Assignment 1

AA 651

Python Assignment 1

August 29, 2021

Professor: Dr. Suman Majumdar
Discipline of Astronomy, Astrophysics and Space Engineering (DAASE)

Student's name: Keshav Aggarwal
Roll No.: 2103121014

Index

1 Introduction

As it was my first time working with a programming language, I was quite afraid of whether I will be able to do it at all. After a few lectures and a bit of working with online codes I found, I hoped I would be somewhat better. But, this assignment did not use any of that. To make this assignment I broke down the problem to its most basic levels and tried to write the code for each step separately and then combining it to make it work as a whole. I am quite interested in how this language applies in astronomy and to apply it as well. I hope I will be somewhat better by then. During the course of this assignment, I also understood why self explanatory, clean coding is a must and I have attempted to do the best I could with my current skill. Being a first time user of Python, or for any programming language for that matter, it was really exciting whenever I got the output. During this assignment, to keep my code self explanatory and clean, I have attempted to make tables in the solutions for almost all problems; for better readability, and of course, presentation.

In this assignment, I have attempted to explain the best to my ability on how I have tackled these problems. To make matters easier for the reader, I have inserted the code in the assignment, as well as the I/O screenshots of the code when run.

2 Problems

2.1 Problem 1

Encode the following algorithm and run it to determine the smallest positive number that can be represented on the computer you are using:

$inputs \leftarrow 1.0$
for $k = 1, 2, 3, ..., 100do$
$s \leftarrow 0.5 * s$
$t \leftarrow s + 1.0$
if $t \Leftarrow 1.0$ then
$s \leftarrow 2.0 * s$
output $k - 1, s$
stop
endif
end

Do this for both single precision and double precision floating point numbers.

Answer-

In this problem, we have been given an algorithm which we have to write in the proper form and determine the value of the smallest integer that can be represented on my computer.

To begin, I first wrote a first rough draft of the algorithm and took out errors line by line. In this rough draft I assigned the values specified in the problem and then attempted to generate an output at first. After making a few adjustments, I was able to at least get an output. I set an initial value of s and then operated on it with the for... loop with the given range of k.Within the same loop I defined t which depended on s for its values and used an if... break to stop after the desired values were obtained.

Then I made two copies of the code, one for single precision(float32) and the other for the more precise, double precision(float64); the default setting that the computer uses.

Both the code used, as well as the I/O screenshots have been attached in the following pages.

```
1
2  import numpy as np
3
4  s=1.00
5
6  #range gives you a range of numbers
7  #range(inclusive_start, exclusive_end, step).
8  for k in range(1,101,1):
9
10     s= np.float32(0.5*s)
11
12     t= np.float32(s + 1.00)
13
14
15     if t <= 1.00:
16
17       s = 2.0*s
18       break
19 # Gives Single Precision output
20 print ("Single Precision:", k-1,s)
```

Listing 1: Single Precision

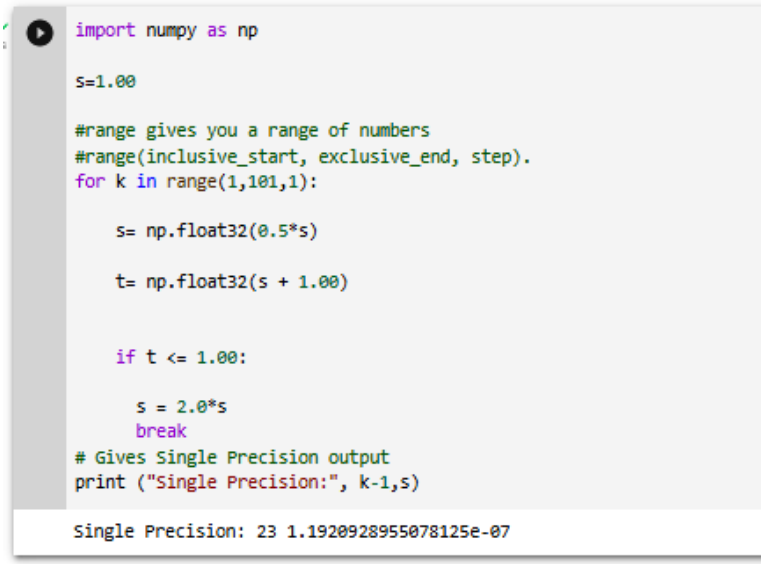


Figure 1: Single Precision

```
import numpy as np

s=1.00

# range gives you a range of numbers.
# range(inclusive_start, exclusive_end, step).
for k in range(1,101,1):

    s= np.float64(0.5*s)

    t= np.float64(s + 1.00)


    if t <= 1.00:

       s = 2.0*s
       break
# Gives Double Precision output
print ("Double Precision:", k-1,s)
```

Listing 2: Double Precision

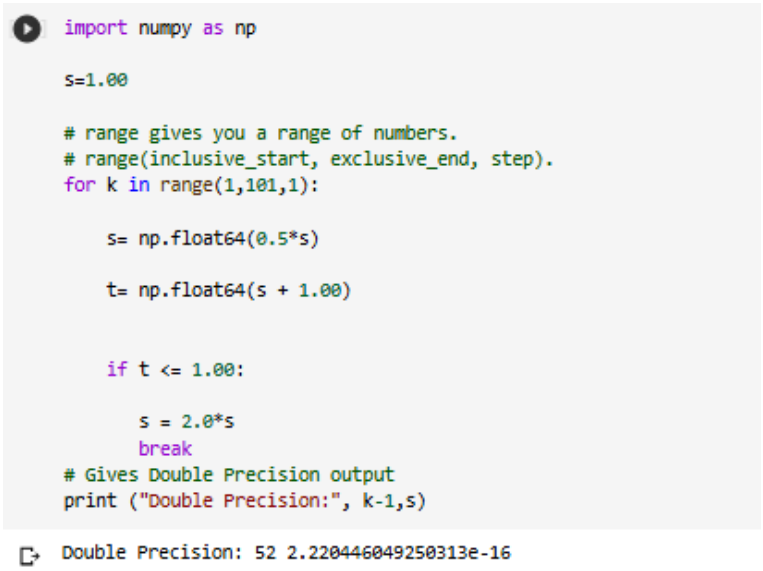


Figure 2: Double Precision

2.2 Problem 2

Evaluate the expression

$$y = y = \sqrt{(x^2 + 1.0)} - 1.0$$

in two ways:

$$y = \sqrt{(x^2 + 1.0)} - 1.0$$

and

$$y = \frac{x^2}{[\sqrt{(x^2 + 1.0)} + 1.0]}$$

for small values of x, x=0.1, 0.01 and 0.001.

Determine the fractional error in both the methods of performing the subtraction. Which method is superior and why?

Answer-

In this problem, we have been asked to evaluate a given function of y in terms of x using two methods, The first being the function itself,and the other a rationalised form, for given values of x.

To begin, I defined x and y as lists. The list for x contains the values for which we have to calculate the values of y using both methods. The list y contains the values of the first function calculated using a high precision online calculator.

For the solution of this problem, I have attempted to use the indexing method to work with the problem in minimum effort possible.

After indexing,I defined the functions which solve y for different methods we will be using.I have used float128 to keep the precision as high as possible.

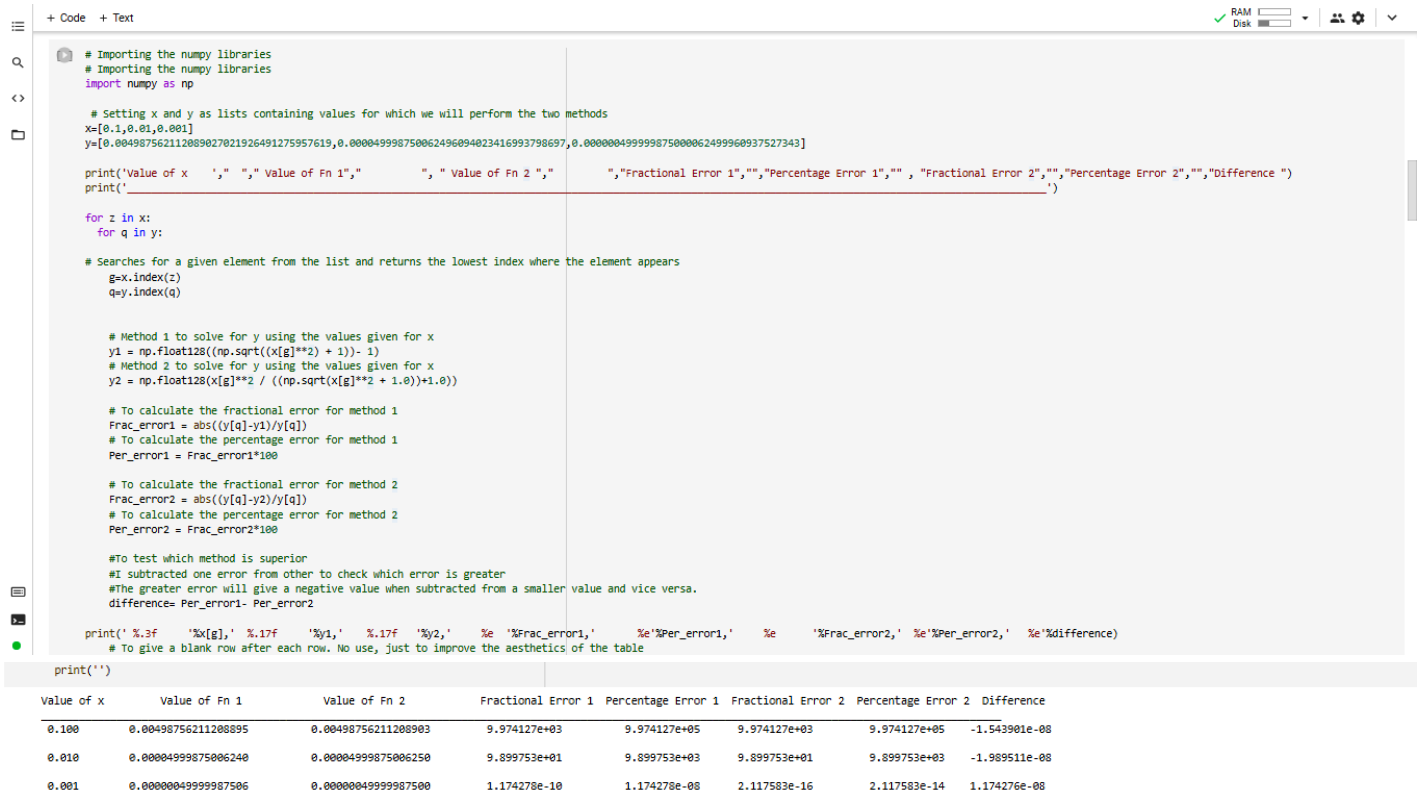
I have calculated the difference between the errors of the two methods to better gauge which method is better. Clearly, as method 1 is superior, the difference between 1 and 2 is negative, implying method 1 shows less error and is more precise.

Then, I use calculated the fractional error and percentage errors for both methods. Lastly, using the print commands, I tried to get the output in the form of table for better presentation and readability.

Both the code used, as well as the I/O screenshots have been attached in the following pages.

```
# Importing the numpy libraries
# Importing the numpy libraries
import numpy as np

 # Setting x and y as lists containing values for which we will perform the two methods
x=[0.1,0.01,0.001]
y=[0.004987562112089027021926491275957619,0.000049998750062496094023416993798697,
0.000000499999875000062499960937527343]

print('Value of x    ',"  "," Value of Fn 1","          ", " Value of Fn 2 ","         ","
    Fractional Error 1","","Percentage Error 1","" , "Fractional Error 2","","Percentage Error
     2","","Difference ")
print('
    ------------------------------------------------------------------------------------------
------------------------------------------------------------')

for z in x:
  for q in y:

# Searches for a given element from the list and returns the lowest index where the element
    appears
    g=x.index(z)
    q=y.index(q)


    # Method 1 to solve for y using the values given for x
    y1 = np.float128((np.sqrt((x[g]**2) + 1))- 1)
    # Method 2 to solve for y using the values given for x
    y2 = np.float128(x[g]**2 / ((np.sqrt(x[g]**2 + 1.0))+1.0))

    # To calculate the fractional error for method 1
    Frac_error1 = abs((y[q]-y1)/y[q])
    # To calculate the percentage error for method 1
    Per_error1 = Frac_error1*100

    # To calculate the fractional error for method 2
    Frac_error2 = abs((y[q]-y2)/y[q])
    # To calculate the percentage error for method 2
    Per_error2 = Frac_error2*100

    #To test which method is superior
    #I subtracted one error from other to check which error is greater
    #The greater error will give a negative value when subtracted from a smaller value and
    vice versa.
    difference= Per_error1- Per_error2

print(' %.3f     '%x[g],'  %.17f     '%y1,'    %.17f    '%y2,'     %e  '%Frac_error1,'       %e
    '%Per_error1,'     %e      '%Frac_error2,'  %e'%Per_error2,'   %e'%difference)
    # To give a blank row after each row. No use, just to improve the aesthetics of the table
  print('')
```

Listing 3: Problem 2



```
# Importing the numpy libraries
# Importing the numpy libraries
import numpy as np

 # Setting x and y as lists containing values for which we will perform the two methods
x=[0.1,0.01,0.001]
y=[0.004987562112089027021926491275957619,0.000049998750062496094023416993798697,0.000000499999875000062499960937527343]

print('Value of x    ',"  "," Value of Fn 1","          ", " Value of Fn 2 ","         ","Fractional Error 1","","Percentage Error 1","" , "Fractional Error 2","","Percentage Error 2","","Difference ")
print('_____________________________________________________________________________________________________________________________________________________')

for z in x:
  for q in y:

# Searches for a given element from the list and returns the lowest index where the element appears
    g=x.index(z)
    q=y.index(q)


    # Method 1 to solve for y using the values given for x
    y1 = np.float128((np.sqrt((x[g]**2) + 1))- 1)
    # Method 2 to solve for y using the values given for x
    y2 = np.float128(x[g]**2 / ((np.sqrt(x[g]**2 + 1.0))+1.0))

    # To calculate the fractional error for method 1
    Frac_error1 = abs((y[q]-y1)/y[q])
    # To calculate the percentage error for method 1
    Per_error1 = Frac_error1*100

    # To calculate the fractional error for method 2
    Frac_error2 = abs((y[q]-y2)/y[q])
    # To calculate the percentage error for method 2
    Per_error2 = Frac_error2*100

    #To test which method is superior
    #I subtracted one error from other to check which error is greater
    #The greater error will give a negative value when subtracted from a smaller value and vice versa.
    difference= Per_error1- Per_error2

print(' %.3f     '%x[g],'  %.17f     '%y1,'    %.17f   '%y2,'     %e  '%Frac_error1,'       %e'%Per_error1,'     %e      '%Frac_error2,'  %e'%Per_error2,'   %e'%difference)
    # To give a blank row after each row. No use, just to improve the aesthetics of the table
print('')
```

Value of x	Value of Fn 1	Value of Fn 2	Fractional Error 1	Percentage Error 1	Fractional Error 2	Percentage Error 2	Difference
0.100	0.00498756211208895	0.00498756211208903	9.974127e+03	9.974127e+05	9.974127e+03	9.974127e+05	-1.543901e-08
0.010	0.00004999875006240	0.00004999875006250	9.899753e+01	9.899753e+03	9.899753e+01	9.899753e+03	-1.989511e-08
0.001	0.00000049999987506	0.00000049999987500	1.174278e-10	1.174278e-08	2.117583e-16	2.117583e-14	1.174276e-08

Figure 3: Problem 2

2.3 Problem 3

It is desired to calculate all integral powers of the number

$$x = \sqrt{(5.0)} - 1.0)/2.0$$

It turns out that the integral powers of x satisfy a simple recursive relation

$$x^{n+1} = x^{n+1} - x^n$$

Show that the above recurrence relation is unstable by calculating

$$x^{16}, x^{30}, x^{40} and x^{50}$$

from the recurrence relation and comparing with the actual values.

Answer-

In this problem, we have been asked to calculate the given integral powers of the given number. We have to check whether the integral powers of x satisfy the given recursive relation and then to check whether the function is stable or unstable for the given integral powers. We also have to compare the values with the actual values and verify our findings.

To begin, I defined the value of x as given in the problem and defined the powers of x to be calculated as a list. Further, I defined the functions f1 and f2 which calculate the values of given integral powers of x and calculated the difference between the two. Lastly, I again try to make tables of the output I received for better presentation purposes.

Both the code used, as well as the I/O screenshots have been attached in the following pages.

```
import numpy as np

#defining the value of x
x  = (np.sqrt(5.0)-1.0)/2.0

#defining an array of powers of x for which we have to check the function
y = np.array([16,30,40,50])

print('Power of x'," ","LHS= Value from f1","    ", "RHS= Value from f2 "," ", "Difference")
print('-------------------------------------------------------------------------------')

#defining functions f1 and f2 which take values of power from y
for i in range(0,4):

  def f1(a,b):
      p = np.float64(a**b)
      return p
  def f2(a,b):
      q = np.float64(a**(b-2) - (a**(b-1)))
      return q

  #defining the two sides of the equation as LHS and RHS
  LHS=f1(x,y[i])
  RHS=f2(x,y[i])
  #calculating the difference between the values of LHS and RHS
  diff= np.float64(abs(f1(x,y[i])-f2(x,y[i])))

  print('    %d       '%y[i],'   %e  '%LHS,'        %e'%RHS,'       %e      '%diff)
```

Listing 4: Problem 3

```python
import numpy as np

#defining the value of x
x  = (np.sqrt(5.0)-1.0)/2.0

#defining an array of powers of x for which we have to check the function
y = np.array([16,30,40,50])


print('Power of x'," ","LHS= Value from f1","    ", "RHS= Value from f2 "," ", "Difference")
print('_____________________________________________________________________________')

#defining functions f1 and f2 which take values of power from y
for i in range(0,4):

  def f1(a,b):
      p = np.float64(a**b)
      return p
  def f2(a,b):
      q = np.float64(a**(b-2) - (a**(b-1)))
      return q

  #defining the two sides of the equation as LHS and RHS
  LHS=f1(x,y[i])
  RHS=f2(x,y[i])
  #calculating the difference betwwen the values of LHS and RHS
  diff= np.float64(abs(f1(x,y[i])-f2(x,y[i])))

  print('    %d       '%y[i],'   %e  '%LHS,'        %e'%RHS,'       %e      '%diff)
```

```
Power of x    LHS= Value from f1      RHS= Value from f2     Difference
_____________________________________________________________________________
    16          4.531039e-04            4.531039e-04        1.084202e-19
    30          5.374905e-07            5.374905e-07        2.117582e-22
    40          4.370130e-09            4.370130e-09        2.481542e-24
    50          3.553186e-11            3.553186e-11        1.938705e-26
```

Figure 4: Problem 3

2.4 Problem 4

The recurrence relation

$$y_{n+1} = e - (n+1)y_n$$

(where e is the base of natural logarithm) can be obtained from integration by parts to the integral

$$y_n = \int_0^1 x^n e^x \, dx$$

Show that the above recurrence relation is unstable by calculating y_{15} and $y_{20}$from the recurrence relation.

Answer-

The given problem asks us to calculate the value of the recurrence relation for given values by two methods, The first being the recurrence relation itself and the second being the method of integration by parts of the given equation. I have attempted to solve the given problem by calculating the values of y for n=15,20 using both methods and then comparing the two to get a better understanding on how the function, as well as the system I used for these computations behave.

Lastly, using the print commands, I tried to get the output in the form of table for better presentation and readability.

Both the code used, as well as the I/O screenshots have been attached in the following pages.

```
1
2  from scipy.integrate import quad
3  import numpy as np
4
5  #defining the recurrence function
6  def R(N,n):
7      return (np.e - (n+1)*N)
8
9  #defining the function to be integrated
10 def f(x,n):
11     return (x**n) * np.exp(x)
12 #defining the integration function with its limits
13 def I(n):
14     return quad(f, 0, 1, args=n)
15
16 n=np.array([15,20])
17
18 # creating a list of calculated values using recurrence relation
19 N = np.float64(I(1)[0])
20 rec = [0.0,np.float64(N)]
21 print("Y_n","    ","Value from recurrence relation")
22 for m in range(1,22,1):
23     N = np.float64(R(N,m))
24     rec.append(N)
25     #all values calculated using recurrence relation from y_2 to y_22
26     print("y_%d "%(m+1),'        ',N)
27
28 print('')
29 print('Value of n','  ','Using numerical integration','  ','Using recurrence relation','    ',
       'Fractional Error','       ','Estimated error')
30 print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
31 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
32 # comparing both methods for functions given in question
33 for k in range(len(n)):
34     print('  ',n[k],'           ',I(n[k])[0],'         ',rec[n[k]],'     ',(abs(I(n[k])[0] -
       rec[n[k]])/I(n[k])[0]),'    ',I(n[k])[1])
35     print('')
```

Listing 5: Problem 4

```python
from scipy.integrate import quad
import numpy as np

#defining the recurrence function
def R(N,n):
    return (np.e - (n+1)*N)

#defining the function to be integrated
def f(x,n):
    return (x**n) * np.exp(x)
#defining the integration function with its limits
def I(n):
    return quad(f, 0, 1, args=n)

n=np.array([15,20])

# creating a list of calculated values using recurrence relation
N = np.float64(I(1)[0])
rec = [0.0,np.float64(N)]
print("Y_n","     ","Value from recurrence relation")
for m in range(1,22,1):
    N = np.float64(R(N,m))
    rec.append(N)
    #all values calculated using recurrence relation from y_2 to y_22
    print("y_%d "%(m+1),'        ',N)

print('')

print('Value of n','  ','Using numerical integration','  ','Using recurrence relation','    ','Fractional Error','       ','Estimated error')
print('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')
```

```python
# comparing both methods for functions given in question
for k in range(len(n)):
    print('  ',n[k],'           ',I(n[k])[0],'         ',rec[n[k]],'     ',(abs(I(n[k])[0] - rec[n[k]])/I(n[k])[0]),'    ',I(n[k])[1])
    print('')
```

```
Y_n      Value from recurrence relation
y_2          0.7182818284590451
y_3          0.5634363430819098
y_4          0.4645364561314058
y_5          0.395599547802016
y_6          0.34468454164694906
y_7          0.30549003693040166
y_8          0.27436153301583177
y_9          0.24902803131655915
y_10          0.22800151529345358
y_11          0.21026516023105568
y_12          0.19509990568637692
y_13          0.18198305453614516
y_14          0.17051906495301283
y_15          0.1604958541638526
y_16          0.15034816183740363
y_17          0.16236307722318344
y_18          -0.2042535615582568
y_19          6.599099498065924
y_20          -129.26370813285942
y_21          2717.256152618507
y_22          -59776.91707577869

Value of n    Using numerical integration    Using recurrence relation      Fractional Error         Estimated error
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
   15            0.16042630893253407          0.1604958541638526       0.0004335026578949139      1.7810898193252774e-15

   20            0.12380383076256998          -129.26370813285942       1045.101037396495      1.6808102031436923e-11
```

Figure 5: Problem 4

2.5 Problem 5

Compute the dot product of the following two vectors

$$x = [2.718281823, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]$$

and

$$y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]$$

Compute the summation in four ways:

1. forward order summation $\{x_i y_i\}$for $i = 1, n$.

2. reverse order summation $\{x_i y_i\}$for $i = n, 1$.

3. largest to smallest order (add positive numbers in order from largest to smallest, then add negative numbers in order from smallest to largest and then add the two partial sums).

4. smallest to largest order (reverse order of adding in the previous method).

Use both single and double precision for a total of eight answers. Compare the results with the correct value $1.006571x10^{-9}$.

Answer-

In this problem we were given two vectors and were tasked with computing the dot product of the two in four different methods. While the first two methods involves simple dot products, the third and fourth were quite challenging to make a clear outline of what we were supposed to do. Initially, I had assumed that the problem had two parts, one of calculation of dot product, and the other of summation of the two vectors in four different ways each for single and double precision.
After reading the statements again and again, I concluded that the first two methods involved calculation and summation of the vectors in the forward and reverse orders respectively.
The third and fourth methods were really tricky. The third method actually asked us to separate the terms according to their signs, then arrange the positives from largest to smallest, the negatives in smallest to largest, calculate the dot products and finally summing to get the final values.
The fourth method used the same approach, just with reverse summation.
Then, for double precision, I just replaced float 32 by float64 .The code for each of the four methods is really large and thus to make it easier, I have divided the whole code into four separate parts, one for each method.Lastly, using the print commands, I tried to get the output in the form of table for better presentation and readability.
This question was the hardest for me to attempt and required a lot of help, suggestions and a bit of copying as well. I even used the help of my younger brother as well as my classmates to guide me in solving this problem. I know the answer I came up with may be wrong, but in the process I learnt a lot of things from lots of places, and honestly , It was quite a lot of fun and equally exciting when I got an output, whether correct or not.

Both the code used, as well as the I/O screenshots have been attached in the following pages.

```
import numpy as np

#defining the two vectors
x = [2.718281823, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]
y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]

#given correct value for dot product
correct_val = 1.006571 * 10**(-9)
#defining empty lists to store the values
list1 = []
list2 = []

dot_32 = 0
dot_64 = 0
#calculation of products
#prod_32 is for single precision
#prod_64 for double precision
for i in range(len(x)):

    prod_32 = np.float32(x[i]*y[i])
    list1.append(prod_32)

    prod_64 = np.float64(x[i]*y[i])
    list2.append(prod_64)

#summation of dot products of the vectors with single and double precision methods
#dot_32 is for single precision
#dot_64 for double precision
for i in range(len(list1)):
  dot_32 = dot_32 +list1[i]
  dot_64 = dot_64 +list2[i]
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100

#getting the output for forward summation
print('''Dot Product in Forward order:
-----------------------------
For Single Precision: {}                                          = {}
For Double Precision: {}  = {}

% Error in Single precision method = {}
% Error in Double precision method = {}
'''.format(list1, dot_32, list2, dot_64, e32, e64))
```

Listing 6: Problem 5(a) Forward summation

Code for forward order summation-

```
import numpy as np

#defining the two vectors
x = [2.718281823, -3.141592654, 1.414213562, 0.5772156649, 0.3010299957]
y = [1486.2497, 878366.9879, -22.37492, 4773714.647, 0.000185049]

#given correct value for dot product
correct_val = 1.006571 * 10**(-9)
#defining empty lists to store the values
list1 = []
list2 = []

dot_32 = 0
dot_64 = 0
#calculation of products
#prod_32 is for single precision
#prod_64 for double precision
for i in range(len(x)):

    prod_32 = np.float32(x[i]*y[i])
    list1.append(prod_32)

    prod_64 = np.float64(x[i]*y[i])
    list2.append(prod_64)

#summation of dot products of the vectors with single and double precision methods
#dot 32 is for single precision
#dot_64 for double precision
for i in range(len(list1)):
  dot_32 = dot_32 +list1[i]
  dot_64 = dot_64 +list2[i]
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100

#getting the output for forward summation
print('''Dot Product in Forward order:
----------------------------
For Single Precision: {}                                          = {}
For Double Precision: {}  = {}

% Error in Single precision method = {}
% Error in Double precision method = {}
'''.format(list1, dot_32, list2, dot_64, e32, e64))
```

```
Dot Product in Forward order:
----------------------------
For Single Precision: [4040.0457, -2759471.2, -31.642916, 2755462.8, 5.57053e-05]                                  = -0.09720572353398893
For Double Precision: [4040.045543949203, -2759471.2767027467, -31.64291531266504, 2755462.8740109736, 5.57052996742893e-05]  = -7.431385965330528e-06

% Error in Single precision method = 9657115547.791454
% Error in Double precision method = 738387.3106150016
```

Figure 6: Forward summation

```
list1.clear()
list2.clear()

dot_32 = 0
dot_64 = 0

#calculation of products
#prod_32 is for single precision
#prod_64 for double precision
for i in range(0, len(x)):
  j = (len(x) - 1) - i

  prod_32 = np.float32(x[j] * y[j])
  list1.append(prod_32)

  prod_64 = np.float64(x[j]*y[j])
  list2.append(prod_64)
#summation of dot products of the vectors with single and double precision methods
for i in range(0, len(x)):
  j = (len(x) - 1) - i
#dot_32 is for single precision
#dot_64 for double precision
  dot_32 = dot_32 +list1[j]
  dot_64 = dot_64 +list2[j]
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100


#getting the output
print('''Dot Product in Reverse order:
----------------------------
For Single Precision: {}                                          = {}
For Double Precision: {}  = {}

% Error in Single precision method = {}
% Error in Double precision method = {}
'''.format(list1, dot_32, list2, dot_64, e32, e64))
```

Listing 7: Problem 5(b)Reverse summation

Code for reverse order summation-

```
list1.clear()
list2.clear()

dot_32 = 0
dot_64 = 0

#calculation of products
#prod_32 is for single precision
#prod_64 for double precision
for i in range(0, len(x)):
  j = (len(x) - 1) - i

  prod_32 = np.float32(x[j] * y[j])
  list1.append(prod_32)

  prod_64 = np.float64(x[j]*y[j])
  list2.append(prod_64)
#summation of dot products of the vectors with single and double precision methods
for i in range(0, len(x)):
  j = (len(x) - 1) - i
#dot_32 is for single precision
#dot_64 for double precision
  dot_32 = dot_32 +list1[j]
  dot_64 = dot_64 +list2[j]
#calculation of error compared to correct value
#e32 is for single precision
#dot_64 for double precision
  dot_32 = dot_32 +list1[j]
  dot_64 = dot_64 +list2[j]
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100


#getting the output
print('''Dot Product in Reverse order:
-----------------------------
For Single Precision: {}                                          = {}
For Double Precision: {}  = {}

% Error in Single precision method = {}
% Error in Double precision method = {}
'''.format(list1, dot_32, list2, dot_64, e32, e64))
```

```
Dot Product in Reverse order:
-----------------------------
For Single Precision: [5.57053e-05, 2755462.8, -31.642916, -2759471.2, 4040.0457]                                          = -0.09720572353398893
For Double Precision: [5.57052996742893e-05, 2755462.8740109736, -31.64291531266504, -2759471.2767027467, 4040.045543949203]  = -7.431385965330528e-06

% Error in Single precision method = 9657115547.791454
% Error in Double precision method = 738387.3106150016
```

Figure 7: Reverse summation

```

list1.sort(reverse=True)
list2.sort(reverse=True)

pos_list1 = []
neg_list1 = []

pos_list2 = []
neg_list2 = []
#pspos_32 means partial sum of positive terms for single precision
#psneg_32 means partial sum of negative terms for single precision
#pspos_64 means partial sum of positive terms for double precision
#psneg_64 means partial sum of negative terms for double precision
pspos_32=0
psneg_32=0
pspos_64=0
psneg_64=0

dot_32 = 0
dot_64 = 0
#summation of dot products of the vectors with single precision
for i in range(0, len(list1)):
  if list1[i] > 0:
    pos_list1.append(list1[i])
    pspos_32 = pspos_32 + list1[i]
  else:
    neg_list1.append(list1[i])
    psneg_32 = psneg_32 + list1[i]
#summation of dot products of the vectors with double precision
for i in range(0, len(list2)):
  if list2[i] > 0:
    pos_list2.append(list2[i])
    pspos_64 = pspos_64 + list2[i]
  else:
    neg_list2.append(list2[i])
    psneg_64 = psneg_64 + list2[i]
#final value of dot products of the vectors with single and double precision methods
dot_32 = pspos_32 + psneg_32
dot_64 = pspos_64 + psneg_64
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100
#getting the output
print('''Dot Product in Largest to smallest order:
----------------------------------------
For Single Precision:
---------------------
    List of Positive terms: {}    = {}
    List of Negative terms: {}               = {}

    Dot Product = {}
    % Error= {}

For Double Precision:
---------------------
    List of Positive terms: {} = {}
    List of Negative terms: {}                     = {}

    Dot Product = {}
    % Error= {}
'''.format(pos_list1,pspos_32, neg_list1,psneg_32,dot_32, e32, pos_list2, pspos_64, neg_list2,
     psneg_64, dot_64, e64))
```

Listing 8: Problem 5(c)Largest to smallest order

Code for largest to smallest order(add positive numbers in order from largest to smallest, then add negative numbers in order from smallest to largest and then add the two partial sums).

```
list1.sort(reverse=True)
list2.sort(reverse=True)

pos_list1 = []
neg_list1 = []

pos_list2 = []
neg_list2 = []
#pspos_32 means partial sum of positive terms for single precision
#psneg_32 means partial sum of negative terms for single precision
#pspos_64 means partial sum of positive terms for double precision
#psneg_64 means partial sum of negative terms for double precision
pspos_32=0
psneg_32=0
pspos_64=0
psneg_64=0

dot_32 = 0
dot_64 = 0
#summation of dot products of the vectors with single precision
for i in range(0, len(list1)):
  if list1[i] > 0:
    pos_list1.append(list1[i])
    pspos_32 = pspos_32 + list1[i]
  else:
    neg_list1.append(list1[i])
    psneg_32 = psneg_32 + list1[i]
#summation of dot products of the vectors with double precision
for i in range(0, len(list2)):
  if list2[i] > 0:
    pos_list2.append(list2[i])
    pspos_64 = pspos_64 + list2[i]
  else:
    neg_list2.append(list2[i])
    psneg_64 = psneg_64 + list2[i]
#final value of dot products of the vectors with single and double precision methods
dot_32 = pspos_32 + psneg_32
dot_64 = pspos_64 + psneg_64
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100
#getting the output
print('''Dot Product in Largest to smallest order:
-----------------------------------------
For Single Precision:
---------------------
    List of Positive terms: {}    = {}
    List of Negative terms: {}               = {}

    Dot Product = {}
    % Error= {}

For Double Precision:
---------------------
    List of Positive terms: {} = {}
    List of Negative terms: {}                    = {}

    Dot Product = {}
    % Error= {}
'''.format(pos_list1,pspos_32, neg_list1,psneg_32,dot_32, e32, pos_list2, pspos_64, neg_list2, psneg_64, dot_64, e64))
```

```
Dot Product in Largest to smallest order:
-----------------------------------------
For Single Precision:
---------------------
    List of Positive terms: [2755462.8, 4040.0457, 5.57053e-05]    = 2759502.795710002
    List of Negative terms: [-31.642916, -2759471.2]               = -2759502.8929157257

    Dot Product = -0.09720572363585234
    % Error= 9657115557.911297

For Double Precision:
---------------------
    List of Positive terms: [2755462.8740109736, 4040.045543949203, 5.57052996742893e-05] = 2759502.919610628
    List of Negative terms: [-31.642915331266504, -2759471.2767027467]                     = -2759502.9196180594

    Dot Product = -7.431488484144211e-06
    % Error= 738397.4955710238
```

Figure 8: largest to smallest order

```
list1.sort(reverse=False)
list2.sort(reverse=False)

pos_list1 = []
neg_list1 = []

pos_list2 = []
neg_list2 = []
#pspos_32 means partial sum of positive terms for single precision
#psneg_32 means partial sum of negative terms for single precision
#pspos_64 means partial sum of positive terms for double precision
#psneg_64 means partial sum of negative terms for double precision
pspos_32=0
psneg_32=0
pspos_64=0
psneg_64=0

dot_32 = 0
dot_64 = 0
#summation of dot products of the vectors with single precision
for i in range(0, len(list1)):
  if list1[i] > 0:
    pos_list1.append(list1[i])
    pspos_32 = pspos_32 + list1[i]
  else:
    neg_list1.append(list1[i])
    psneg_32 = psneg_32 + list1[i]
#summation of dot products of the vectors with double precision
for i in range(0, len(list2)):
  if list2[i] > 0:
    pos_list2.append(list2[i])
    pspos_64 = pspos_64 + list2[i]
  else:
    neg_list2.append(list2[i])
    psneg_64 = psneg_64 + list2[i]
#final value of dot products of the vectors with single and double precision methods
dot_32 = pspos_32 + psneg_32
dot_64 = pspos_64 + psneg_64
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100
#getting the output
print('''Dot Product in Smallest to largest order:
----------------------------------------
For Single Precision:
---------------------
    List of Positive terms: {}    = {}
    List of Negative terms: {}               = {}

    Dot Product = {}
    % Error = {}

For Double Precision:
---------------------
    List of Positive terms: {} = {}
    List of Negative terms: {}                    = {}

    Dot Product = {}
    % Error= {}
'''.format(pos_list1,pspos_32, neg_list1,psneg_32,dot_32, e32, pos_list2, pspos_64, neg_list2,
     psneg_64, dot_64, e64))
```

Listing 9: Problem 5(d)smallest to largest order

Code for smallest to largest order (reverse order of adding in the previous method).

```
list1.sort(reverse=False)
list2.sort(reverse=False)

pos_list1 = []
neg_list1 = []

pos_list2 = []
neg_list2 = []
#pspos_32 means partial sum of positive terms for single precision
#psneg_32 means partial sum of negative terms for single precision
#pspos_64 means partial sum of positive terms for double precision
#psneg_64 means partial sum of negative terms for double precision
pspos_32=0
psneg_32=0
pspos_64=0
psneg_64=0

dot_32 = 0
dot_64 = 0
#summation of dot products of the vectors with single precision
for i in range(0, len(list1)):
  if list1[i] > 0:
    pos_list1.append(list1[i])
    pspos_32 = pspos_32 + list1[i]
  else:
    neg_list1.append(list1[i])
    psneg_32 = psneg_32 + list1[i]
```

```
#summation of dot products of the vectors with double precision
for i in range(0, len(list2)):
  if list2[i] > 0:
    pos_list2.append(list2[i])
    pspos_64 = pspos_64 + list2[i]
  else:
    neg_list2.append(list2[i])
    psneg_64 = psneg_64 + list2[i]
#final value of dot products of the vectors with single and double precision methods
dot_32 = pspos_32 + psneg_32
dot_64 = pspos_64 + psneg_64
#calculation of error compared to correct value
#e32 is for single precision
#e64 for double precision
e32 = (abs(correct_val - dot_32)/correct_val)*100
e64 = (abs(correct_val - dot_64)/correct_val)*100
#getting the output
print('''Dot Product in Smallest to largest order:
----------------------------------------
For Single Precision:
--------------------
    List of Positive terms: {}    = {}
    List of Negative terms: {}              = {}

    Dot Product = {}
    % Error = {}

For Double Precision:
--------------------
    List of Positive terms: {} = {}
    List of Negative terms: {}                    = {}

    Dot Product = {}
    % Error= {}
'''.format(pos_list1,pspos_32, neg_list1,psneg_32,dot_32, e32, pos_list2, pspos_64, neg_list2, psneg_64, dot_64, e64))
```

```
Dot Product in Smallest to largest order:
----------------------------------------
For Single Precision:
--------------------
    List of Positive terms: [5.57053e-05, 4040.0457, 2755462.8]    = 2759502.795710002
    List of Negative terms: [-2759471.2, -31.642916]               = -2759502.8929157257

    Dot Product = -0.09720572363585234
    % Error = 9657115557.911297

For Double Precision:
--------------------
    List of Positive terms: [5.57052996742893e-05, 4040.045543949203, 2755462.8740109736] = 2759502.919610628
    List of Negative terms: [-2759471.2767027467, -31.64291531266504]                     = -2759502.9196180594

    Dot Product = -7.431488484144211e-06
    % Error= 738397.4955710238
```

Figure 9: smallest to largest order

3 Bibliography

During the process of making this assignment I have taken help from various websites to solve various issues I faced. Few of the same are listed below:

- https://www.pythoncheatsheet.org/
- https://www.tutorialspoint.com/index.htm
- https://www.stackoverflow.com/
- https://www.github.com/
- https://www.javatpoint.com/python-tutorial
- https://www.programiz.com/python-programming