EMORY UNIVERSITY SCHOOL OF MEDICINE

Department of Biomedical Informatics

BMI 500:

Introduction to Biomedical Informatics

Lecture 2. Coding, documentation, security
& data management basics

Homework- Week 2

Python Tutorial Assessment

Seyedeh Somayyeh Mousavi

Problem 1.

Write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula.

Solution:

Using a loop for the calculation of every term and adding it to the sum of other previous terms N as input determines an integer

In [3]:
```
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       sum= 0
       for i in range(0,N,1):

          sum += (-1)**i / (2*i +1)

       return sum

    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [4]:
```
N = 6
LeibnizFormula(N)
```

Out[4]:
```
0.7440115440115441
```

Problem 2.a

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula. a. Use a for-loop and an if-statement with the modulo operator % to determine whether to add or subtract each term.

Solution:

Using a loop for the calculation of every term and Using the operator % to add or subtract. N as input determines an integer.

In [1]:
```
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       sum= 0
       for i in range (0,N,1):

          # determination to add or subscribe
          if ((-1)**i % (2*i +1))>= 0 :

              sum+= (-1)**i / (2*i +1)

          else:
              sum-= (-1)**i / (2*i +1)

       return sum
    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [8]:
```
N=6
print(LeibnizFormula(N))
```

```
0.7440115440115441
```

Problem 2.b

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula.

Use a for-loop with the quantity (-1)**n to determine whether to add or subtract each term.

Solution:

Using a loop for the calculation of every term and using the the quantity (-1)**n to add or subtract. N as input determines an integer.

In [1]:
```
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       sum= 0
       for i in range(0,N,1):

          # determination to add or subscribe
          term = 1 / (2*i +1)
          if (-1)**i > 0:
              sum+= term

          else:
              sum-= term

       return sum
    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [3]:
```
N =  110000
print(LeibnizFormula(N))
```

```
0.7853958906701609
```

Problem 2.c

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula.

c. Construct a Python list and compute the sum of the terms in the list.

Solution:

Using a loop for the calculation of every term and adding to a list. Then, all of them sum. N as input determines an integer.

In [3]:
```
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       Term_list = []
       for i in range(0,N,1):

          # calculation of every term and adding to a list
          Term_list.append ((-1)**i / (2*i +1))

       return sum (Term_list)
    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [4]:
```
N = 10000
print(LeibnizFormula(N))
```

```
0.7853731633975086
```

Problem 2.d

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula. d. Construct a Python set and compute the sum of the terms in the set.

Solution:

Using a loop for the calculation of every term and adding to a set. Then, all of them sum. N as input determines an integer.

In [1]:
```
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       Term_set = set()
       for i in range(0,N,1):

          # calculation of every term and adding to a set
          Term_set.add ((-1)**i / (2*i +1))

       return sum (Term_set)
    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [2]:
```
N = 6
print(LeibnizFormula(N))
```

```
0.7440115440115441
```

Problem 2.e

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula.

e. Construct a Python dictionary and compute the sum of the terms in the dictionary.

Solution:

Using a loop for the calculation of every term and adding to a dictionary. Then, all of them sum. N as input determines an integer.

In [8]:
```
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       Term_dictionary = {0:1}
       for i in range(1,N,1):

          # calculation of every term and adding to a dictionary
          Term_dictionary[i] = ((-1)**i / (2*i +1))

       return sum(Term_dictionary.values())

    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [9]:
```
N=6
print(LeibnizFormula(N))
```

```
0.7440115440115441
```

Problem 2.f

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula.

f. Construct a NumPy array and compute the sum of the terms in the array

Solution:

Using a loop for the calculation of every term and adding to a array. Then, all of them sum. N as input determines an integer.

In [1]:
```
import numpy as np
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :

       Term_array = np.zeros (N)
       for i in range(0,N,1):

          Term_array[i] = ((-1)**i / (2*i +1))

       return sum(Term_array[:])

    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [2]:
```
N=5
print(LeibnizFormula(N))
```

```
0.8349206349206351
```

Problem 2.g

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula.

g. Construct a NumPy array, use array indexing to compute the sum of the positive terms in the array, use array indexing to compute the sum of the negative terms in the array, and add the two sums together. You can write x[::2] to access the first, third, etc. terms and x[1::2] to access the second, fourth, etc. terms.

Solution:

Using a loop for the calculation of every term and adding to a array. Then, all of them sum. N as input determines an integer.

In [1]:
```
import numpy as np
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       Term_array = np.zeros (N)

       for i in range(0,N,1):

          Term_array[i] = ((-1)**i / (2*i +1))

       Term_Negative = Term_array[1::2]
       Term_Positive = Term_array[::2]

       print ("Term_Positive=", Term_Positive) , print ("Term_Negative=", Term_Negative)
       return (sum (Term_Positive) + sum (Term_Negative))

    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [2]:
```
N = 5
print(LeibnizFormula(N))
```

```
Term_Positive= [1.         0.2        0.11111111]
Term_Negative= [-0.33333333 -0.14285714]
0.834920634920635
```

Problem 2.J.

For each of the following items, write a function in Python 3.x that, given an integer n, returns the sum of the first n terms of the series in the Leibniz formula.

J. Combine the first and second terms, the third and fourth terms, etc. to change this series from an alternating to a non-alternating series and compute the sum of the combined terms.

Solution:

Using a loop for the calculation of every term and adding to an array. Then,Negative and Positive terms are separated. After that, they are combined to make a new array and all of them are sum. N as input determines an integer.

In [5]:
```
import numpy as np
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :
       Term_array = np.zeros (N)
       for i in range(0,N,1):

          Term_array[i] = ((-1)**i / (2*i +1))

       Term_Negative = Term_array[1::2]
       Term_Positive = Term_array[::2]

       if len (Term_Positive) > len (Term_Negative):
           Term_Negative =np.append(Term_Negative,0)

       newTerm_array = Term_Negative + Term_Positive
       print(newTerm_array)
       return (sum(newTerm_array))

    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

In [7]:
```
N = 5
print(LeibnizFormula(N))
```

```
[0.66666667 0.05714286 0.11111111]
0.8349206349206351
```

Problem 3. (Fastest implementations of the Leibniz formula)

Which of these implementations of the Leibniz formula is the most accurate, fastest, and/or clearest? Which function would you use to calculate π (and why)? You can use the. built-in constants math.pi or np.pi for assessing the accuracy of your functions and the time package or the %timeit command for timing.

Solution:

Using functions in the solution of problems 1&2 to calculate the time of the sum first n terms of the series in the Leibniz formula from 1 to 10000.

After running all function and according to results, function is related to NumPy array is chosen as the fastest algorithm. As we know, an array in Python and other programming languages can store multiple values in one variable and allows the programmer to process the data stored in an array in a batch or in a loop.

In [19]:
```
# N: the first number terms of the series in the Leibniz formula.
N=10000
import timeit
```

In [20]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem1
```

```
550 µs ± 24.4 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [21]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2a
```

```
586 µs ± 35.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [22]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2b
```

```
581 µs ± 28.9 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [23]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2c
```

```
593 µs ± 25.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [24]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2d
```

```
570 µs ± 24.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [25]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2e
```

```
616 µs ± 42.3 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [26]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2f
```

```
513 µs ± 20.8 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [27]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2g
```

```
575 µs ± 35 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [28]:
```
%timeit for x in range(1,N,1): LeibnizFormula_problem2j
```

```
547 µs ± 27.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Problem 3. (The most accurate implementations of the Leibniz Formula)

Which of these implementations of the Leibniz formula is the most accurate, fastest, and/or clearest? Which function would you use to calculate π (and why)? You can use the. built-in constants math.pi or np.pi for assessing the accuracy of your functions and the time package or the %timeit command for timing.

Solution:

Using functions in the solution of problems 1&2 to calculate the Root Mean Squared Error of the sum of the first n terms of the series in the Leibniz formula from 1 to N.

After running all function and according to results, they have same accuracy in a same N. It can be predictable because we implement different algorithms of a numeral series that their sum approximately close to pi in a big N.

In [22]:
```
import numpy as np
import sklearn.metrics as met
N=10000
Y= np.pi*np.ones(N)
```

In [23]:
```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem1(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem1=',rmse)
```

```
rmse_LeibnizFormula_problem1= 0.0334931149681905
```

In [24]:
```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2a(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2a=',rmse)
```

```
rmse_LeibnizFormula_problem2a= 0.0334931149681905
```

In [25]:
```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2b(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2b=',rmse)
```

```
rmse_LeibnizFormula_problem2b= 0.0334931149681905
```

In [26]:
```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2c(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2c=',rmse)
```

```
rmse_LeibnizFormula_problem2c= 0.0334931149681905
```

In [27]:

```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2d(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2d=',rmse)
```

```
rmse_LeibnizFormula_problem2d= 0.0334931149681905
```

In [28]:

```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2e(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2e=',rmse)
```

```
rmse_LeibnizFormula_problem2e= 0.0334931149681905
```

In [29]:

```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2f(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2f=',rmse)
```

```
rmse_LeibnizFormula_problem2f= 0.0334931149681905
```

In [30]:

```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2g(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2g=',rmse)
```

```
rmse_LeibnizFormula_problem2g= 0.0334931149681905
```

In [31]:

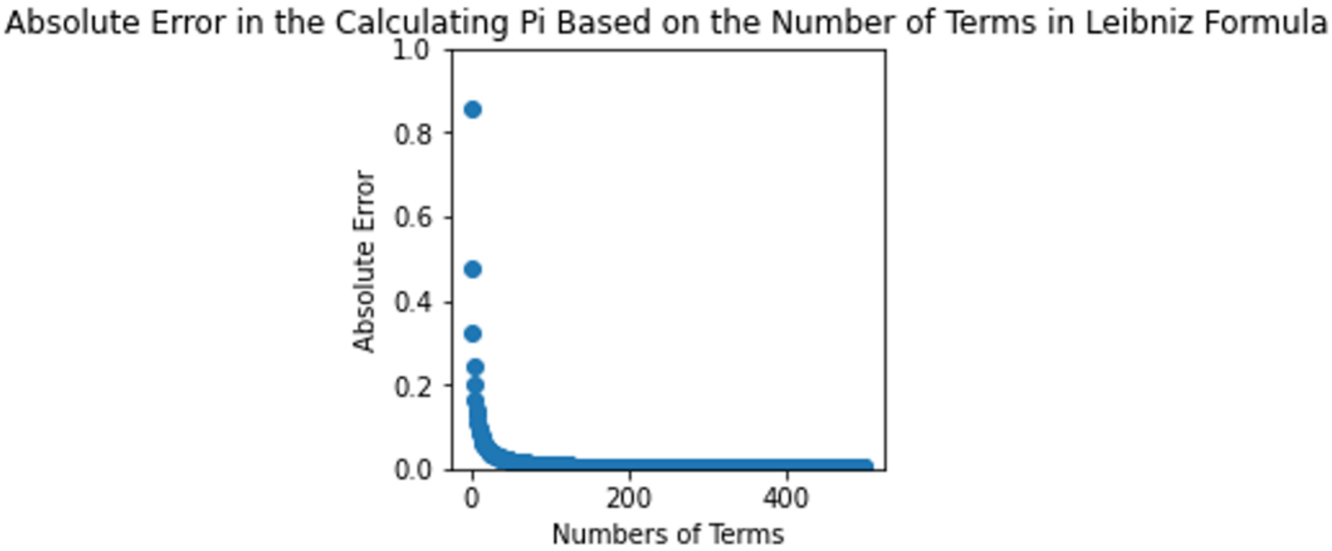
```
Yh = np.zeros (N)
for i in range(1,N,1):Yh[i-1] = LeibnizFormula_problem2j(i)*4
rmse = met.mean_squared_error(Y, Yh)**0.5
print('rmse_LeibnizFormula_problem2j=',rmse)
```

```
rmse_LeibnizFormula_problem2j= 0.0334931149681905
```

Problem 4.

Choose your favorite implementation of this formula, and plot the absolute error in the sum as a function of the number of terms in the sum. You should have accurate, informative, and legible labels for both the x-axis and y-axis, and you should use logarithmic axes or take the logarithm of the errors and the numbers of terms so that you can see the sums converge over a wide range of values. By default, most plotting packages generate large plots that are difficult to read in papers and presentations. Use a command like plt.figure(figsize=(2, 2)) to make your plot more legible.

According to previous session, my favorite function is constructed by array in NumPy.



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

```python
import numpy as np
def LeibnizFormula (N):
    # N must be a positive number
    if N > 0 :

       Term_array = np.zeros (N)
       for i in range(0,N,1):

          Term_array[i] = ((-1)**i / (2*i +1))

       return sum(Term_array[:])

    # If N be a negative number
    else:
        print("IT CAN NOT BE CALCULATED")
```

```python
N=100
Y= np.pi*np.ones(N)
Yh = np.zeros(N)
for i in range(1,N+1,1):Yh[i-1] = LeibnizFormula(i)*4
error=abs(Y-Yh)
```

```python
plt.figure(figsize=(3, 3))
plt.ylim(0,1);
plt.plot(error,'o')
plt.title('Absolute Error in the Calculating Pi Based on the Number of Terms in Leibniz Formula')
plt.xlabel('Numbers of Terms')
plt.ylabel('Absolute Error')
```

Problem 5. Would you do anything differently for computing π using the Leibniz formula, if you were using Matlab instead of Python?

Absolutely there are some other methods for computing pi but there are not limited to MATLAB. As we know python is one of the most favorite languages. In the solving this kind of question, there are not really differently between their ability.