value.

Question 6

In [56]:
```
#question 6a
import numpy
from numpy import array

A1 = numpy.array([[3, 2, 1, 0], [-3, -2, 7, 1], [3, 2, -1, 5], [0, 1, 2, 3]])
b = numpy.array([[6], [3], [9], [6]])
#x = A^-1 * b

def solve(a, b):
    x = numpy.linalg.solve(A1,b)
    return x

print("b is given as vector: ")
print(solve(A1,b))
```

```
b is given as vector:
[[1.]
 [1.]
 [1.]
 [1.]]
```

In [77]:
```
#question 6b

upper = 2
lower = 2
(n,n) = A1.shape

Ab = numpy.zeros((5,4)) #setting up banded array dimensions
                        #banded matrix is number of diagonals, number of elements in centr

for j in range(n):   # Convert the matrix into banded format
    M = max(1, j+1-upper)
    m = min(n, j+1+lower)
    for i in range(M-1,m):
        Ab[upper+i-j,j] = A1[i,j]

print("banded matrix: ")
print(Ab)
```

```
banded matrix:
[[ 0.  0.  1.  1.]
 [ 0.  2.  7.  5.]
 [ 3. -2. -1.  3.]
 [-3.  2.  2.  0.]
 [ 3.  1.  0.  0.]]
```

In [78]:
```
import scipy
from scipy.linalg import solve_banded

x = solve_banded((lower,upper), Ab,b)
print("b is given as (solve_banded): ")
print(x)
```

```
b is given as (solve_banded):
[[1.]
 [1.]
 [1.]
 [1.]]
```

In [65]:
```
import time
start = time.time()

x = solve(A1,b)
end = time.time()
t = start - end

start2 = time.time()
x2 =  solve_banded((lower,upper), Ab,b)
end2 = time.time()

t2 = abs(start2 - end2)
if(t2 > t):
    print("solve_banded was faster")
if(t > t2):
    print("numpy was faster")
```

```
solve_banded was faster
```

Observation

I noticed that getting the matrix into banded form was pretty tricky, however solving the matrix in banded form was a lot faster.

Discussion

I believe that solving the matrix in banded form was quicker for the computer where as solving it Ax = b (normal way) would be alot quicker for us. However, the difference between these 2 methods for the computer was marginal as they both rounded to 0 if I was to print it. Nonetheless it would be a safe assumption to make that solve_banded would be a lot faster if there was a larger matrix instead.

Conclusion

In conclusion, solve_banded is a very useful function which allows the computer to solve matrix problems in a very short time frame.