UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

Numerical Methods in Computer Science

Professors:

Nicoletta Del buono

Francesea Mazzia

Student:

Chandrika Patibandla

Table of Contents

Task 1:

Let A be a sparse matrix with uniformly distributed value with density 0.15, of size 20000 rows and 200 colums. Let B be the matrix of size 15000 rows and 300 columns with the first 200 columns equals to A and the last 100 columns $B_{:,200:300} = A * c_a + 10^{-6} c_b$, with c_A, c_B random matrices with uniformly distributed values and density 0.25.

Let q_1, q_2 and q_3 three query vectors defined as $Bx_i + 10^{-5} * c_i$ with x_1 a sparse random vector with density 0.25, x_2 a sparse random vector with density 0.6, x_3 a sparse random vector with density 0.85 and c_i random vectors with uniformly distributed values and density 0.2.

Compute the condition number of the least square problems $min_x \|Bx - q_i\|_2, \qquad i = 1, 3.$

Solve the least square problems using all the studied techniques (QR, SVD, system of normal equations (solving the linear system with the Cholesky factorisation), Truncated SVD, LGK bidiagonalization (implemented for sparse matrices), gradient descent) and compare them.

For the truncated SVD and the LGK bidiagonalization, plot the relative residuals for each k and determine the values of k for which the relative residual is less than 0.1.

Understanding and changing the dimensions of the both matrices:

As per the question A be the matrix size of 20000 rows and 200 columns, and B be another matrix of size 15000 rows and 300 columns.

The first 200 columns of B equal to A and last 100 columns is $B_{:,200:300} = A * C_a + 10^{-6} C_b$

But A has 20000 rows and B has 15000 rows which means the size of the rows of both matrices is different. So, it's unworkable to compute task 1.

So, for this reason I am considering the size of 2000 rows, 20 columns for matrix A and size of 2000 rows, 30 columns for matrix B.

After changing the dimensions, the matrices:

A be the sparse matrix with density of 0.15 with size of 2000 rows and 20 columns that is:

$$A = [\quad]_{2000\ x\ 20}.$$

B be the sparse matrix with density of 0.25 with size of 2000 rows and 30 columns that is:

$$B = [\quad]_{2000\ x\ 30}$$

i.e.,

$$B_{2000\ x\ 0:20} = A$$

$$B_{2000\ x\ 20:30} = A * C_a + 10^{-6} C_b$$

Task 1 Implementation:

I have separated the task 1 into 3 parts.

- Generating the Matrix.
- Computing the Query vectors
- The main part is implementing the least squares problem with different techniques and comparing them.

Generating the Matrix A:

```
1 # The size of matrix A is 2000*20, with the density of 0.15
2 A = sparse.rand(2000, 20, format='csr', density = 0.15, random_state = None)
3 # To understand the size of the matix
4 print("The shape of the matrix A: \n", A.shape)
```

```
The shape of the matrix A:
 (2000, 20)
```

For above code:

Line 2: we know the matrix A is sparse matrix, with density of 0.15 and uniformly distributed values for the reason the *random_state = NONE*.

Line 4: To just see the shape of the matrix A.

Generating the Matrix B:

Before generating the matrix B, I have created the C_a *and* C_b matrix, because we know that

$$B_{2000\,x\,20:30} = A * C_a + 10^{-6} C_b$$

The size of C_a *and* C_b ?

- There is no specification of the size of C_a *and* C_b in the task, so, I have chosen for C_a *is* 10 *columns* $\left(because\ we\ are\ computing\ B_{:x\,20:30}\ that\ means\ only\ 10\ columns\right)$ *and* 20 *rows* (*which is equal to same number of columns of* A)

  because we are going to multiple the A with C_a.

  For C_b is 10 columns (which same number of columns of C_a) and 2000 rows (which same number of rows of A) because we are adding $A * C_a$ *with the* $10^{-6} C_b$.

```
1  # To compute with the matrix A, C_a has column is 10 and number of rows is 20.
2  C_a = sparse.random(20, 10, format='csr', density=0.25)
3  print("The shape of the C_a: \n", C_a.shape)
4
5  # to compute with the matrix A * C_a, so we C_b should be with size of 2000 rows and 10 columns.
6  C_b = sparse.random(2000, 10, format='csr',density=0.25)
7  print("The shape of the C_b: \n", C_b.shape)
8
```

```
The shape of the C_a:
 (20, 10)
The shape of the C_b:
 (2000, 10)
```

For above code:

Line 2: Matrix C_a is a random matrix with size of 20 rows, 10 columns with density of 0.25.

Line 3: shape of the matrix C_a.

Line 6: Matrix C_b is a random matrix with size of 200 rows, 10 columns with density of 0.25.

Line 7: shape of the matrix C_b.

Now, computing the $A * C_a + 10^{-6} C_b$ is:

```
1  # The first element i.e., A * C_a
2  first_element_of_B= A.dot(C_a)
3  print("Shape of A * C_a size: \n", first_element_of_B.shape)
4
5  # the second element i.e., multiplied by 10 power of -6.
6  exp = 10 ** -6
7  second_element_of_B = exp * C_b
8  print("\n Shape 10 ** -6 * C_b size: \n", second_element_of_B.shape)
9
10 # Adding first element + second element
11 B200_300 = first_element_of_B + second_element_of_B
12 print("\n Shape of A * C_(a )+ 10**(-6) * C_b: \n", B200_300.shape)
```

```
Shape of A * C_a size:
 (2000, 10)

 Shape 10 ** -6 * C_b size:
 (2000, 10)

 Shape of A * C_(a )+ 10**(-6) * C_b:
 (2000, 10)
```

For above code:

Line 2: computing the A * C_a, I named it as first element

Line 3: Shape of A * C_a

Line 6: for the 10^{-6}

Line 7: computing the $10^{-6}\ C_b$, I named it as second element

Line 8: Shape of the $10^{-6}\ C_b$

Line 11: adding the first element and second element.

Complete B matrix

```
1  #For B matrix we are Stack matrix (A and  B200_300) in sequence horizontally
2  B = sparse.hstack([A, B200_300])
3  print("The shape of B: \n", B.shape)
```

```
The shape of B:
 (2000, 30)
```

For above code:

Line 2: I am using hstack() function is used to stack the input matrices horizontally, in order to compute the matrix B.

Compute the vector

As per the task the we have to compute the three query vectors $q_1, q_2 and\ q_3$. Which defined as:

$$Bx_i + 10^{-5} * c_i$$

where x_i sparse random matrices, for i = 1,2,3 with respect to the density of 0.25, 0.6 and 0.85.

Now, I am computing x_i for i = 1,2 and 3 with size of 30 rows (because we are multiplying with B, for the reason we have to choose the same number of columns of B) and 1 column.

```
1  #Computing the xi vectors, where i=1,2,3 with respect to the density of 0.25, 0.6 and 0.85
2
3  # x1 with density of 0.25 with 30 rows because the B matrix has 30 columns
4  x_1 = sparse.random(30,1, format='csr', density = 0.25, dtype= float)
5  print("The sparse random vector of x1",x_1.shape)
6
7  # x2 with denisty of 0.6 because the B matrix has 30 columns
8  x_2 = sparse.random(30,1, format='csr', density = 0.6, dtype= float)
9  print("The sparse random vector of x2",x_2.shape)
10
11 # x3 with density of 0.85 because the B matrix has 30 columns
12 x_3 = sparse.random(30,1, format='csr', density = 0.85, dtype= float)
13 print("The sparse random vector of x3",x_3.shape)
```

```
The sparse random vector of x1 (30, 1)
The sparse random vector of x2 (30, 1)
The sparse random vector of x3 (30, 1)
```

For above code:

Line 4: creating the x_1 sparse random vector with size 30 rows, 1 column with density 0.25.

Line 8: creating the x_2 sparse random vector with size 30 rows, 1 column with density 0.6.

Line 12: creating the x_3 sparse random vector with size 30 rows, 1 column with density 0.85.

To compute $Bx_i + 10^{-5} * c_i$, still now for above process we have the B, x_i. So now we have to compute the c_i where its random vectors for i = 1,2,3 with uniform distributed and with density of 0.2. the size of c_i *should be* same as Bx_i because we are adding them. So, the size is 2000 rows and one column.

```
1  #Computing the ci vectors, where i=1,2,3 with respect to the density of 0.2
2  c_1 = sparse.random(2000,1, format='csr', density = 0.2, dtype= float, random_state = None)
3  c_2 = sparse.random(2000,1, format='csr', density = 0.2, dtype= float, random_state = None)
4  c_3 = sparse.random(2000,1, format='csr', density = 0.2, dtype= float, random_state = None)
```

For overall query vectors is $Bx_i + 10^{-5} * c_i$:

```
1  # This exponistional has multiple with the ci vector for the query 1,2 and 3
2  exp_1 = 10 ** -5
3  #Computing query vectors
4  # qi = B*xi + (10 ** -5) * Ci where i = 1,2 and 3
5  query_1 = B.dot(x_1) + exp_1 * (c_1)
6  print(query_1.shape)
7  query_2 = B.dot(x_2) + exp_1 * (c_2)
8  print(query_2.shape)
9  query_3 = B.dot(x_3) + exp_1 * (c_3)
10 print(query_3.shape)
```

```
(2000, 1)
(2000, 1)
(2000, 1)
```

For above code:

Line 2: 10^{-5}

Line 5: $Bx_1 + 10^{-5} * c_1$

Line 7: $Bx_2 + 10^{-5} * c_2$

Line 9: $Bx_3 + 10^{-5} * c_3$

Still, from all the above implementations we have a B matrix as well as query vectors for i = 1,2,3.

Implementing The Least Square Problem with Different Techniques and Compare them:

To compute the least square problem, the condition is

$$min_x ||Bx - q_i||_2 \text{ where i = 1,2 and 3}$$

Different techniques I have chosen for the task 1 is:

1) QR Least Square Problem.
2) SVD Decomposition
3) Truncated SVD
4) Cholesky decomposition
5) Gradient descent
6) LGK decomposition.

QR Least Square Problem.

The matrix A_{mxn}, if the rank of the A_{mxn} is n and if A = QR is the QR factorization, then solution is:

$$A_{mxn} = Q_{mxm} * R_{mxn}$$

applying transpose on both sides

$$A^T = (QR)^T = R^T Q^T$$

The least square solution of Ax = b from above we know A = QR. So;

QRx = b

then $Q^T QRx = Q^T b$ already we know $Q^T Q = I$.

The solution is Rx = $Q^T b$

As per above the definition, the rank of matrix should be equal to *'n'*. so, the rank of the matrix is:

```
1 from numpy import linalg
2 # rank
3 r = linalg.matrix_rank(B.todense())
4 print("The rank of matrix is:", r)
5 n = B.shape[1]
6 print("n:", n)
```

```
The rank of matrix is: 30
n: 30
```

For above code:

Line 3: The rank of the matrix is 30.

Line 5: number of columns of the matrix.

By the above output we understand that the rank of the matrix and number of columns are same, so we compute the factorization.

```
1  start = time.time()
2  # Q R
3  Q,R = linalg.qr(B.todense())
4
5  sol = []
6
7  # R, Q(TRANSPOSE) * B (where B is query_1)
8  qr_1 = linalg.solve(R, Q.T*(query_1)) # Solve a least square, .T is used as transpose of Q
9  # the error of least square solution of ||q1 - Bx||
10 els_qr1 = Norm((query_1 - B*qr_1),2)/sNorm(query_1)
11 print("the error of least square solution of ||q1 - Bx||", els_qr1)
12 # appending the solution
13 sol.append(els_qr1)
14
15 # R, Q(TRANSPOSE) * B (where B is query_1)
16 qr_2 = linalg.solve(R, Q.T*(query_2)) ## Solve a least square, .T is used as transpose of Q
17 # the error of least square solution of ||q2 - Bx||
18 els_qr2 = Norm((query_2 - B.todense()*qr_2),2)/sNorm(query_2)
19 print("the error of least square solution of ||q2 - Bx||", els_qr2)
20 # appending the solution
21 sol.append(els_qr2)
22
23 # R, Q(TRANSPOSE) * B (where B is query_1)
24 qr_3 = linalg.solve(R, Q.T*(query_3)) ## Solve a least square, .T is used as transpose of Q
25 # the error least square solution of ||q3 - Bx||
26 els_qr3 = linalg.norm((query_3 - B*qr_3),2)/sNorm(query_3)
27 print("the error of least square solution of ||q3 - Bx||", els_qr3)
28 # appending the solution
29 sol.append(els_qr3)
30 end = time.time()
31 time_qr = end - start
32 # just for the observation which have least error
33 r = min(sol)
34 print(r)
35
```

```
the error of least square solution of ||q1 - Bx|| 3.386739002820412e-06
the error of least square solution of ||q2 - Bx|| 1.7541721852436692e-06
the error of least square solution of ||q3 - Bx|| 1.3242159711232488e-06
1.3242159711232488e-06
```

For above code:

Line 2: find the Q and R with the help of NumPy linear algebra functions.

Line 4: sol will help to store solutions

Line 7: solving the R, $Q^T * query\_1$

Line 9: finding the error of the least square solution i.e., $||q1 - Bx||_2$ and saved the solution.

Line 15: solving the R, $Q^T * query\_2$

Line 17: Line 9: finding the error of the least square solution i.e., $||q2 - Bx||_2$ and saved the solution.

Line 23: solving the R, $Q^T * query\_3$

Line 25: Line 9: finding the error of the least square solution i.e., $||q1 - Bx||_2$ and saved the solution.

SVD Decomposition:

For each $A \epsilon R^{m*n}$ of rank r, there are orthogonal matrices U_{mxm}, V_{nxn} and a rectangular diagonal matrix of singular values of A such that they are σ_i for i = 1,2…r then $D_{rxr} = diag(\sigma_1 \ldots \sigma_r)$, then

$$A = U * E_{mxn} * V^T$$

The least square solution Ax = b, then

$$(U * E * V^T)\, x = b$$

$U * E * y = b$ where $y = V^T x$

by solving the equation:

$$x = V.E^{\dagger}U^T.b$$

We can understand that the equation is a pseudo inverse of the matrix.

```
1 U, E, VT = sp.sparse.linalg.svds(B)
```

performed the svd on the matrix B, but anyway we can use pseudo inverse (i.e., linalg.pinv function can be used from numpy)

```
1  start = time.time()
2  # It will take long process to compute the VE ** -1U.T, instead of that we can use pinv function to compute it.
3  # But this function is possiable if the rank of matrix is equal to  n
4  rank = linalg.matrix_rank(B.todense())
5  #print("The rank of matrix is:", r)
6  n = B.shape[1]
7  #print(n)
8  if rank == n:
9      # pseudo inverse for the matrix B
10     Pseudo_inverse = linalg.pinv(B.todense())
11     #print(Pseudo_inverse)
12
13     # Least-squares solution with query_1
14     svd1 = Pseudo_inverse*query_1
15
16     # Error of solution ||qi-B*x||
17     els_svd1 = Norm((query_1 - B * svd1),2)/sNorm(query_1)
18     print('error of the least-squares solution: ||q1-B*x||', els_svd1)
19
20     # Least-squares solution with query_2
21     svd2 = Pseudo_inverse*query_2
22
23     # Error of solution ||qi-B*x||
24     els_svd2 = Norm((query_2 - B * svd2),2)/sNorm(query_2)
25     print('error of the least-squares solution: ||q2-B*x||', els_svd2)
26
27     # Least-squares solution with query_1
28     svd3 = Pseudo_inverse*query_3
29
30     # Error of solution ||qi-B*x||
31     els_svd3 = Norm((query_3-B * svd3),2)/sNorm(query_3)
32     print('error of the least-squares solution: ||q3-B*x||', els_svd3)
33 else:
34     print("The rank is not same as n")
35 end = time.time()
36 time_svd = end - start
```

Line 10: pseudo inverse for the matrix B

Line 14: Least square solution with Query_1 (i.e., solution of pseudo inverse) * query_1

Line 16: Error of the solution

Line 21: Least square solution with Query_2 (i.e., solution of pseudo inverse) * query_2

Line 24: Error of the solution

Line 28: Least square solution with Query_3 (i.e., solution of pseudo inverse) * query_3

Line 31: Error of the solution

Solution are shown below:

```
error of the least-squares solution: ||q1-B*x|| 4.596215605686414e-06
error of the least-squares solution: ||q2-B*x|| 1.453798640625841e-06
error of the least-squares solution: ||q3-B*x|| 1.7752472114148469e-06
```

Truncated SVD:

In truncated SVD, the truncation are nothing but number of columns i.e.,

$$1 < \text{Truncate} < \text{rank of matrix B}$$

so, we can take t (truncate) as the largest singular value.

corresponding left & right singular vectors:

$$B_k = U_k E_k V_k^T$$

In our case, the rank of the matrix is 30.

```
1  start = time.time()
2  # rank
3  rank = linalg.matrix_rank(B.todense())
4  n = B.shape[1]
5  #print(n)
6  B_dense = B.todense()
7  # For Truncated SVD implementations
8  if rank == n:
9      U, E, VT = np.linalg.svd(B_dense, full_matrices=False)
10     V = VT.T
11 else:
12     print("The rank is not same")
13 # residuals
14 residual_1 = np.zeros(rank)
15 residual_2 = np.zeros(rank)
16 residual_3 = np.zeros(rank)
17
18 # this array will help to see, the errors of each quearys, with respect to the each column
19 q_1 = []
20 q_2 = []
21 q_3 = []
22 for i in range(1,30):
23
24         # E inverse
25         E_inverse = np.diag(np.hstack([1/E[:i], np.zeros(n-i)]))
26         A_plus = V * E_inverse * U.T
27
28         # truncated SVD with query 1
29         tsvd1 = A_plus*query_1
30         # error
31         els_tsvd1 = Norm((query_1-B *tsvd1),2)/sNorm(query_1)
32         q_1.append(els_tsvd1)
33         residual_1[i-1] = els_tsvd1
34
35         # truncated SVD with query 2
36         tsvd2 = A_plus*query_2
37         #error
38         els_tsvd2 = Norm((query_2-B_dense*tsvd2),2)/sNorm(query_2)
39         q_2.append(els_tsvd2)
40         residual_2[i-1] = els_tsvd2
41
42         # truncated SVD with query 3
43         tsvd3 = A_plus*query_3
44         #error
45         els_tsvd3 = Norm((query_3-B_dense*tsvd3),2)/sNorm(query_3)
46         q_3.append(els_tsvd3)
47         residual_3[i-1] = els_tsvd3
48
```

code:

Line 8- 12: finding rank of the matrix and columns of the matrix is same, if yes we find the U, E and V^T. Otherwise we will say the rank of the matrix is not the same as columns of the matrix.

Line 22: As per the definition, the number of truncates will be between the 1 and rank of the matrix, for the reason I started the loop from 1 and ended the loop 30.

Line 23 - 46: For each iteration, we are finding the solutions, also finding the error of least square solution with respect to the query 1, 2 and 3.

```
49 # printing the error of least squares
50 for i in range(0,30):
51     if residual_1[i] < 0.1:
52         print(f'The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k ={i} => {residual_1[i]
53 print("--------------------------------------------------------------------------------")
54 for i in range(0,30):
55     if residual_2[i] < 0.1:
56         print(f'The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k ={i} => {residual_2[i]
57 print("--------------------------------------------------------------------------------")
58 for i in range(0,30):
59     if residual_3[i] < 0.1:
60         print(f'The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k ={i} => {residual_3[i]
61 print("--------------------------------------------------------------------------------")
```

As per task, I have determined the values of k for each relative if residual is less than 0.1

code:

Line 51: if the result residual of truncated svd with query 1 is less than 0.1. Then printing the value along with the iteration number that is K.

Line 54: if the result residual of truncated svd with query 2 is less than 0.1. Then printing the value along with the iteration number that is K.

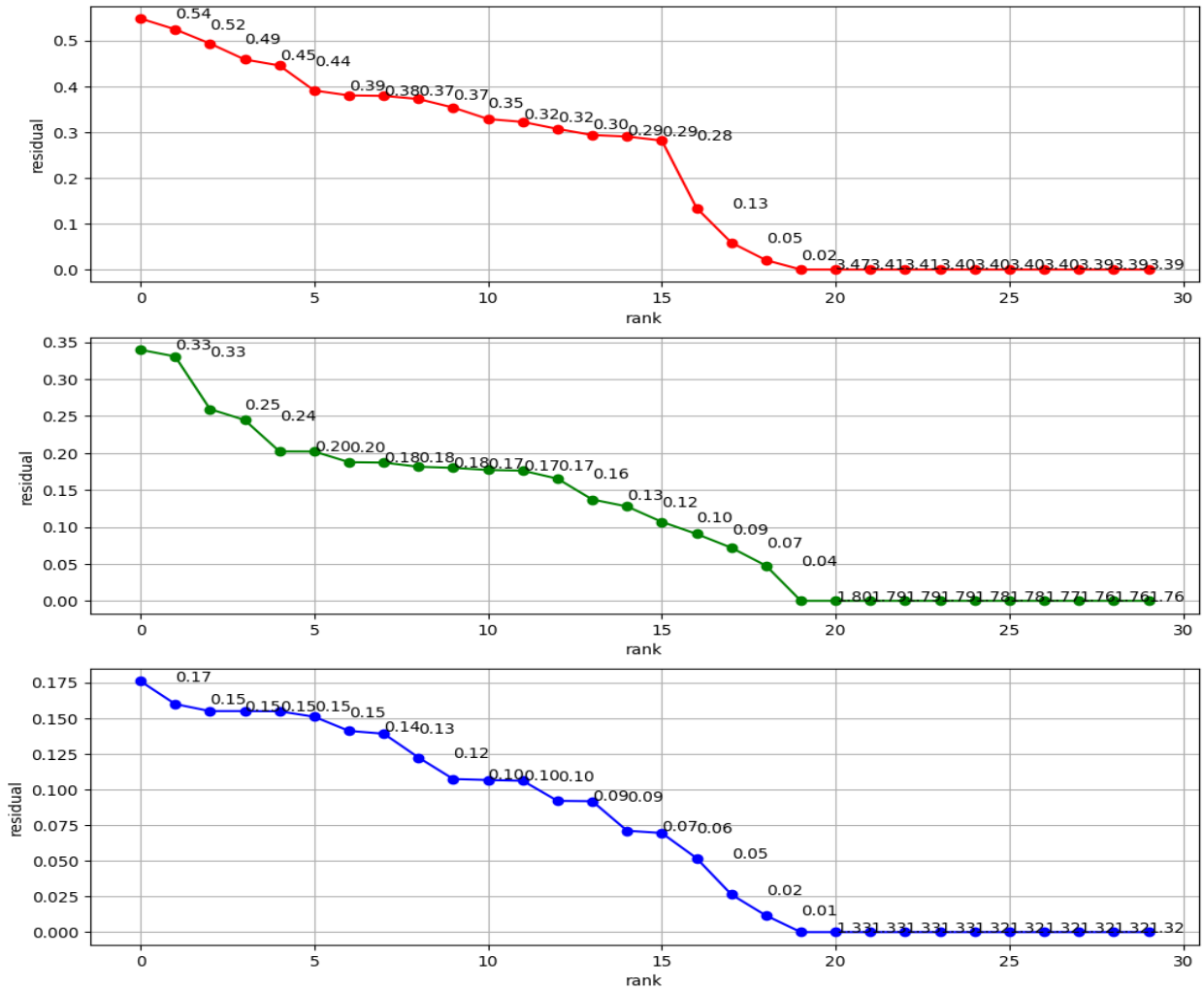
Line 58: if the result residual of truncated svd with query 3 is less than 0.1. Then printing the value along with the iteration number that is K.

The results are shown below

```
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =17 => 0.05867469635573534
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =18 => 0.020317964420784138
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =19 => 3.475275619930046e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =20 => 3.419012166890906e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =21 => 3.4167506243025395e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =22 => 3.4020146748369277e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =23 => 3.401958676897994e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =24 => 3.4006112005868546e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =25 => 3.4006056310134537e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =26 => 3.3994918111185057e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =27 => 3.392146161566063e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =28 => 3.3905485745214405e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =29 => 0.0
--------------------------------------------------------------------------------
```

```
--------------------------------------------------------------------------------
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =12 => 0.09202883680184198
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =13 => 0.09178894983605532
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =14 => 0.07106309584988661
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =15 => 0.06948288425031642
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =16 => 0.05181166191312288
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =17 => 0.026380872616480807
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =18 => 0.011568432031816128
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =19 => 1.3352375997924343e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =20 => 1.3340252136779784e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =21 => 1.3338075437395584e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =22 => 1.3328095324297572e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =23 => 1.3296662747898866e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =24 => 1.3286074802716813e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =25 => 1.3276826915072914e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =26 => 1.3275344447998532e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =27 => 1.3262641015403511e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =28 => 1.3262299827376518e-06
The residual of least square problem with Truncated SVD when it is lessthan 0.1 for k =29 => 0.0
--------------------------------------------------------------------------------
```

Also for truncated SVD, plotting the result residual.



Cholesky Decomposition

Matrix A is positive definite and real matrix, then cholesky decomposition of the form:

$$A = L * L^T$$

Where L is the lower triangular matrix with positive diagonal entries.

The least square solution of Ax = b from above we know $A = L * L^T$

so, $L * L^T x = b$

Lz = b for z

$L^T x$ = z for x

The system normal equation:

$$A^T * A x = A^T b$$

then

$B = A^T * A$, the B is symmetric positive definite given that A.

y = $A^T b$

Algorithm:

compute the cholesky factor.

solve the lower triangular system Lz = y for z

Solve the upper triangular system $L^T x = z$ *for x*

```
2  # matrix B
3  B_dense = B.todense()
4  # Matrix B transpose
5  BT = B_dense.T
6  #print(BT)
7
8  # multiplying the B and B transpose
9  BTB = BT.dot(B_dense)
10
11 # implemeantation of cholesky
12 CF = linalg.cholesky(BTB)
13
```

Code:

Line 12: computing the Cholesky factor.

As per above explanation of algorithm:

```
14 # y = AT b
15 y_1 = BT*(query_1)
16 # solve the lower triangular system Lz = y for z
17 z_1 = linalg.solve(CF, y_1)
18 # Solve the upper triangular system LTx = z for x
19 x1 = linalg.solve(CF.T, z_1)
20 #error
21 els_ch1 = Norm((query_1-B*x1),2)/sNorm(query_1)
22 print('error of the least-squares solution: ||q1-B*x||', els_ch1)
23
```

code:

Line 15: computing y = B^T *query* 1

Line 17: solve the lower triangular system Lz = y for z

Line 19:Solve the upper triangular system $L^T x$ = z for x

Line 21: finding the error of least square solution

we have to compute same process for both query 2 and query 3 as well:

```
24 # y = AT b
25 y_2 = BT*(query_2)
26 # solve the lower triangular system Lz = y for z
27 z_2 = linalg.solve(CF, y_2)
28 # Solve the upper triangular system LTx = z for x
29 x2 = linalg.solve(CF.T, z_2)
30 #error
31 els_ch2 = Norm((query_2-B_dense*x2),2)/sNorm(query_2)
32 print('error of the least-squares solution: ||q2-B*x||', els_ch2)
33
34 # y = AT b
35 y_3 = BT*(query_3)
36 # solve the lower triangular system Lz = y for z
37 z_3 = linalg.solve(CF, y_3)
38 # Solve the upper triangular system LTx = z for x
39 x3 = linalg.solve(CF.T, z_3)
40 #error
41 els_ch3 = Norm((query_3-B_dense*x3),2)/sNorm(query_3)
42
43 print('error of the least-squares solution: ||q3-B*x||', els_ch3)
44 end = time.time()
45 time_cd = end - start
```

```
error of the least-squares solution: ||q1-B*x|| 3.3867401407386174e-06
error of the least-squares solution: ||q2-B*x|| 1.7541763403091793e-06
error of the least-squares solution: ||q3-B*x|| 1.3242215922861897e-06
```

Gradient Descent:

For matrix A, the linear least square define as:

$$F(x) = ||Ax - b||^2$$

by solving the $(Ax - b)^T(Ax - b)$

$$x^T A^T Ax - 2x^T A^T b + b^T b$$

after derivating

$$\text{gradient} = 2A^T(A * x_k - b)$$

$$x_{k+1} = x_k - gradient$$

First I created the gradient function to compute:

$$2A^T(A * x_k - b)$$

```
1  def gradient(A, q, x):
2      d_a = 2*(A.T * ((A * (x)) - q))
3      return d_a
4
```

and return the value of gradient.

```
5  def Gradient_descent(A, q,tollerance,alpha):
6      x0 = np.zeros((30,1))
7      xk = np.zeros((30,1))
8      gradient_x = gradient(A, q, x0)
9      current_val = 1
10     iteration = 0
11     # iteration will continue till the absolute error is grather than then the tollerance
12     while(current_val > tollerance) and all(gradient_x!=0):
13         xk = x0 - alpha * gradient_x
14         current_val = linalg.norm(xk - x0)
15         gradient_x = gradient(A, q, xk)
16         x0 = xk
17         iteration = iteration + 1
18
19     return xk, current_val
```

Code:

Line 12: Condition will be continued until the error will less than the tolerance.

Line 13: Next position (xk) = current position (x0) - apha (0.0001) * gradient

Line 15: Update the gradient value.

Line 16: Update next position as current position

```
start = time.time()
# computing the gradient descent
gd1, res1= Gradient_descent(B, query_1, 1e-08,alpha= 0.0001)
print("shape of the Matrix of B:", B.shape)
# error of least square solution  with the query 1
els_gd1 = Norm((query_1-(B * gd1)),2)/sNorm(query_1)
print('error of the least-squares solution: ||q1-B*x||', els_gd1)

# error of least square solution  with the query 2
gd2, res2 = Gradient_descent(B_dense, query_2.todense(), 1e-12, alpha= 0.0001)
els_gd2 = Norm(query_2-B_dense*gd2)/sNorm(query_2)
print('error of the least-squares solution: ||q2-B*x||', els_gd2)

# error of least square solution  with the query 3
gd3, res3 = Gradient_descent(B_dense, query_3.todense(), 1e-12, alpha= 0.0001)
els_gd3 = Norm(query_3-B_dense*gd3)/sNorm(query_3)
print('error of the least-squares solution: ||q3-B*x||', els_gd3)
end = time.time()
time_gd = end - start
```

```
shape of the Matrix of B: (2000, 30)
error of the least-squares solution: ||q1-B*x|| 3.5687365478795217e-06
error of the least-squares solution: ||q2-B*x|| 1.7571459510019382e-06
error of the least-squares solution: ||q3-B*x|| 1.294628163567752e-06
```

Code:

Line 7: Finding the error of least square solution with gradient descent for query 1.

Line 12: Finding the error of least square solution with gradient descent for query 2.

Line 17: Finding the error of least square solution with gradient descent for query 3.

LGK Decomposition:

```
def LGK(A,b,k):

    (m,n) = A.shape
    toll = 1e-12
      # beta
    beta = np.zeros((k+2,1))
    # alfa
    alfa = np.zeros((k+1,1))
    # alfag
    alfag = np.zeros((k+1,1))
    cg = np.zeros((k+1,1))
    da = np.zeros((k+1,1))
    du = np.zeros((k+1,1))
    sg = np.zeros((k+1,1))
    gam = np.zeros((31,1))
    P = np.zeros((m,k+2))
    Z = np.zeros((n,k+1))
```

Code:

Function LGK, will take an input of matrix, query vector and number of iteration. In our case the matrix is B, query are query_1, query_2 & query_3 with respect to the task. and number of iteration are n i.e., 30.

```
# figure
fig =  plt.figure(figsize=(15,15))
# LGK implementation for the query 1
P1, Z1,alfa1,beta1, res1= LGK(B_dense,query_1.todense(),30)
els_lgk1 = abs(res1)/linalg.norm(query_1.todense(),2)
for i in range (0, len(els_lgk1)):
    if els_lgk1[i] < 0.1:
        print(f'The residual of least square problem with LGK when it is lessthan 0.1 for k ={i} => {els_lgk1[i]}')
print("-----------------------------------------------------------------------------------------------------")
# axis 1 for the first position
axis_1 = fig.add_subplot(3,1,1)
axis_1.set_title("with query vector 1")
axis_1.grid(True)
for x,y in zip(np.arange(1,30), els_lgk1):
    axis_1.text(x, y, str(y)[:5])
axis_1.plot(els_lgk1, 'ro-')

```

As per task, I have determined the values of k for each relative if residual is less than 0.1

code:

Line 5: computing the LGK.

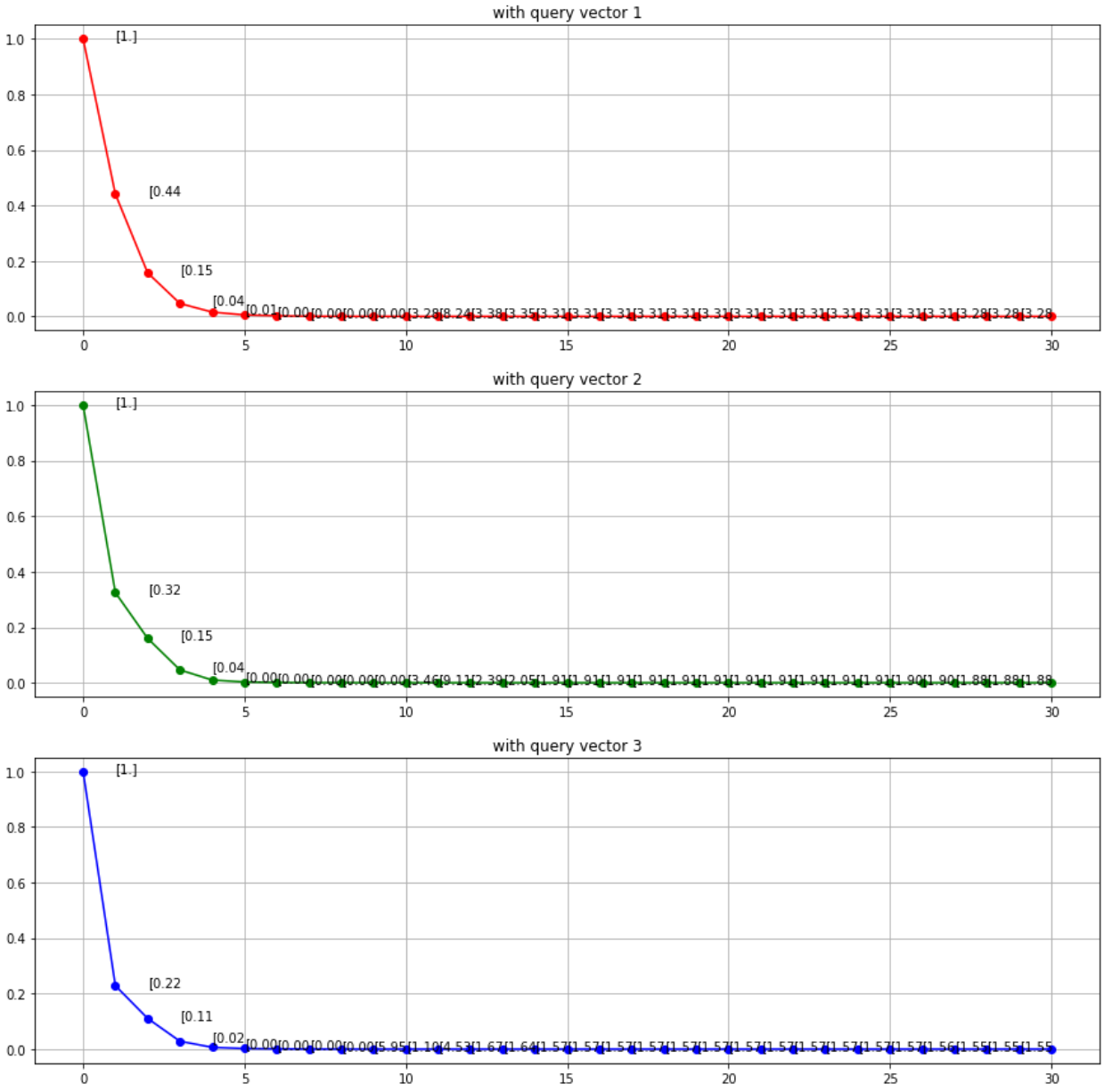
Line 6: finding the errors

Line 7: if the result residual of LGK with query 1 is less than 0.1. Then printing the value along with the iteration number that is K.

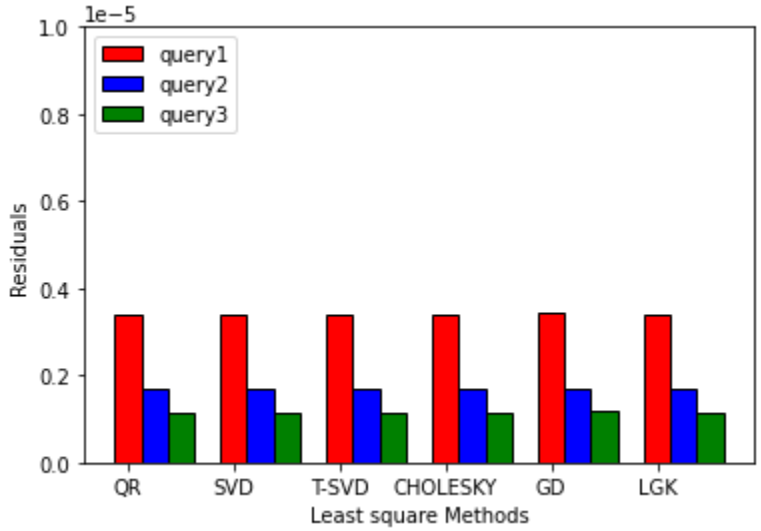
```
The residual of least square problem with LGK when it is lessthan 0.1 for k =15 => [3.3148086e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =16 => [3.31480858e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =17 => [3.31480858e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =18 => [3.31480858e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =19 => [3.31480858e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =20 => [3.31480858e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =21 => [3.31480858e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =22 => [3.31480856e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =23 => [3.31480846e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =24 => [3.31476365e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =25 => [3.3120144e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =26 => [3.2896712e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =27 => [3.28966876e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =28 => [3.28966875e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =29 => [3.28966875e-06]
The residual of least square problem with LGK when it is lessthan 0.1 for k =30 => [3.28966875e-06]
-----------------------------------------------------------------------------------------------------
```

The above process is the same for query 2 and query 3.

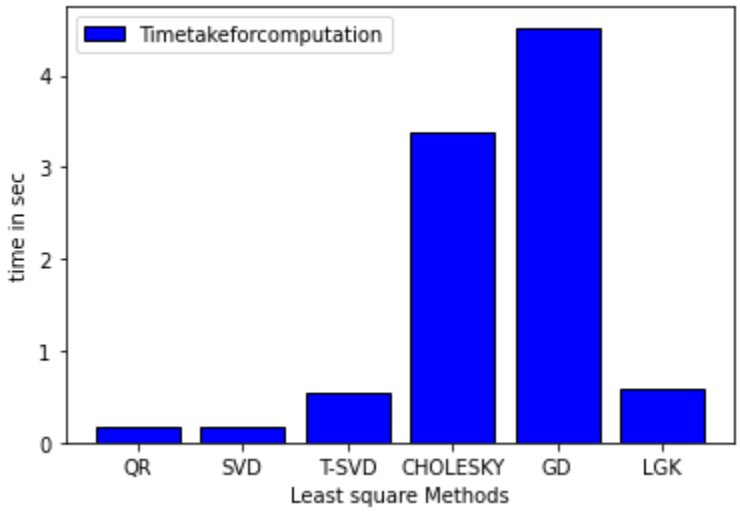
Also for LGK, plotting the result residual.


with query vector 1
[1.]
[0.44
[0.15
[0.04
[0.01
with query vector 2
[1.]
[0.32
[0.15
[0.04
with query vector 3
[1.]
[0.22
[0.11
[0.02

Compare the results:



The error comparison between all the least square methods, and results of all least square methods error almost are same.



Time taken for computation between all the least squares methods, the gradient descent took more and QR and SVD took less time compared to other methods.

Task 2:

Use the gradient descent algorithm (with constant step size) and the pure Newton method to approximate the minimum point ($x^\star = [1, 1]^\top$) of the (Rosenbrock type) function:

$$f(x, y) = \frac{1}{2}10^{-3}(x - 1)^2 + (x^2 - y)^2.$$

Given function:

$$f(x, y) = \frac{1}{2} * 10^{-3}(x - 1)^2 + (x^2 - y)^2$$

In this task , also we have to show the graphical representation of the results such as behavior of the object function and order of the method:

In this task the starting point is the same for both algorithms.

Order of converges:

$$\text{Error} = |current - actual|$$

$$error = |x_n - x^*|$$

$$e_{n+1} = Me_n^\alpha \quad \text{Where } \alpha \text{ is order.}$$

$$\text{So } \alpha = \frac{\ln\left(\frac{e_{n+1}}{e_n}\right)}{\ln\left(\frac{e_n}{e_{n-1}}\right)}$$

$$\text{Then finding } M = \frac{e_{n+1}}{e_n^\alpha}$$

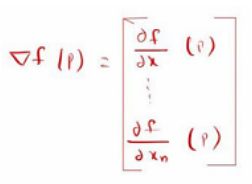
```python
def order_p_and_M(iteration,err_abs, q):
    orderP = []
    m = []
    init = 0
    while init < iteration:
        if init > 1:
            #orderP.append(log)
            alpha_ = (log((err_abs[init])/(err_abs[init-1]))/log((err_abs[init -1])/(err_abs[init -2])))
            orderP.append(alpha_)
            m.append((err_abs[init])/(err_abs[init-1])**alpha_)
        init = init + 1
    return orderP, m
```

for finding order and M the code is implemented according to the above explanation.

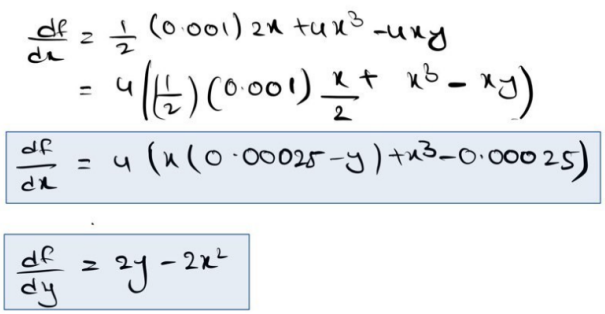
Gradient descent algorithm:

Gradient:

The gradient is the first derivative of the scalar function denoted by ∇, which means vector differential operator, for function $f: R^n \to R$ then its gradient is $\nabla f: R^n \to R^n$ is defined at the point $p = (x_1 \ldots x_n)$ in n dimension space.



In our case the function is already above and derivation is below:



```
1  def gradient(a):
2      x, y = a
3      dx = 4 * (x *(0.00025 - y) + pow(x,3) - 0.00025)
4      dy = 2*y - 2*(x)**2
5      return np.array([dx,dy])
```

For above code:

Line 1: function for gradient

Line 2: x and y

Line 3: $\frac{df}{dx}$

Line 4: $\frac{df}{dy}$

Gradient descent:

It is an iterative solver, generally the iterative solver does not give an exact solution, in such cases the iterative solvers are used to get the approximate solution, the main purpose being to minimize the objective function.

The basic principle of the gradient descent is to choose the step size appropriately so that we can get close to the exact solution.

The gradient descent method rule is

$$x_{k+1} = x_k + \alpha_k(-\nabla f(x_k))$$

Where $x_{k+1} = next\ position,\ x_k = current\ position$

$\alpha_k = step\ length\ of\ my\ gradient,$ and

$-\nabla f(x_k) = descent\ direction$/Derivate.

```
1  def gradientDescent(X,maximum_iteration,toll, alfa = 0.001, iteration = 0):
2      # soln_x is next position value, at starting with 1, 1 shape and filled with zeros
3      soln_x = np.zeros((1,1))
4      # saving each and every step solns values as solnx_history
5      solnx_history = []
6      current_step = 1
7      # saving the function history
8      historyoffunction = []
9      # save the error
10     err_abs = []
11     # result are like table, saving the each soln_x and each iteration
12     result = np.empty((0,2))
13     solnx_history.append(X)
14
15     # starting point has to be compute with gradient.
16     g = gradient(X)
```

Code:

Line 1: Function gradient Descent, the X is input, maximum iteration is 100.

Line 5: saving the soln x values for each and every step as a solnx_history.

Line 16: starting Point has to be computed with the gradient.

```
18     # condition to stop the iterattion process using the present iteration is less than total number of iteration,
19     #  if the current position is less than the tol
20     while all(g!=0) and (current_step > toll) and (iteration < maximum_iteration):
21
22         # g is gradient, it will be update after every iteration
23         # X current position
24         # alfa = 0.001 , we have tried with low running rate
25         soln_x = X - alfa * g
26
27         # saving the infprmation both iteration and soln of X
28         result = np.vstack((result, [soln_x,iteration]))
29
30         # update the G value
31         g = gradient(soln_x)
32
33         # it is nothing but error ||xn - x*|| also it will help to find the order,
34         current_step = scipy.linalg.norm(soln_x - X)
35         # saving the information
36         err_abs.append(current_step)
37
38         # increment the iteration value
39         iteration = iteration + 1
40
41         # every time, we are evaluation the current position with function i.e f(x[0], x[1]) = f(x,y)
42         historyoffunction.append(functn(X))
43
44         # changing the position like current position as previous position
45         X = soln_x
46         solnx_history.append(soln_x)
47     # the output will be last solution of x
48         # iteration value
49         # history of function that is f(x,y)
50         # errors
51         # result i.e solution of x with the respect to the iteration
52     return soln_x, iteration, historyoffunction,solnx_history, err_abs, result
```

Code:

Line 20: Iteration starts here, and the iteration will end if the iteration reaches maximum iteration.

Line 25: computing gradient descent, with current position (X) - alfa (0.001) * gradient to find the next position.

Line 31: update gradient value

Line 34: computing the error and also it will help to find the order.

Line 39: increment the iteration value

Line 45: updating the current position as the previous position.

Results:

```
# viewing the solution of graident descent for each iteration
i = 0
while i < len(table[:,0]):
    print(f"For the iteration {table[i,1]} the solution of Gradient Descent method is {table[i,0]}")
    i = i+1
```

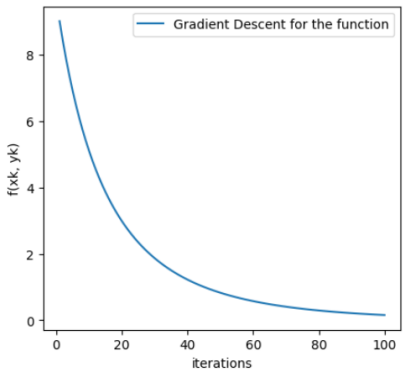
Using results from gradient descent functions, I have printed a solution of gradient descent method for each iteration.

And the approximate solution of the gradient method is:

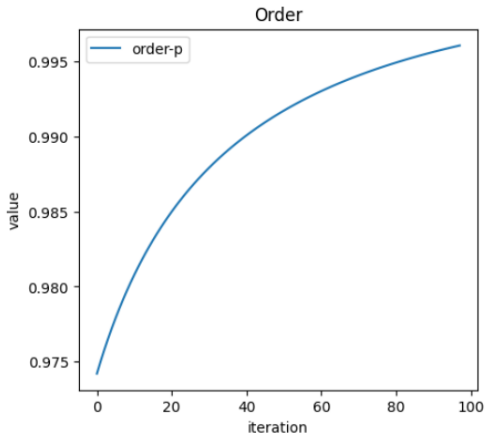
```
print('The approximate solution of the Gradient Descent method is: ',sol_gradientDescent,' which converges with ', numberofi
```

```
The approximate solution of the Gradient Descent method is:  [1.27129546 1.22584167]  which converges with  100  iterated
```

Graphical representation:



The behavior of objective function, of the results of gradient descent algorithm. Here, Y-axis represents the function results values on each iteration (X-axis)



Here the Y-axis represents the values of alpha with respect to the iteration (x-axis).

```
1 # P_ ORDER OF THE METHOD
2 i = len(order_P_gradientDescent)
3 print("The order of convergence of gradient descent: ",round(order_P_gradientDescent[i-2])
4 print("The M value is ", (M_values[i-2]))
```

```
The order of convergence of gradient descent:  1
The M value is  0.9599602316487349
```

Line 3: The order of convergence of gradient descent.

Line 4: The value of M.

Newton Method:

Hessian matrix:

Suppose f:$R^n \rightarrow R$ is a function, if all second order derivatives of f exist. then the hessian matrix H(f) is n*n matrix.

$$Hf = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_n} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

so, in our case:

$$dxx = 12x^2 - 4y + 0.001$$

$$dxy = -4x$$

$$dyx = -4x$$

$$dyy = 2$$

It is an iterative solver, can be used to get approximate solution:

$$x_{k+1} = x_k + d_N$$

where x_{k+1} = Next position

x_k = Current position

d_N = Solution.

then $Hf(x_k)\, d_N = -\nabla f(x_k)$ but the initial point is $x_k \approx x^*$ (using gradient method).

```
1  def newtonmethod(X, maximum_iteration, toll, iteration = 0):
2       # soln_x is next position value, at starting with 1, 1 shape and filled with zeros
3      soln_x = np.zeros((1,1))
4      # saving each and every step solns values as solnx_history
5      solnx_history = []
6      current_step = 1
7      # saving the function history
8      historyoffunction = []
9      # save the error
10     err_abs = []
11     # result are like table, saving the each soln_x and each iteration
12     result = np.empty((0,2))
13     solnx_history.append(X)
14
15     # the initial step is the derviate of the input
16     X = gradient(X)
```

Line 1: Function Newton method, the X is input, maximum iteration is 100.

Line 5: saving the soln x values for each and every step as a solnx_history.

Line 16: Starting Point has to be computed with the gradient because the condition $x_k \approx x^*$ (using gradient method)

```
18      # condition to stop the iterattion process using the present iteration is less than total number of iteration,
19      #  if the current position is less than the tol
20      while (current_step > toll) and (iteration < maximum_iteration):
21
22          # solving the hessian matrix and gradient matrix
23          dn_slove = scipy.linalg.solve(hessian(X), -gradient(X))
24
25          # dv it will be update after every iteration
26          # X current position
27          soln_x = X + dn_slove
28
29          # saving the results
30          result = np.vstack((result, [soln_x,iteration]))
31
32          # it is nothing but error ||xn - x*|| also it will help to find the order,
33          current_step = scipy.linalg.norm(soln_x - X)
34
35          # saving the information
36          err_abs.append(scipy.linalg.norm(soln_x - X))
37
38          # increment the iteration with 1
39          iteration = iteration + 1
40
41          # saving the function history
42          historyoffunction.append(functn(X))
43
44          # update the X value as soln X
45          X = soln_x
46
47          # saving the information
48          solnx_history.append(soln_x)
49
50
51      return soln_x, iteration, historyoffunction,solnx_history, result, err_abs
```

Code:

Line 20: Iteration starts here, and the iteration will end if the iteration reaches maximum iteration.

Line 23: solving the hessian and gradient.

Line 27: Newton method (finding the next position by computing the current position (X) + d_N

Line 36: computing the error and also it will help to find the order.

Line 39: increment the iteration value

Line 45: updating the current position as the previous position.

Results:

```
1  i = 0
2  while i < len(table_nn[:,0]):
3      print(f"\nFor the iteration {table_nn[i,1]} the solution of newton method is {table_nn[i,0]}")
4      i = i+1
5
6  print('\nThe solution of the the solution of newton method is: ',sol_nn,' which converges with ',num_iternn,' iterated \n')
7
```

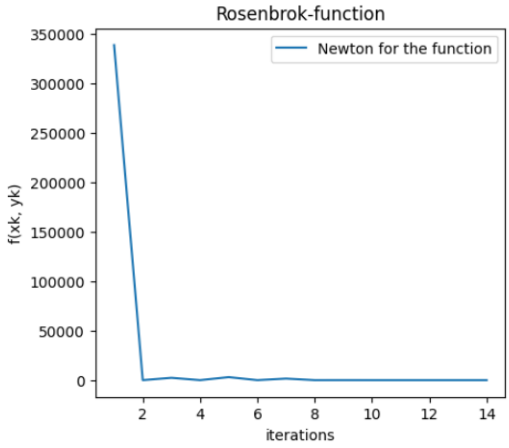
Using results from Newton method functions, I have printed a solution of Newton method for each iteration.

And the approximate solution of the Newton method is:

```
1 it('The approximate solution of the Gradient Descent method is: ',sol_nn,' which converges with ', num_iternn,' iterated \n')
```

```
The approximate solution of the Gradient Descent method is:  [1. 1.]  which converges with  14  iterated
```

Graphical representation:



The behavior of objective function, of the results of the Newton method. Here, Y-axis represents the function results values on each iteration (X-axis)

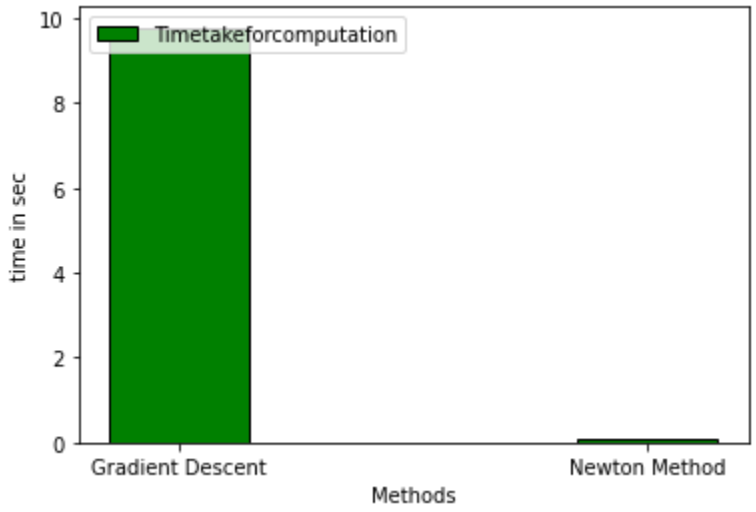
Here the Y-axis represents the values of alpha with respect to the iteration (x-axis).

```
1 # order
2 print("The value of m", m_val[len(order_P_newton)- 2])
3 print("The order of convergence is: ",round(order_P_newton[len(order_P_newton)- 2]))
4
```

```
The value of m 109.44492407795768
The order of convergence is:  2
```

Line 3: The order of convergence of gradient descent.

Line 4: The value of M.



Overall, to reach the minimum point the gradient descent took 100 iterations (yes it didn't reach to [1,1]) whereas the Newton method reached the minimum point [1,1] in 14 iterations only.