2021FC04586_MFDS_01

December 9, 2021

[9]:
```python
import time
import pandas as pd
from copy import deepcopy
from numpy.linalg import norm
import random
import numpy as np
import math
```

0.1 Q) Implementing Gaussian Elimination Method

(i) Find the approximate time your computer takes for a single addition by adding first 10^6 positive integers using a for loop and dividing the time taken by 10^6. Similarly find the approximate time taken for a single multiplication and division. Report the result obtained in the form of a table. (0.5)

[2]:
```python
def add_time(N) :

    val = 1
    start_time = time.time()
    for i in range(2, N + 1) :
        val += i
    return (time.time() - start_time)/10**6

def mul_time(N) :

    val = N
    start_time = time.time()
    for i in range(1, N) :
        val = val * i
    return (time.time() - start_time)/10**6

def div_time(N) :

    val = N
    start_time = time.time()
    for i in range(N - 1, 0, -1) :
        val = i / val
```

```
    return (time.time() - start_time)/10**6
```

[2]:
```
N = 10**6

pd.DataFrame([add_time(N), mul_time(N), div_time(N)],
             columns = ['Time for', N, 'operations'],
             index = ['Addition', 'Multiplication', 'Division'])
```

[2]:
```
                 Time for 10**6 operations
Addition                      8.468103e-08
Multiplication                5.028525e-04
Division                      6.194782e-08
```

(ii) Write a function to implement Gauss elimination with and without pivoting. Also write the code to count the number of additions, multiplications and divisions performed during Gaussian elimination. Ensure that the Gauss elimination performs 5S arithmetic which necessitates 5S arithmetic rounding for every addition, multiplication and division performed in the algorithm. If this is not implemented correctly, the rest of the answers will be considered invalid.

Note that this is not same as simple 5 digit rounding at the end of the computation. Do not hardwire 5S arithmetic in the code and use dS instead. The code can then be run with various values of d. (0.5 + 0.5)

[15]:
```
def significant_digits(a, S = 5):

    if a != 0:
        dec_digits = int(math.floor(math.log10(abs(a))))
        return  round(a, S - dec_digits - 1)
    else:
        return 0

def pivoting(inp, b, c) :

    pivot_ele = []

    for i in range(c, len(inp)) :
        pivot_ele.append(inp[i][c])
    pivot_row = c + np.argmax(np.absolute(pivot_ele))

    if pivot_row > c :
        inp[c], inp[pivot_row] = inp[pivot_row], inp[c]
        b[c], b[pivot_row] = b[pivot_row], b[c]

    return inp, b

def forward_elimination(inp, b, pivot, S) :

    operation_count = {'add_count' : 0, 'mul_count' : 0, 'div_count' : 0}
    for c in range(len(inp[0]) - 1) :

        if pivot :
            inp, b = pivoting(inp, b, c)

        for i in range(c + 1, len(inp)) :
            wt = inp[i][c]/ inp[c][c]
            operation_count['div_count'] +=1
            inp[i][0] = 0.0

            for j in range(c + 1, len(inp[i])) :
                inp[i][j] = significant_digits(inp[i][j] - inp[c][j]*wt, S)
```

```
                operation_count['mul_count'] += 1
                operation_count['add_count'] += 1

            b[i] = significant_digits(b[i] - b[c]*wt, S)
            operation_count['mul_count'] += 1
            operation_count['add_count'] += 1

    return inp, b, operation_count

def back_substitution(inp, b, S) :

    li = []
    operation_count = {'add_count' : 0, 'mul_count' : 0, 'div_count' : 0}
    for i in range(len(inp) - 1, -1, -1) :
        x = 0

        for j in range(len(inp) - 1, -1, -1) :

            if i != j :
                b[i] -= li[len(inp) - j - 1]*inp[i][j]
                operation_count['mul_count'] +=1
                operation_count['add_count'] +=1
            else :
                li.append(significant_digits(b[i]/inp[i][j], S))
                operation_count['div_count'] +=1
                break

    return li, operation_count

def get_operation_count(fe_count, bs_count, display_df = False) :

    df = pd.DataFrame([[fe_count['add_count'], bs_count['add_count']],
                       [fe_count['mul_count'], bs_count['mul_count']],
                       [fe_count['div_count'], bs_count['div_count']]],
                      columns = ['FE_operation_count', 'BS_operation_count'],
                     index = ['Addition', 'Multiplication', 'Division'])
    ind = df.index
    df['Total'] = df.sum(axis = 1)
    df = df.append(pd.Series(df.sum(axis = 0)), ignore_index = True)
    df.index = list(ind) + ['Total']

    if display_df :
        display(df)

    return df.Total[:3]

def gauss_elimination(A, b, pivot = True, S = 5, display_df = False) :
```

```
    A, b, fe_count = forward_elimination(A, b, pivot, S)
    solution, bs_count = back_substitution(A, b, S)

#     print('Matrix A : \n', np.array(A))
#     print()
#     print('RHS ( b ) :', b)
#     print()
#     print('Solution for x : ', np.array(solution))

    count = get_operation_count(fe_count, bs_count, display_df)

    return solution, count
```

(iii) Generate random matrices of size n × n where n = 100, 200, . . . , 1000. Also generate a random b R n for each case. Each number must be of the form m.dddd (Example : 4.5444) which means it has 5 Significant digits in total. Perform Gaussian Elimination with and without pivoting for each of the 10 cases above. Report the number of additions, divisions and multiplications for each case in the form of a table. No need to write matrices. (0.5 + 0.5)

[16]:
```
def generate_number(before_dec = 1, after_dec = 4) :

    num = ''
    for i in range(before_dec):
        num += str(random.randint(1, 9))
    num += '.'
    for i in range(after_dec):
        num += str(random.randint(0, 9))

    return float(num)

def random_matrices(n, bd = 1, ad = 4) :

    mat = []
    for i in range(0, n) :
        row = []
        for j in range(0, n) :
            row.append(generate_number(bd, ad))
        mat.append(row)
    return mat

def random_b(n, bd = 1, ad = 4) :

    y = []
    for j in range(0, n) :
            y.append(generate_number(bd, ad))
    return y

def get_gauss_res(start, end, step, pivot = True, S = 5, display_df = False) :

    res = {}
    gauss_time = {}
    for n in range(start, end + 1, step) :

        M = random_matrices(n)
        b = random_b(n)
        start_time = time.time()
        sol, count = gauss_elimination(M, b, pivot, S, display_df)
        gauss_time[n] = time.time() - start_time

        res[n] = count
```

```
    return res, gauss_time
```

[17]:
```
pivot_res, pivot_gauss_time = get_gauss_res(100, 1001, 100)
# pivot_res
```

[18]:
```
wo_pivot_res, wo_pivot_gauss_time = get_gauss_res(100, 1001, 100, pivot = False)
# wo_pivot_res
```

[19]:
```
x = pd.DataFrame.from_dict(pivot_res, orient='index')
x.columns = ['AddCount_withPivot', 'MulCount_withPivot', 'DivCount_withPivot']
y = pd.DataFrame.from_dict(wo_pivot_res, orient='index')
y.columns = ['AddCount_withPivot', 'MulCount_withPivot', 'DivCount_withPivot']
pd.concat([x,y], axis = 1)
```

[19]:
```
      AddCount_withPivot  MulCount_withPivot  DivCount_withPivot  \
100               338250              338250                5050
200              2686500             2686500               20100
300              9044750             9044750               45150
400             21413000            21413000               80200
500             41791250            41791250              125250
600             72179500            72179500              180300
700            114577750           114577750              245350
800            170986000           170986000              320400
900            243404250           243404250              405450
1000           333832500           333832500              500500

      AddCount_withPivot  MulCount_withPivot  DivCount_withPivot
100               338250              338250                5050
200              2686500             2686500               20100
300              9044750             9044750               45150
400             21413000            21413000               80200
500             41791250            41791250              125250
600             72179500            72179500              180300
700            114577750           114577750              245350
800            170986000           170986000              320400
900            243404250           243404250              405450
1000           333832500           333832500              500500
```

(iv) Using the time calculated in the first step and using the theoretical operation count (total time for an operation = number of operations $\times$ time for one operation), generate the approximate time taken for Gaussian elimination with and without pivoting for the 10 cases. Present this data in a tabular form. Assuming T1(n) is the time calculated for an n$\times$n matrix, plot a graphs of log(T1(n)) vs log(n) and fit a straight line to the observed curve and report the slope of the lines. (1 + 1)

[20]:
```
a = pd.DataFrame.from_dict(pivot_gauss_time, orient='index', columns =␣
↪['Time_with_pivoting'])
a['n'] = a.index
a['Theoretical_Time'] = y.sum(axis = 1)
b = pd.DataFrame.from_dict(wo_pivot_gauss_time, orient='index', columns =␣
↪['Time_wo_pivoting'])
b['n'] = b.index

gauss_time_res = pd.merge(a, b)
gauss_time_res
```

[20]:

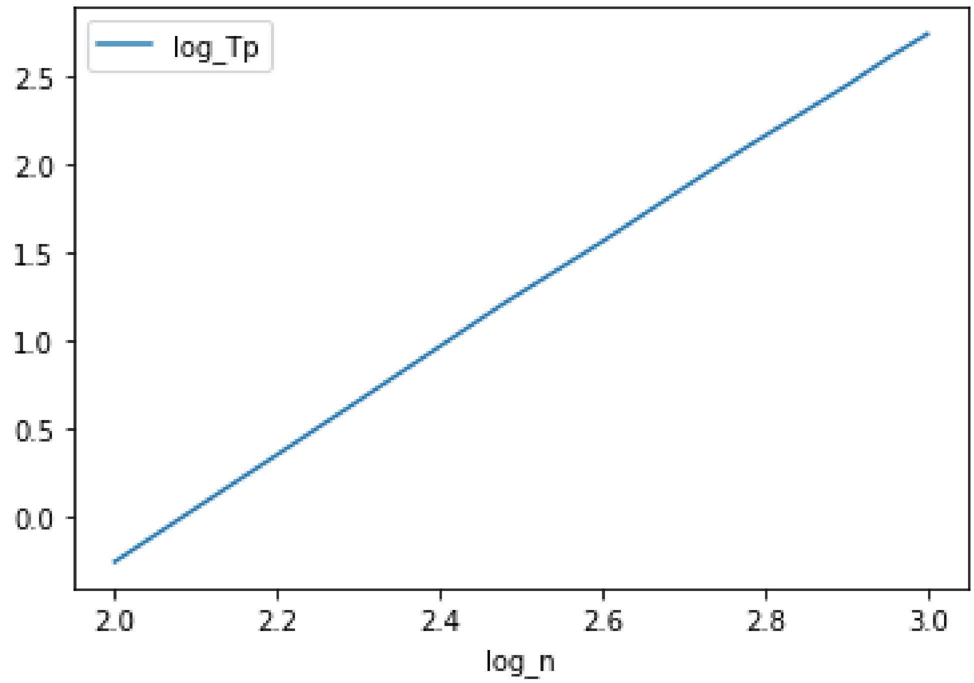
	Time_with_pivoting	n	Theoretical_Time	Time_wo_pivoting
0	0.555032	100	681550	0.572826
1	4.582948	200	5393100	4.498146
2	16.108717	300	18134650	15.163270
3	37.046841	400	42906200	38.135107
4	73.221718	500	83707750	86.049525
5	126.519888	600	144539300	130.411208
6	195.849088	700	229400850	203.893503
7	286.412439	800	342292400	315.006830
8	410.949612	900	487213950	424.412999
9	552.754176	1000	668165500	626.429346

[21]:
```
df = gauss_time_res.copy()
df['log_n'] = np.log10(df.n)
df['log_Tp'] = np.log10(df.Time_with_pivoting)
df['log_Twp'] = np.log10(df.Time_wo_pivoting)
# df
```

[22]:
```
slope_pivot = np.polyfit(df.log_n, df.log_Tp, 1)[0]
print('Slope with pivoting', slope_pivot)
print()
df.plot(x = 'log_n', y = 'log_Tp')
```

```
Slope with pivoting 2.996664272394546
```

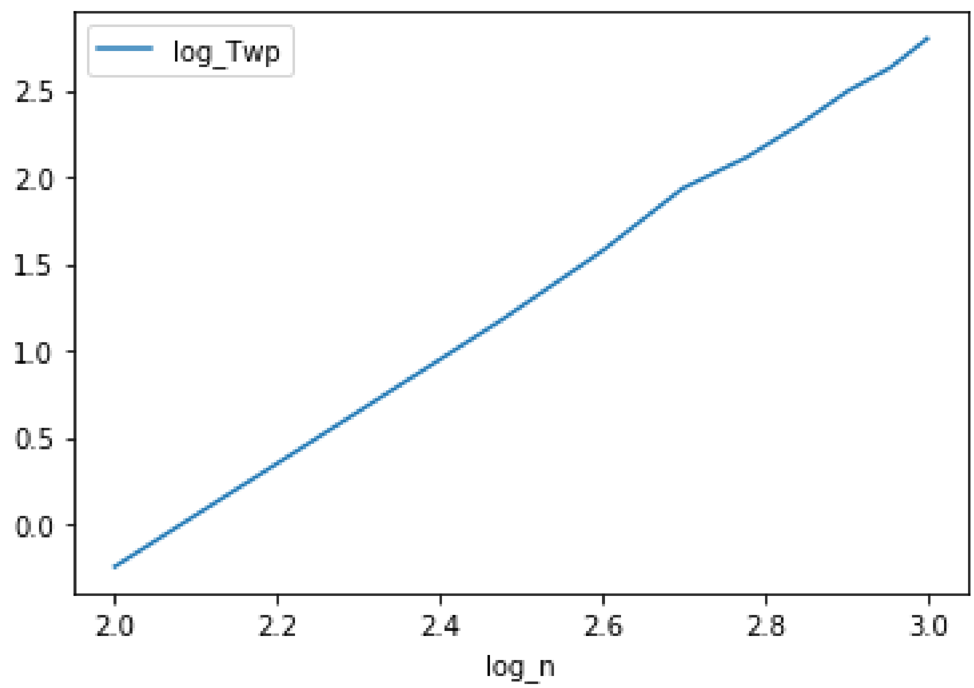
[22]: <matplotlib.axes._subplots.AxesSubplot at 0x118a8f9e8>



[23]:
```
slope_wo_pivot = np.polyfit(df.log_n, df.log_Twp, 1)[0]
print('Slope with pivoting', slope_wo_pivot)
print()
df.plot(x = 'log_n', y = 'log_Twp')
```

```
Slope with pivoting 3.039334657572297
```

[23]: <matplotlib.axes._subplots.AxesSubplot at 0x128cfdc18>


log_Twp
2.5
2.0
1.5
1.0
0.5
0.0
2.0
2.2
2.4
2.6
2.8
3.0
log_n

0.2 Implementing Gauss Seidel and Gauss Jacobi Methods

(i) Write a function to check whether a given square matrix is diagonally dominant or not. If not, the function should indicate if the matrix can be made diagonally dominant by interchanging the rows? Code to be written and submitted. (1) Deliverable(s): The code

[76]:
```
def check_diagonal_dominance(mat) :
    nond_ind = []
    nond_vec = []

    dominant = True

    for i in range(len(mat)) :

        remaining_total = 0
        for j in range(len(mat)) :
            if i != j :
                remaining_total += np.absolute(mat[i][j])

        if not np.absolute(mat[i][i]) > remaining_total :
            dominant = False
            nond_ind.append(i)
            nond_vec.append((i, mat[i]))

    if not dominant :
        for i, row in nond_vec :
            for ind in nond_ind :
                val = 0
                for j, x in enumerate(row) :
                    if j != ind :
                        val = sum([abs(x) ])
                if abs(row[ind]) > val :
                    nond_ind.remove(ind)
                    break

        if len(nond_ind) == 0:
            print("Can be converted to diagonally dominant")
        else:
            print("Cannot be converted to diagonally dominant")
    else :
        print('Matrix is diagonally Dominant')
```

[77]:
```
mat = [[7, -1], [1, -5]]
check_diagonal_dominance(mat)
```

```
Matrix is diagonally Dominant
```

[78]:
```python
mat = [[1, -5], [7, -1]]
check_diagonal_dominance(mat)
```

```
Can be converted to diagonally dominant
```

[80]:
```python
mat = [[1, 1], [7, -1]]
check_diagonal_dominance(mat)
```

```
Cannot be converted to diagonally dominant
```

(ii) Write a function to generate Gauss Seidel iteration for a given square matrix. The function should also return the values of 1, ∞ and Frobenius norms of the iteration matrix. Generate a random 4 $\times$4 matrix. Report the iteration matrix and its norm values returned by the function along with the input matrix. (1) Deliverable(s): The input matrix, iteration matrix and the three norms obtained

[10]:
```
def random_matrices(n) :

    mat = []
    for i in range(0, n) :
        row = []
        for j in range(0, n) :
            row.append(round(random.uniform(1, 10),5))
        mat.append(row)
    return mat
```

[11]:
```
def gauss_seidel_iterations(M) :

    n = len(M)
    L = [[0] * n]*n
    U = [[0] * n]*n

    mat = M.copy()
    for i in range(n) :
        if mat[i][i] != 1 :
            mat[i] = [val/mat[i][i] for val in mat[i]]
        L[i] = mat[i][:i] + [0]*(n - i)
        U[i] = [0]*(i+1) + mat[i][i+1:]

    iter_matrix = np.linalg.inv(np.eye(n) + np.matrix(L)) * np.matrix(U)

    norms = {'one' : np.linalg.norm(iter_matrix, ord = 1),
             'inf': np.linalg.norm(iter_matrix, ord = np.inf),
             'fro': np.linalg.norm(iter_matrix, ord = 'fro')}
    return M, iter_matrix, norms

matrix = random_matrices(4)
gauss_seidel_iterations(matrix)
```

[11]:
```
([[2.29556, 6.21535, 4.02451, 4.08206],
  [7.83561, 8.22769, 4.1541, 6.78558],
  [1.65026, 9.71322, 2.44551, 5.6722],
  [7.12227, 2.37694, 4.14573, 5.2087]],
 matrix([[ 0.        ,  2.70755284,  1.75317134,  1.77824147],
         [ 0.        , -2.57852789, -1.16473359, -0.86877686],
         [ 0.        ,  8.41445854,  3.44309573,  4.57011421],
         [ 0.        , -9.22283669, -4.60617871, -5.6725373 ]]),
```

```
{'one': 22.92337596144171,
 'inf': 19.501552696936752,
 'fro': 16.25810632820597})
```

(iii) Repeat part (ii) for the Gauss Jacobi iteration. (1) Deliverable(s): The input matrix, iteration matrix and the three norms obtained

[12]:
```
def gauss_jacobi_iterations(M) :

    n = len(M)
    L = [[0] * n]*n
    U = [[0] * n]*n

    mat = M.copy()
    for i in range(n) :
        L[i] = mat[i][:i] + [0]*(n - i)
        U[i] = [0]*(i+1) + mat[i][i+1:]

#     print(mat)
    iter_matrix = -1.0 * (np.matrix(L) + np.matrix(U))

    norms = {'one' : np.linalg.norm(iter_matrix, ord = 1),
             'inf': np.linalg.norm(iter_matrix, ord = np.inf),
             'fro': np.linalg.norm(iter_matrix, ord = 'fro')}
    return M, iter_matrix, norms

matrix = random_matrices(4)
gauss_jacobi_iterations(matrix)
```

```
[[6.64901, 7.43489, 3.38613, 7.54083], [2.01682, 8.71565, 5.77965, 2.81791],
[9.80488, 7.96192, 7.36545, 3.32262], [2.67719, 8.31644, 3.14664, 4.53993]]
```

[12]:
```
([[6.64901, 7.43489, 3.38613, 7.54083],
  [2.01682, 8.71565, 5.77965, 2.81791],
  [9.80488, 7.96192, 7.36545, 3.32262],
  [2.67719, 8.31644, 3.14664, 4.53993]],
 matrix([[ 0.     , -7.43489, -3.38613, -7.54083],
         [-2.01682,  0.     , -5.77965, -2.81791],
         [-9.80488, -7.96192,  0.     , -3.32262],
         [-2.67719, -8.31644, -3.14664,  0.     ]]),
 {'one': 23.713250000000002, 'inf': 21.08942, 'fro': 20.635397403621766})
```

(iv) Write a function that perform Gauss Seidel iterations. Generate a random 4 × 4 matrix A and a suitable random vector b R^4 and report the results of passing this matrix to the functions written above. Write down the first ten iterates of Gauss Seidel algorithm. Does it converge? Generate a plot of xk+1 −xk 2 for the first 10 iterations. Take a screenshot and paste it in the assignment document. (1) Deliverable(s): The input matrix and the vector, the 10 successive iterates and the plot

[13]:
```
def guass_seidel_iterations(A, b, niter = 10):

    iter_values = [0.0] * len(A)
    results = [deepcopy(iter_values)]

    for each_iter in range(niter):
#         print(iter_values)
        for i in range(len(A)):
            val = []
            for j in range(len(A[i])) :
                if j != i :
                    val.append(A[i][j] * iter_values[j])
            iter_values[i] = (b[i] - sum(val)) / A[i][i]
        results.append(deepcopy(iter_values))

    results = pd.DataFrame(results)
    results['||xk+1 - xk||2'] = (results - results.shift(1)).apply(norm, axis =␣
↪1)
    results['nIter'] = range(0, niter + 1)

    return results

A = random_matrices(4)
b = random_matrices(4)[0]
print('Input Matrix : \n', np.array(A))
print('Vector : \n', np.array(b))
seidel_res = guass_seidel_iterations(A, b, niter= 10)
display(seidel_res)
seidel_res.plot(x = 'nIter', y = '||xk+1 - xk||2')
```

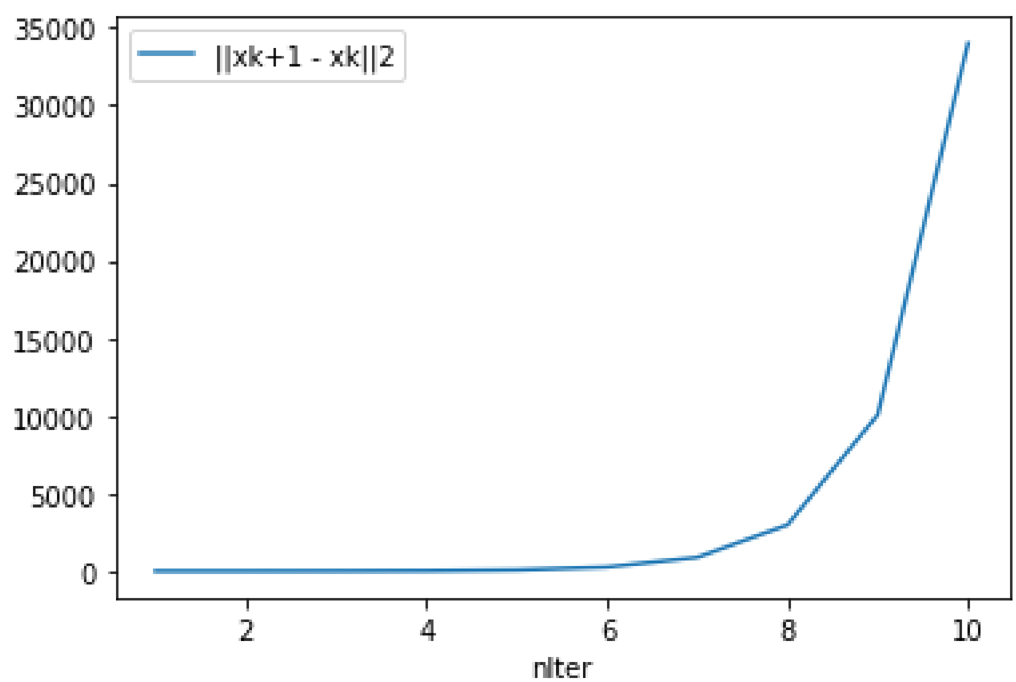
```
Input Matrix :
 [[3.38533 3.89831 8.8751  5.46898]
 [4.90075 9.92805 3.77127 6.34037]
 [6.32863 1.20674 5.1729  9.97752]
 [5.27454 4.04326 5.51683 7.61688]]
Vector :
 [5.32622 7.3108  7.26063 9.75208]
```

```
          0          1          2          3  ||xk+1 - xk||2  \
0  0.000000   0.000000   0.000000   0.000000             NaN
1  1.573324  -0.040256  -0.511855   0.582930        1.754643
```

2	2.019858	-0.438522	-2.089604	1.627872	1.984743
3	4.926657	-1.941405	-7.310738	4.194359	6.674970
4	16.199035	-7.161491	-24.834099	11.851427	22.803831
5	55.780036	-24.933351	-83.881452	36.643583	77.354609
6	190.993905	-85.081849	-283.092567	119.225811	261.544500
7	649.104493	-288.284738	-955.435969	396.830781	883.327173
8	2197.268706	-974.390820	-3224.881728	1332.699227	2982.245426
9	7425.109546	-3290.595967	-10885.509059	4490.546361	10067.389720
10	25074.176697	-11109.381231	-36744.632153	15148.854896	33984.066300

	nIter
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

[13]: <matplotlib.axes._subplots.AxesSubplot at 0x127ca5048>



(v) Repeat part (iv) for the Gauss Jacobi method. (1) Deliverable(s): The input matrix and the vector, the 10 successive iterates and the plot

[9]:
```
def guass_jacobi_iterations(A, b, niter = 10):

    iter_values = [0.0] * len(A)
    results = [deepcopy(iter_values)]

    for each_iter in range(niter):
        iter_buffer = [0.0] * len(A)
        for i in range(len(A)):
            val = []
            for j in range(len(A[i])) :
                if j != i :
                    val.append(A[i][j] * iter_values[j])
            iter_buffer[i] = (b[i] - sum(val)) / A[i][i]
        iter_values = iter_buffer
        results.append(deepcopy(iter_values))

    results = pd.DataFrame(results)
    results['||xk+1 - xk||2'] = (results - results.shift(1)).apply(norm, axis =␣
 ↪1)
    results['nIter'] = range(0, niter + 1)

    return results

A = random_matrices(4)
b = random_matrices(4)[0]
print('Input Matrix : \n', np.array(A))
print('Vector : \n', np.array(b))
jacobi_res = guass_jacobi_iterations(A, b, niter= 10)
display(jacobi_res)
jacobi_res.plot(x = 'nIter', y = '||xk+1 - xk||2')
```

```
Input Matrix :
 [[5.97726 5.86593 3.20262 1.81109]
 [4.18644 8.31401 4.5641  6.22353]
 [1.13456 3.49567 5.22528 2.9911 ]
 [1.56356 8.59559 1.48004 5.84953]]
Vector :
 [7.88136 9.62491 6.70183 3.87868]
```

	0	1	2	3	||xk+1 - xk||2	nIter
0	0.000000	0.000000	0.000000	0.000000	NaN	0
1	1.318557	1.157674	1.282578	0.663075	2.272329	1
2	-0.705670	-0.706715	-0.157757	-1.715030	3.911955	2
3	2.616284	2.883411	2.890318	1.930096	6.819244	3
4	-3.644596	-3.191206	-2.319314	-5.004581	12.301610	4

5	7.209387	8.012323	7.073582	6.913411	21.762193	5
6	-12.429302	-11.530791	-9.600403	-14.827424	38.965332	6
7	22.271143	23.785821	20.182989	23.358385	69.257421	7
8	-39.915811	-38.621638	-32.836664	-45.348666	123.667500	8
9	70.555247	73.229243	61.745898	76.393280	220.185005	9
10	-126.777153	-125.450982	-106.756487	-141.425658	392.749350	10

[9]: <matplotlib.axes._subplots.AxesSubplot at 0x12798ca58>