ML Prerequsite-test 23250031

January 6, 2024

$Q1$. How many multiplications and additions do you need to perform a matrix multiplication between a (n, k) and (k, m) matrix? Explain.

Ans. We will need to perform $n * m * k$ multiplications and $n * m * (k - 1)$ additions during a matrix multiplication.

Lets see these matrices:- $\begin{Bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{Bmatrix}$ and $\begin{Bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{Bmatrix}$

Here, we can see that the k rows will multiply with the k columns n times for the first matrix and m times for the second matrix. Hence, it needs $n * m * k$ multiplications, and we can see that while multiplying, the addition needed in one whole multiplication is $(k - 1)$, and it will also be repeated n times for the first matrix and m times for the second matrix. Hence, it needs $n * m * (k - 1)$ additions.

$Q2$. Write Python code to multiply the above two matrices. Solve using list of lists and then use numpy. Compare the timing of both solutions. Which one is faster? Why?

Ans. Code for Matrix Multiplication using list of lists:-

[58]:
```
import time
start_time = time.time()

mat1 = [[1,2,3],[4,5,6]];
mat2 = [[7,8,9],[10,11,12],[13,14,15]];

# print(len(mat2))
res=[]
for n in range(len(mat1)):
    temp = []
    for m in range(len(mat2[0])):
        tempr = 0
        for k in range(len(mat1[0])):
            tempr += mat1[n][k] * mat2[k][m]
        temp.append(tempr)
    res.append(temp)

print("Matrix Multiplication using list of lists:-")
print(res)

print("\nTime Taken:-")
```

```
print("--- %f seconds ---" % (time.time() - start_time))
```

```
Matrix Multiplication using list of lists:-
[[66, 72, 78], [156, 171, 186]]

Time Taken:-
--- 0.001009 seconds ---
```

Code for Matrix Multiplication using Numpy:-

[73]:
```
import time
import numpy as np
start_time = time.time()

mat1 = np.array([[1,2,3],[4,5,6]]);
mat2 = np.array([[7,8,9],[10,11,12],[13,14,15]]);

print("Matrix Multiplication using Numpy:-")
print(np.matmul(mat1, mat2))

print("\nTime Taken:-")
print("--- %f seconds ---" % (time.time() - start_time))
```

```
Matrix Multiplication using Numpy:-
[[ 66  72  78]
 [156 171 186]]

Time Taken:-
--- 0.000998 seconds ---
```

Looking inside the Numpy Configurations:-

[78]:
```
import numpy as np
np.show_config()
```

```
openblas64__info:
    libraries = ['openblas64_', 'openblas64_']
    library_dirs = ['openblas\\lib']
    language = c
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'),
('HAVE_BLAS_ILP64', None)]
    runtime_library_dirs = ['openblas\\lib']
blas_ilp64_opt_info:
    libraries = ['openblas64_', 'openblas64_']
    library_dirs = ['openblas\\lib']
    language = c
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'),
('HAVE_BLAS_ILP64', None)]
    runtime_library_dirs = ['openblas\\lib']
openblas64__lapack_info:
```

```
    libraries = ['openblas64_', 'openblas64_']
    library_dirs = ['openblas\\lib']
    language = c
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'),
('HAVE_BLAS_ILP64', None), ('HAVE_LAPACKE', None)]
    runtime_library_dirs = ['openblas\\lib']
lapack_ilp64_opt_info:
    libraries = ['openblas64_', 'openblas64_']
    library_dirs = ['openblas\\lib']
    language = c
    define_macros = [('HAVE_CBLAS', None), ('BLAS_SYMBOL_SUFFIX', '64_'),
('HAVE_BLAS_ILP64', None), ('HAVE_LAPACKE', None)]
    runtime_library_dirs = ['openblas\\lib']
Supported SIMD extensions in this NumPy install:
    baseline = SSE,SSE2,SSE3
    found = SSSE3,SSE41,POPCNT,SSE42,AVX,F16C,FMA3,AVX2
    not found = AVX512F,AVX512CD,AVX512_SKX,AVX512_CLX,AVX512_CNL,AVX512_ICL
```

The numpy multiplication is faster than a naive list multiplication, and it will be more apparent in matrices with more number of elements. This is because numpy uses an optimised method of matrix multiplication using the BLAS (Basic Linear Algebra Subprograms) library. It basically breaks a bigger matrix multiplication into block multiplications along with parallelisation for utilising the cache and boosting the performance significantly.

*Q*3. Finding the highest element in a list requires one pass of the array. Finding the second highest element requires 2 passes of the array. Using this method, what is the time complexity of finding the median of the array? Can you suggest a better method? Can you implement both these methods in Python and compare against numpy.median routine in terms of time?

Ans. According to the method mentioned in the question, we will have to make at least $\frac{n}{2}$ passes over the array giving a time complexity of $O(n^2)$. We can do better by first sorting the array and then finding the $\frac{n}{2}^{th}$ element in constant time with the help of indexing. Giving us the median in time complexity of $O(nlogn)$.

Code for Median according to question:-

[168]:
```
# import statistics
# print(statistics.median(array))
import time
start_time = time.time()

array = [3,5,12,7,89,4,21,45,54,1,75,53,33]

for i in range(len(array)//2):
    max = array[0]
    max_ind = 0
    for j in range(1, len(array)):
        if array[j] > max:
            max = array[j]
            max_ind = j
```

```
    del array[max_ind]
max = array[0]
for j in range(1, len(array)):
        if array[j] > max:
            max = array[j]

print("Median:-")
print(max)

print("\nTime Taken:-")
print("--- %f seconds ---" % (time.time() - start_time))
```

```
Median:-
21

Time Taken:-
--- 0.001006 seconds ---
```

Code for Median Using Sorting approach:-

[162]:
```
import time
start_time = time.time()

array = [3,5,12,7,89,4,21,45,54,1,75,53,33]

print("Median:-")
print(sorted(array)[(len(array)//2)])

print("\nTime Taken:-")
print("--- %f seconds ---" % (time.time() - start_time))
```

```
Median:-
21

Time Taken:-
--- 0.001002 seconds ---
```

Code for Median Using Numpy:-

[155]:
```
import time
import numpy as np
start_time = time.time()

array = np.array([3,5,12,7,89,4,21,45,54,1,75,53,33])

print("Median:-")
print(np.median(array))
```

```
print("\nTime Taken:-")
print("--- %f seconds ---" % (time.time() - start_time))
```

```
Median:-
21.0

Time Taken:-
--- 0.000998 seconds ---
```

*Q*4. What is the gradient of the following function with respect to x and y?

$$x^2y + y^3sin(x)$$

Ans. Using the formula to calculate gradient:- $\Delta f(x,y) = \frac{\delta f}{\delta x}i + \frac{\delta f}{\delta y}j$

We get,

$$\frac{\delta f}{\delta x} = 2xy + y^3cos(x)$$

$$\frac{\delta f}{\delta y} = x^2 + 3y^2sin(x)$$

$$\Delta f(x,y) = (2xy + y^3cos(x))i + (x^2 + 3y^2sin(x))j$$

*Q*5. Use JAX to confirm the gradient evaluated by your method matches the analytical solution corresponding to a few random values of x and y.

Ans. Using Jax to calculate gradient:-

[51]:
```
from jax import grad
import jax.numpy as jx

def f(x,y):
  return x**2 * y + y**3 * jx.sin(x)

print("Gradient using Jax:-")
dfdx=jax.grad(f,argnums=0)
dfdy=jax.grad(f,argnums=1)

x=3.0
y=6.0

resx=dfdx(x, y)
resy=dfdy(x, y)

print("Radian Values of df/dx and df/dy for 3 and 6:- ", resx," ", resy)
```

```
Gradient using Jax:-
Radian Values of df/dx and df/dy for 3 and 6:-  -177.83838   24.24096
```

*Q*6. Use sympy to confirm that you obtain the same gradient analytically.

Ans. Using Sympy to calculate gradient:-

[9]:
```
from sympy import symbols, diff, sin
```

```
x, y = symbols("X Y")

equation = x**2*y + y**3*sin(x)

dfdx = diff(equation, x)
dfdy= diff(equation, y)

print("Gradient using Sympy:-")
print("(",dfdx,") i"," + (",dfdy,") j")
```

```
Gradient using Sympy:-
( 2*X*Y + Y**3*cos(X) ) i  + ( X**2 + 3*Y**2*sin(X) ) j
```

$Q7$. Create a Python nested dictionary to represent hierarchical information according to question.

[63]:
```
Nested_dict = {"2022": {"branch_1": {"Roll_No": 1, "Name": 'Preyum', "Marks":␣
↪{"Systems": 45, "Algorithms": 52}},

                       "branch_2": {"Roll_No": 2, "Name": 'Dhruv', "Marks":␣
↪{"Algorithms": 56, "Deep Learning": 32}}},

               "2023": {"branch_1": {"Roll_No": 1, "Name": 'Mukul', "Marks":␣
↪{"Systems": 76, "Statistics": 57}},

                       "branch_2": {"Roll_No": 2, "Name": 'Pratyush', "Marks":␣
↪{"Systems": 88, "Deep Learning": 44}}},

               "2024": {"branch_1": {"Roll_No": 1, "Name": 'Mohan', "Marks":␣
↪{"Deep Learning": 23, "Statistics": 20}},

                       "branch_2": {"Roll_No": 2, "Name": 'Devansh', "Marks":␣
↪{"Systems": 56, "Artificial Intelligence": 64}}},

               "2025": {"branch_1": {"Roll_No": 1, "Name": 'Rubait', "Marks":␣
↪{"Digital Cultures": 88, "Machine Learning": 78}},

                       "branch_2": {"Roll_No": 2, "Name": 'Rugved', "Marks":␣
↪{"Combinatorics": 53, "Systems": 82}}}
             }

print(Nested_dict["2022"]["branch_1"])
print(Nested_dict["2022"]["branch_2"])
print(Nested_dict["2023"]["branch_1"])
print(Nested_dict["2023"]["branch_2"])
print(Nested_dict["2024"]["branch_1"])
print(Nested_dict["2024"]["branch_2"])
```

```
{'Roll_No': 1, 'Name': 'Preyum', 'Marks': {'Systems': 45, 'Algorithms': 52}}
```

```
{'Roll_No': 2, 'Name': 'Dhruv', 'Marks': {'Algorithms': 56, 'Deep Learning':
32}}
{'Roll_No': 1, 'Name': 'Mukul', 'Marks': {'Systems': 76, 'Statistics': 57}}
{'Roll_No': 2, 'Name': 'Pratyush', 'Marks': {'Systems': 88, 'Deep Learning':
44}}
{'Roll_No': 1, 'Name': 'Mohan', 'Marks': {'Deep Learning': 23, 'Statistics':
20}}
{'Roll_No': 2, 'Name': 'Devansh', 'Marks': {'Systems': 56, 'Artificial
Intelligence': 64}}
```

*Q*8. Store the same information using Python classes. We have an overall database which is a list of year objects. Each year contains a list of branches. Each branch contains a list of students. Each student has some properties like name, roll number and has marks in some subjects.

[97]:
```python
class year:
    branch = {}
    def add_branch(self, branch_no, roll, name):
        self.branch[branch_no] = {"Roll_No": roll, "Name": name}

    def add_marks(self, branch_no, subject, marks):
        if "Marks" in self.branch[branch_no]:
            self.branch[branch_no]["Marks"][subject] = marks
        else:
            self.branch[branch_no]["Marks"] = {subject: marks}

year_2024 = year()

year_2024.add_branch(1, 232500, "Preyum")
year_2024.add_marks(1, "Stat", 34)
year_2024.add_marks(1, "apti", 54)

year_2024.add_branch(2, 232421, "Dhruv")
year_2024.add_marks(2, "system", 78)
year_2024.add_marks(2, "DL", 32)

year_2024.add_branch(3, 232134, "Pratyush")
year_2024.add_marks(3, "ML", 89)
year_2024.add_marks(3, "Digital Culture", 67)

print("Year 2024 data")
print(year_2024.branch[1])
print(year_2024.branch[2])
print(year_2024.branch[3])

year_2023 = year()

year_2023.add_branch(1, 373772, "Disha")
year_2023.add_marks(1, "Fluid Mechanics", 64)
```

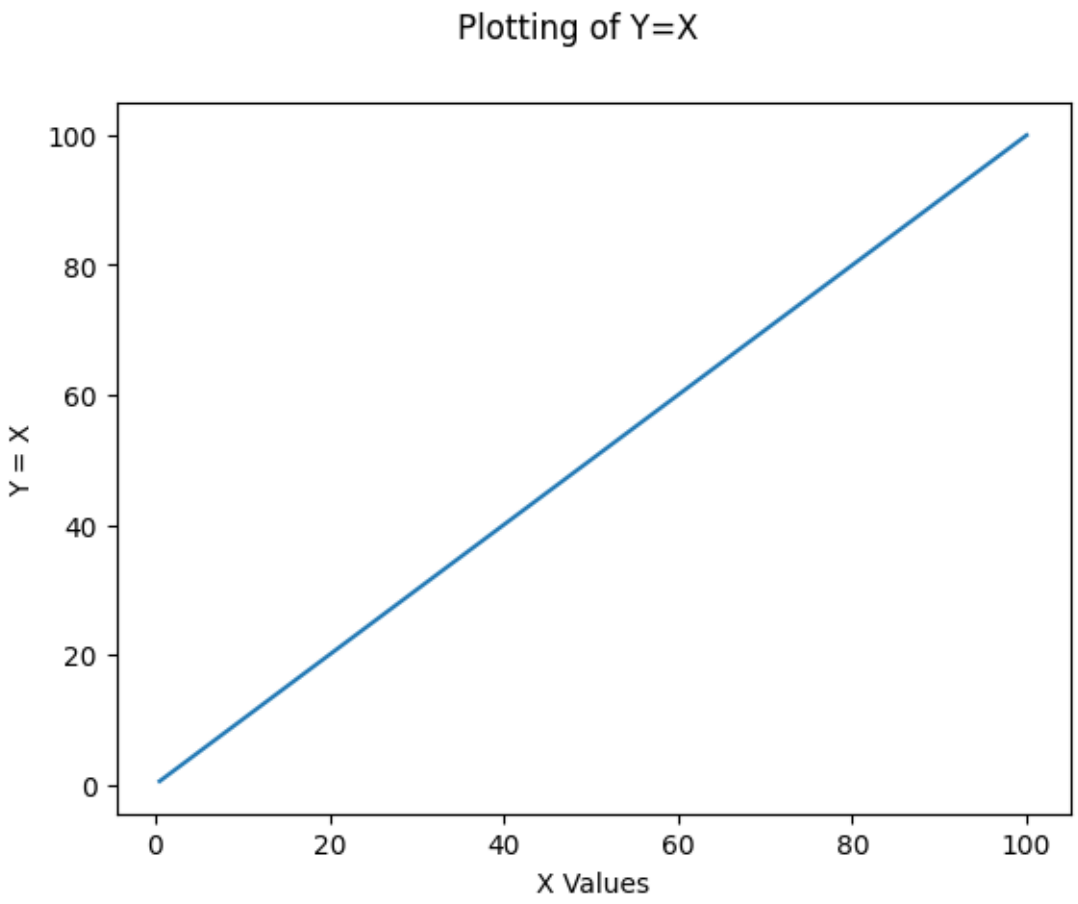
```
year_2023.add_marks(1, "Strength of Meterials", 87)

year_2023.add_branch(2, 654334, "Isha")
year_2023.add_marks(2, "Conitives", 99)
year_2023.add_marks(2, "Human Interactions", 78)

year_2023.add_branch(3, 345673, "Rabina")
year_2023.add_marks(3, "Linguistics", 89)
year_2023.add_marks(3, "Political Science", 56)

print("\nYear 2023 data")
print(year_2023.branch[1])
print(year_2023.branch[2])
print(year_2023.branch[3])
```

```
Year 2024 data
{'Roll_No': 232500, 'Name': 'Preyum', 'Marks': {'Stat': 34, 'apti': 54}}
{'Roll_No': 232421, 'Name': 'Dhruv', 'Marks': {'system': 78, 'DL': 32}}
{'Roll_No': 232134, 'Name': 'Pratyush', 'Marks': {'ML': 89, 'Digital Culture':
67}}

Year 2023 data
{'Roll_No': 373772, 'Name': 'Disha', 'Marks': {'Fluid Mechanics': 64, 'Strength
of Meterials': 87}}
{'Roll_No': 654334, 'Name': 'Isha', 'Marks': {'Conitives': 99, 'Human
Interactions': 78}}
{'Roll_No': 345673, 'Name': 'Rabina', 'Marks': {'Linguistics': 89, 'Political
Science': 56}}
```

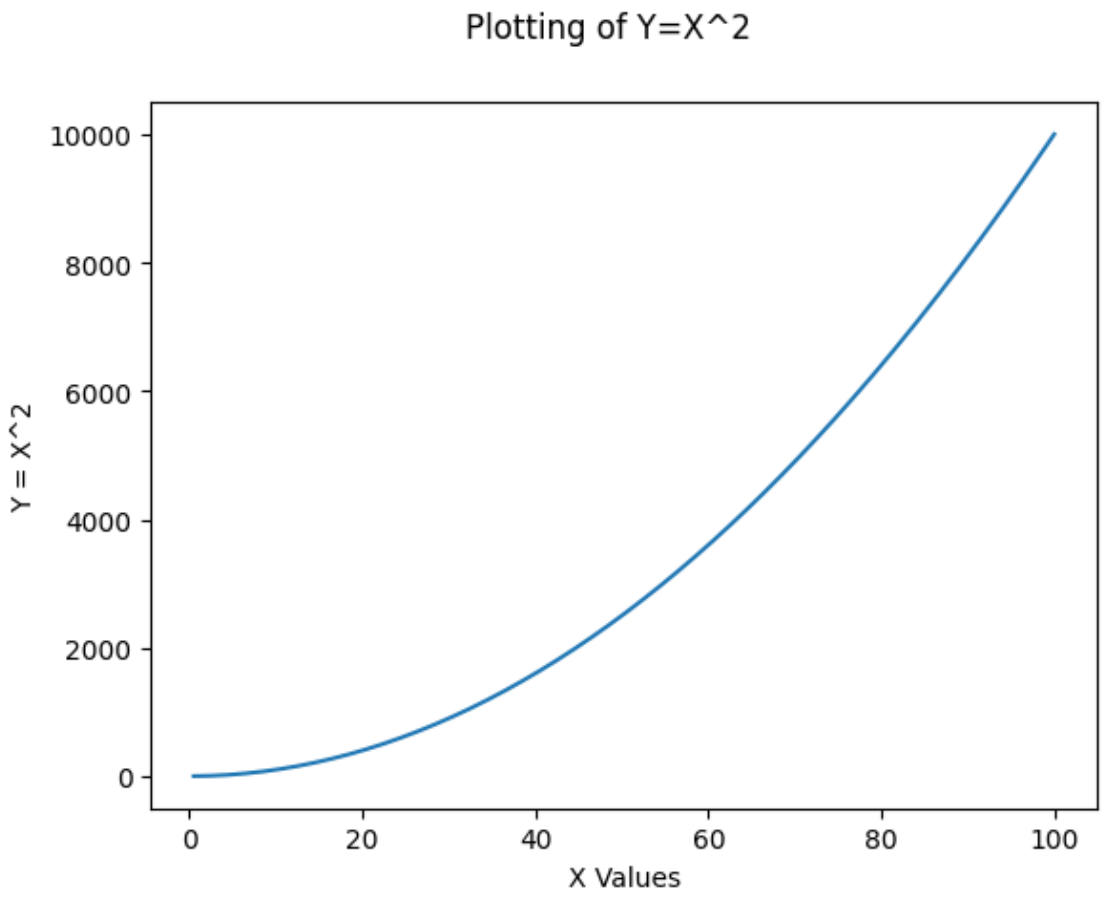
*Q*9. Using matplotlib plot the following functions on the domain: x = 0.5 to 100.0 in steps of 0.5.

[23]:
```
y = []
for i in range(0, 100, 1):
    y.append(i+0.5)
    y.append(i+1)
```

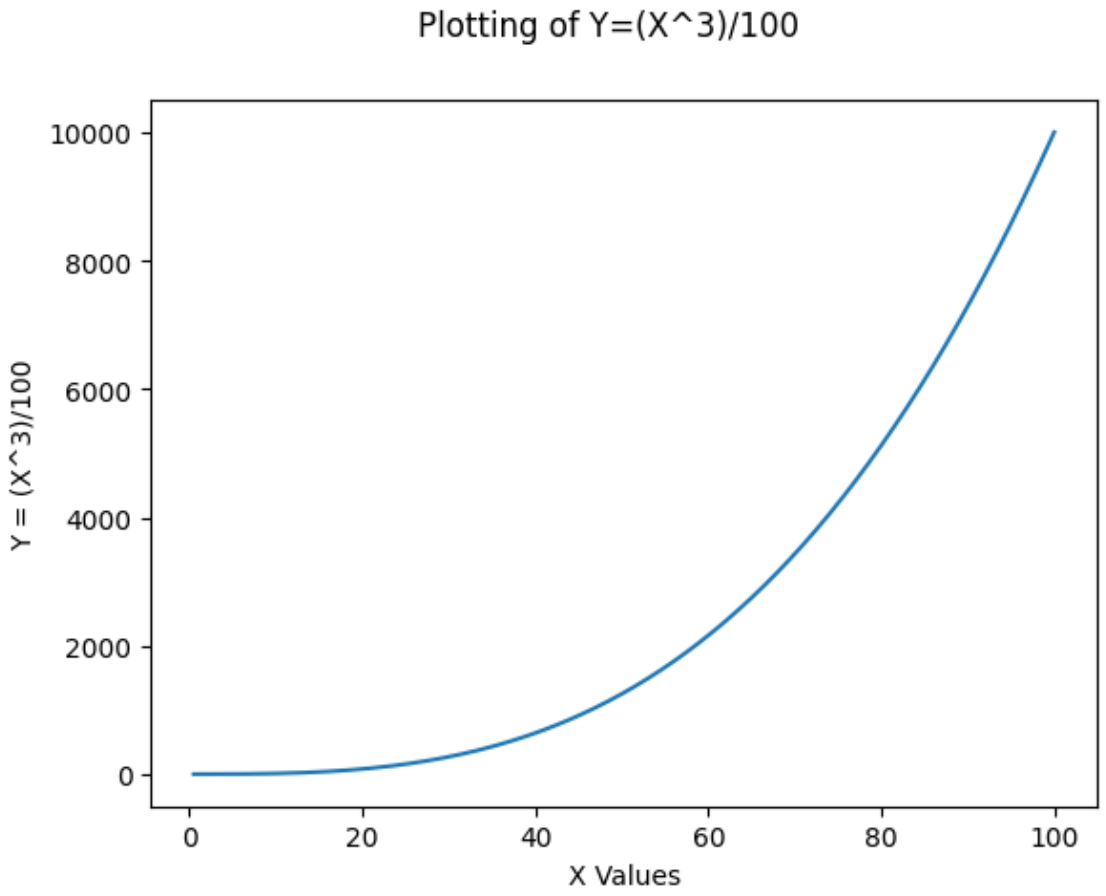
[33]:
```
import matplotlib.pyplot as plt

x = []
for y_elm in y:
    x.append(y_elm)

plt.suptitle('Plotting of Y=X')
plt.plot(y, x)
plt.ylabel('Y = X')
plt.xlabel('X Values')
plt.show()
```



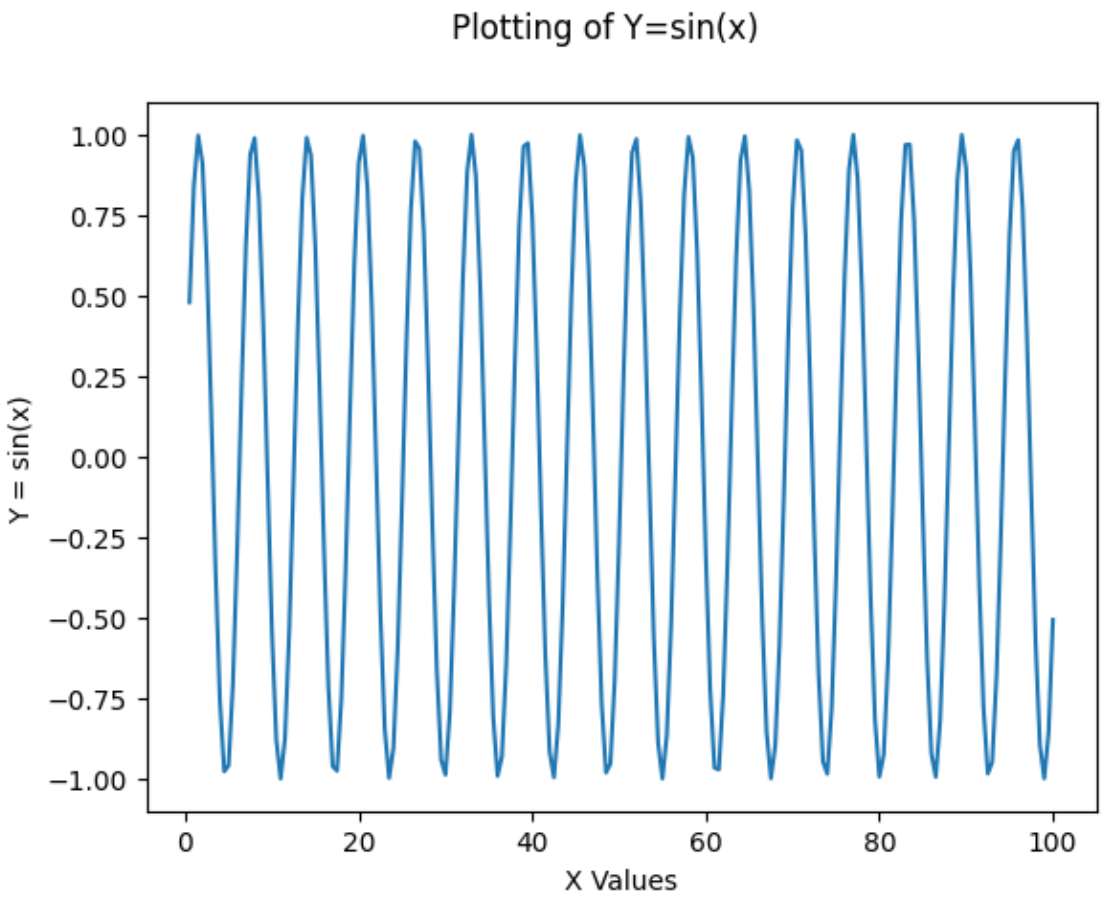
[34]:
```
import matplotlib.pyplot as plt

x = []
for y_elm in y:
    x.append(y_elm ** 2)

plt.suptitle('Plotting of Y=X^2')
plt.plot(y, x)
plt.ylabel('Y = X^2')
plt.xlabel('X Values')
plt.show()
```



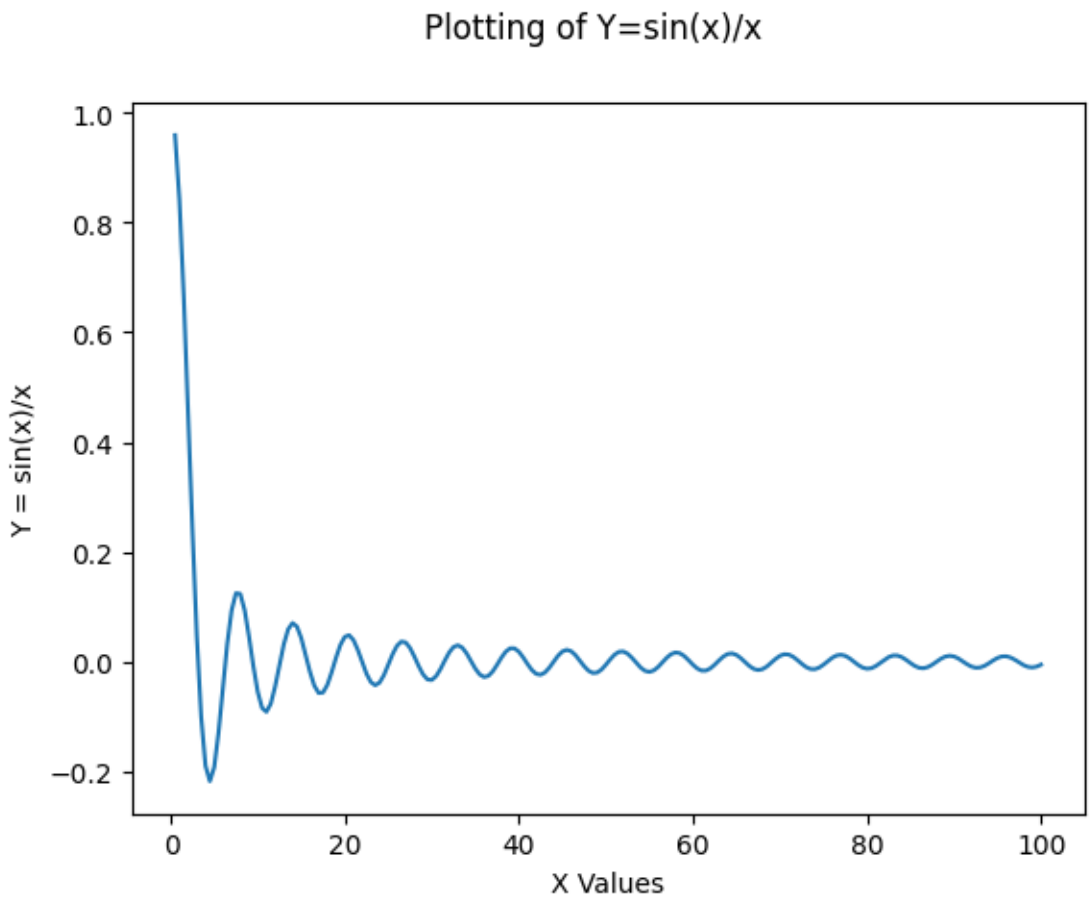
[38]:
```
import matplotlib.pyplot as plt

x = []
for y_elm in y:
    x.append((y_elm ** 3)/100)

plt.suptitle('Plotting of Y=(X^3)/100')
plt.plot(y, x)
plt.ylabel('Y = (X^3)/100')
plt.xlabel('X Values')
plt.show()
```



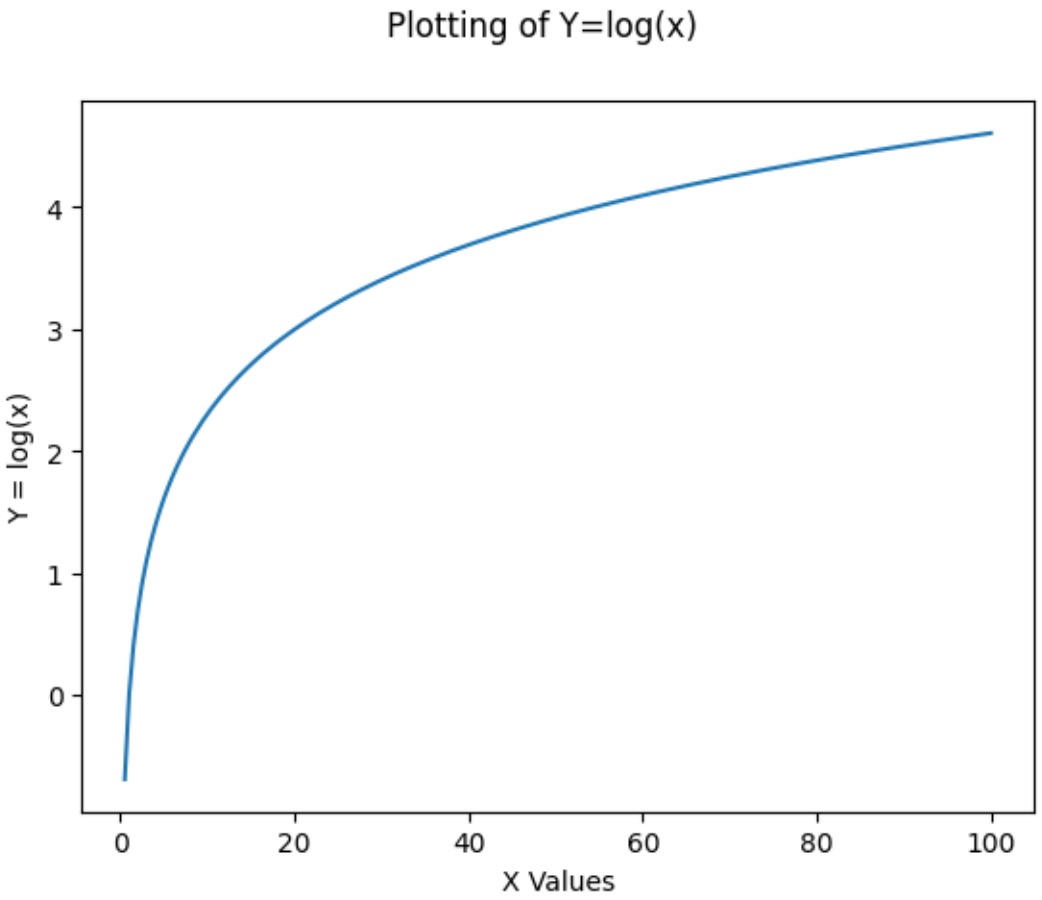
[39]:
```
import matplotlib.pyplot as plt
import math

x = []
for y_elm in y:
    x.append(math.sin(y_elm))

plt.suptitle('Plotting of Y=sin(x)')
plt.plot(y, x)
plt.ylabel('Y = sin(x)')
plt.xlabel('X Values')
plt.show()
```



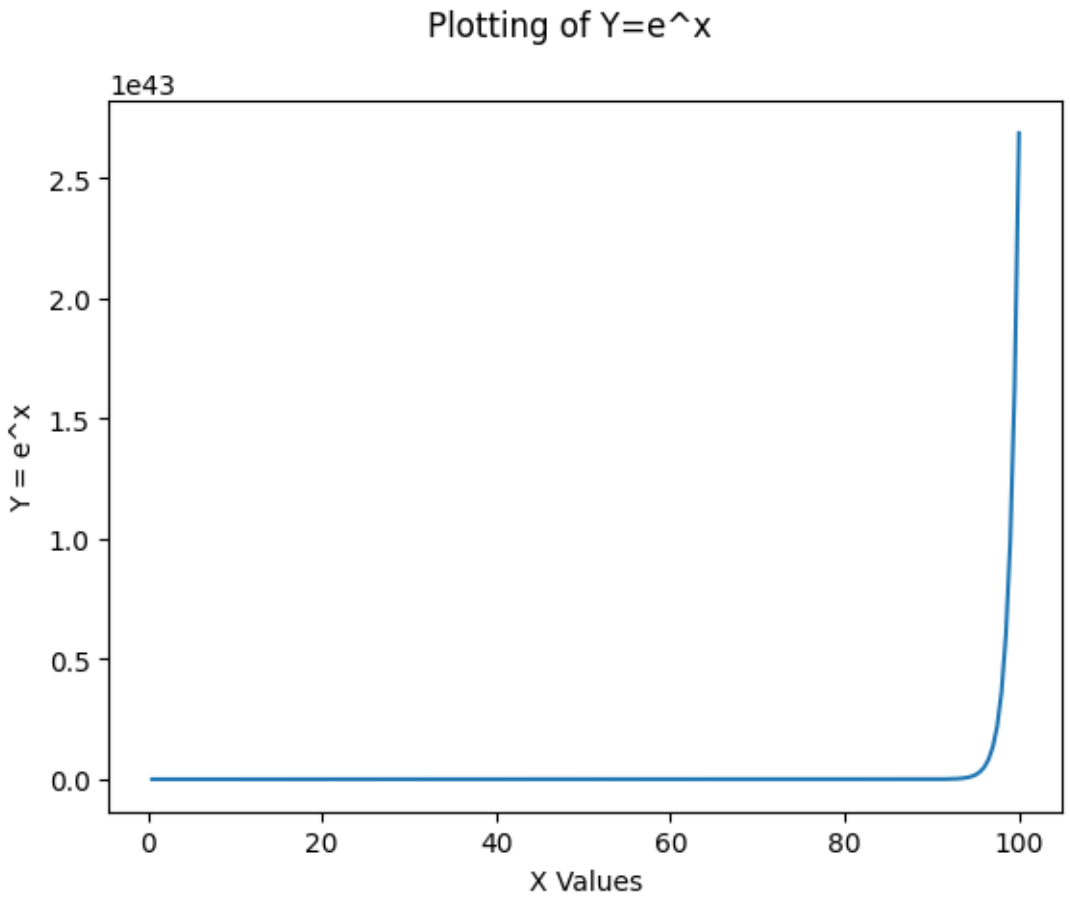
[40]:
```
import matplotlib.pyplot as plt
import math

x = []
for y_elm in y:
    x.append(math.sin(y_elm)/y_elm)

plt.suptitle('Plotting of Y=sin(x)/x')
plt.plot(y, x)
plt.ylabel('Y = sin(x)/x')
plt.xlabel('X Values')
plt.show()
```



[43]:
```
import matplotlib.pyplot as plt
import math

x = []
for y_elm in y:
    x.append(math.log(y_elm))

plt.suptitle('Plotting of Y=log(x)')
plt.plot(y, x)
plt.ylabel('Y = log(x)')
plt.xlabel('X Values')
plt.show()
```



[44]:
```
import matplotlib.pyplot as plt
import math

x = []
for y_elm in y:
    x.append(math.exp(y_elm))

plt.suptitle('Plotting of Y=e^x')
plt.plot(y, x)
plt.ylabel('Y = e^x')
plt.xlabel('X Values')
plt.show()
```



$Q10$. Using numpy generate a matrix of size 20×5 containing random numbers drawn uniformly from the range of 1 to 2. Using Pandas create a dataframe out of this matrix. Name the columns of the dataframe as "a", "b", "c", "d", "e". Find the column with the highest standard deviation. Find the row with the lowest mean.

[134]:
```
import pandas
import numpy as np

matrix = np.random.uniform(1,2, size = (20,5))

df = pandas.DataFrame(matrix, columns= ['a','b','c','d','e'])
print(df)

standard_deviation = dict(df.std())

vals = list(standard_deviation.values())
keys = list(standard_deviation.keys())

print("\nColumn with highest standard deviation:-")
```

```
print("Column \"", keys[vals.index(max(vals))], "\" with standard deviation ",␣
 ↪max(vals))

print("\nRow with least mean value:-")
rows = list(df.mean(axis = 1))
print("Row \"", rows.index(min(rows)), "\" with mean value ", min(rows))
```

```
           a         b         c         d         e
0   1.397961  1.809864  1.501148  1.547764  1.748642
1   1.111476  1.067224  1.156988  1.215723  1.731437
2   1.805599  1.118023  1.099124  1.030712  1.296882
3   1.076872  1.912751  1.609310  1.632673  1.682362
4   1.045389  1.828075  1.879213  1.543981  1.708313
5   1.449278  1.747120  1.576957  1.805881  1.164537
6   1.117274  1.020413  1.590166  1.814976  1.594596
7   1.915075  1.351860  1.239113  1.555772  1.829809
8   1.585068  1.646506  1.827105  1.940596  1.261002
9   1.400897  1.981775  1.458716  1.855007  1.444190
10  1.340567  1.932276  1.829172  1.662288  1.223479
11  1.695456  1.863306  1.982360  1.669557  1.678587
12  1.107161  1.577754  1.361789  1.834584  1.440093
13  1.095025  1.156802  1.642672  1.653846  1.861247
14  1.339497  1.712602  1.201815  1.409876  1.637891
15  1.165316  1.211517  1.591419  1.782690  1.914100
16  1.144533  1.646628  1.991620  1.916159  1.216405
17  1.049540  1.935591  1.259290  1.997732  1.429692
18  1.524998  1.695916  1.348899  1.407045  1.442638
19  1.082724  1.736139  1.268054  1.995999  1.619606

Column with highest standard deviation:-
Column " b " with standard deviation  0.3211616425111993

Row with least mean value:-
Row " 1 " with mean value  1.2565695915482873
```

*Q*11. Add a new column to the dataframe called “f” which is the sum of the columns “a”, “b”, “c”, “d”, “e”. Create another column called “g”. The value in the column “g” should be “LT8” if the value in the column “f” is less than 8 and “GT8” otherwise. Find the number of rows in the dataframe where the value in the column “g” is “LT8”. Find the standard deviation of the column “f” for the rows where the value in the column “g” is “LT8” and “GT8” respectively.

[135]:
```
f = []
g = []
for i in range(len(df.index)):
    f.append(df['a'][i] + df['b'][i] + df['c'][i] + df['d'][i] + df['e'][i])
    g.append("GT8" if f[i] > 8 else "LT8")

df.insert(len(df.columns), "f", f, True)
```

```
df.insert(len(df.columns), "g", g, True)
print(df)
```

```
          a         b         c         d         e         f    g
0   1.397961  1.809864  1.501148  1.547764  1.748642  8.005379  GT8
1   1.111476  1.067224  1.156988  1.215723  1.731437  6.282848  LT8
2   1.805599  1.118023  1.099124  1.030712  1.296882  6.350340  LT8
3   1.076872  1.912751  1.609310  1.632673  1.682362  7.913969  LT8
4   1.045389  1.828075  1.879213  1.543981  1.708313  8.004971  GT8
5   1.449278  1.747120  1.576957  1.805881  1.164537  7.743773  LT8
6   1.117274  1.020413  1.590166  1.814976  1.594596  7.137425  LT8
7   1.915075  1.351860  1.239113  1.555772  1.829809  7.891629  LT8
8   1.585068  1.646506  1.827105  1.940596  1.261002  8.260276  GT8
9   1.400897  1.981775  1.458716  1.855007  1.444190  8.140585  GT8
10  1.340567  1.932276  1.829172  1.662288  1.223479  7.987782  LT8
11  1.695456  1.863306  1.982360  1.669557  1.678587  8.889267  GT8
12  1.107161  1.577754  1.361789  1.834584  1.440093  7.321380  LT8
13  1.095025  1.156802  1.642672  1.653846  1.861247  7.409593  LT8
14  1.339497  1.712602  1.201815  1.409876  1.637891  7.301681  LT8
15  1.165316  1.211517  1.591419  1.782690  1.914100  7.665042  LT8
16  1.144533  1.646628  1.991620  1.916159  1.216405  7.915344  LT8
17  1.049540  1.935591  1.259290  1.997732  1.429692  7.671844  LT8
18  1.524998  1.695916  1.348899  1.407045  1.442638  7.419495  LT8
19  1.082724  1.736139  1.268054  1.995999  1.619606  7.702520  LT8
```

*Q*12. Write a small piece of code to explain broadcasting in numpy.

Ans. The method of broadcasting performs the operations such that a smaller matrix or scalar value gets inside the bigger matrix.

[148]:
```
array = np.array([[1.0, 2.0, 3.0],[4.0, 5.0, 6.0]])

b = 2.0
res = array * b

res2 = res + [1,2,3]

print("Broadcast to do scalar multiplication:-")
print(res)
print("\nBroadcast to do element wise addition:-")
print(res2)
```

```
Broadcast to do scalar multiplication:-
[[ 2.  4.  6.]
 [ 8. 10. 12.]]

Broadcast to do element wise addition:-
[[ 3.  6.  9.]
 [ 9. 12. 15.]]
```

$Q13$. Write a function to compute the argmin of a numpy array. The function should take a numpy array as input and return the index of the minimum element. You can use the np.argmin function to verify your solution.

[21]:
```
array = np.array([[2,3,4],[5,3,10],[7,1,9]])
```

[39]:
```
import numpy as np

def index_min_noaxis(array):
    count = 0
    shape = array.shape
    prod = 1

    for i in shape:
        prod *= i

    array = array.reshape(prod, 1)

    min = array[0]
    min_ind = 0

    for i in range(prod):
        print(array[count], end="")
        if array[count] < min:
            min = array[count]
            min_ind = count
        count += 1
    print()
    return min_ind

def index_min_axis1(array):
    count = 0
    shape = array.shape
    mod = shape[-1]
    temp_minind = 0
    prod = 1

    for i in shape:
        prod *= i

    array = array.reshape(prod, 1)

    min = array[0]
    min_ind = 0
    argmin_axis1 = []

    for i in range(prod):
```

```
        if count % mod == 0 and count != 0:
            argmin_axis1.append(min_ind % mod)
            temp_minind += mod
            min_ind = temp_minind
            min = array[min_ind]
        print(array[count], end="")
        if array[count] < min:
            min = array[count]
            min_ind = count
        count += 1
    print()
    argmin_axis1.append(min_ind % mod)
    return argmin_axis1

# def index_min_axis0(array):
#     count = 0
#     shape = array.shape
#     mod = shape[0]
#     mod_loop = shape[-1]
#     temp_minind = 0
#     prod = 1

#     for i in shape:
#         prod *= i

#     array = array.reshape(prod, 1)

#     min = array[0]
#     min_ind = 0
#     argmin_axis0 = []

#     for i in range(mod_loop):
#         for j in range(mod):
#             if count % mod == 0 and count != 0:
#                 argmin_axis0.append(min_ind % mod)
#                 temp_minind += mod
#                 min_ind = temp_minind
#                 min = array[min_ind]
#         print(array[count], end="")
#         if array[count] < min:
#             min = array[count]
#             min_ind = count
#         count = count * temp_minind
#     print()
#     argmin_axis0.append(min_ind % mod)

#     return argmin_axis0
```

```
print(index_min_noaxis(array))

print(index_min_axis1(array))

# print(index_min_axis0(array))  # axis0 work in progress
```

```
[2] [3] [4] [5] [3] [10] [7] [1] [9]
7
[2] [3] [4] [5] [3] [10] [7] [1] [9]
[0, 1, 1]
```

[43]:
```
print(np.argmin(array))
print(np.argmin(array, axis=0))
print(np.argmin(array, axis=1))
```

```
7
[0 2 0]
[0 1 1]
```