MATH381 NUMERICAL ANALYSIS

HOMEWORK I

Sinan ÇAVUŞOĞLU 2215846
Burak Kaan ÖZALP 2216265
Burak Can TAŞDEMİR 2216364

QUESTION I:

a)

```
def jacobi_iterative_method(n,aug_mat,start_vec):
    """
    n: The number of equations n
    aug_mat: The augmented matrix
    start_vec: The starting vector
    """
    import numpy as np
    from numpy.linalg import inv

    coeff_mat = aug_mat[:,0:n]  # Coefficient Matrix
    b = aug_mat[:,n:]           # constant

    diag = np.diag(np.diag(coeff_mat)) # diagonal matrix
    diag_inv = inv(diag)               # inverse of diagonal matrix

    x_k = ((np.identity(n) - diag_inv @ coeff_mat) @ start_vec) + (diag_inv @ b)
# @ is matrix multiplication
    relex = abs(x_k - start_vec).max() / abs(x_k).max()
# Relative Error
    return x_k, relex
```

First of all, we defined Jacobi iterative method which has inputs **n**(The number of equations n), **aug_mat**(The augmented matrix), **start_vec(**The starting vector**).** Then, we imported "numpy" dictionary(The fundamental package for scientific computing with Python). Then, we divided augmented matrix in two parts which are **coeff_mat(**coefficients matrix) and **b**(right hand side of the equation). After that, we take diagonal part of the coefficient matrix **diag.** Thanks to numpy, find easily inverse of diagonal called **diag_inv.** Then, we wrote iteration algorithm which is $x^{(k)} = (I - D^{-1}A)x^{(k-1)} + D^{-1}b \;\; k \geq 1$. In the equation, $x^{(k)}$ defined by **x_k** and $x^{(k-1)}$defined by **start_vec**.

x_k is column vector after iterate Jacobi method one times and **relex** is relative error after one iteration.

Then, we selected **start_vec1**(starting vector), and write **aug_mat1(**augmented matrix**)** from equation, **headers** is for table.

```
start_vec1 = np.array([1,1,1,1]).reshape((4,1)) # x_0 = [1,1,1,1]
aug_mat1 = np.array([[2,-1,2,-1,-1],    # [x,y,z,w,b] Augmented Matrix
                     [2,1,-2,-2,-2],
                     [-1,2,-4,1,1],
                     [3,0,0,-3,-3]])
headers=["vec1","vec2","vec3","vec4","vec5","vec6","vec7","vec8","vec9","vec10"]
```

After defined function and writed variables, we start ten times iterations:

```
import tabulate as tabulate
vector_list= []
rele_list = []
for k in range(1,11):
    (start_vec1, relex) = jacobi_iterative_method(4,aug_mat1,start_vec1)
    vector_list.append(start_vec1)
    rele_list.append(relex)
result = np.hstack(vector_list)
print(tabulate(result, headers, tablefmt="github"))
print(rele_list)
```

and vectors that we found: (Thanks to tabulate library, we show vectors in tabular data)

vec1	vec2	vec3	vec4	vec5	vec6	vec7	vec8	vec9	vec10
-0.5	0.25	1.125	-1.8125	1.84375	0.015625	-2.74219	5.04297	-4.33789	-0.350586
0	3.5	-0.75	1.375	4.6875	-4.46875	6.51562	1.96094	-7.67578	16.8301
0.25	0.375	1.5625	-0.59375	1.42188	1.42969	-1.77734	3.94727	-0.96582	-1.49268
2	0.5	1.25	2.125	-0.8125	2.84375	1.01562	-1.74219	6.04297	-3.33789

And relative error is

```
[0.75, 1.0, 2.72, 1.382, 0.78, 2.049, 1.686, 1.544, 1.255, 1.46]
```

b)

```
def GaussSeidel_iterative_method(n,aug_mat,start_vec):
    """
    n: The number of equations n
    aug_mat: The augmented matrix
    start_vec: The starting vector
    """
    import numpy as np
    from numpy.linalg import inv

    coeff_mat = aug_mat[:,0:n]  # Coefficient Matrix
    b = aug_mat[:,n:]           # constant

    lower_tri = np.tril(coeff_mat, k = -1)  # Lower Triangular Matrix
    diag = np.diag(np.diag(coeff_mat))      # Diagonal Matrix
    upper_tri = np.triu(coeff_mat, k = 1)   # Upper Triangular Matrix


    x_k = inv(diag + lower_tri) @ ((- upper_tri @ start_vec) + b) # @ is matrix
multiplication
    relex = abs(x_k - start_vec).max() / abs(x_k).max()           # Relative Error

    return x_k, relex
```

First of all, we defined Gauss-Seidel iterative method which has inputs **n**(The number of equations n), **aug_mat**(The augmented matrix), **start_vec(**The starting vector**).** Then, we imported "numpy" dictionary. Then, we divided augmented matrix in two parts which are **coeff_mat(**coefficients matrix) and **b**(right hand side of the equation). Then, we take lower triangular part of the coefficient matrix as **lower_tri,** upper triangular part of the coefficient matrix as **upper_tri** and we take diagonal part of the coefficient matrix **diag.**

x_k is column vector after iterate Gauss-Seidel method one times and **relex** is relative error after one iteration.

After defined function, we start ten times iterations with variables that we defined first example above

```
import tabulate as tabulate
liste = []
rele_list = []
for k in range(1,11):
    (start_vec1, relex) = GaussSeidel_iterative_method(4,aug_mat1,start_vec1)
    liste.append(start_vec1)
    rele_list.append(relex)
result = np.hstack(liste)
print (tabulate(result, headers, tablefmt="github"))
print(np.around(np.array(rele_list), 4))
```

Vectors:

vec1	vec2	vec3	vec4	vec5	vec6	vec7	vec8	vec9	vec10
-0.5	-0.375	-0.15625	-0.0234375	0.0214844	0.0219727	0.0111084	0.00283813	-0.000648499	-0.00119591
3	3	2.5	2.125	1.96875	1.94531	1.9668	1.98877	1.99988	2.00272
1.625	1.46875	1.19531	1.0293	0.973145	0.972534	0.986115	0.996452	1.00081	1.00149
0.5	0.625	0.84375	0.976562	1.02148	1.02197	1.01111	1.00284	0.999352	0.998804

Relative Error:

```
[0.6667 0.0521 0.2    0.1765 0.0794 0.012  0.0109 0.011  0.0056 0.0014]
```

As a result; For the equation, Jacobi method does not converge to solutions, but Gauss-Seidel method converges to solution since relative error after ten iteration is 0.0014 and vectors satisfy the equation.

QUESTION II:

First define variable:

```
aug_mat2 = np.array([[1,0,1,2],
                     [1,-1,0,0],
                     [1,2,-3,0]])

start_vec2 = np.array([0,0,0]).reshape((3,1))

tolerance = 10 ** (-3)
relex_tolerance = 1000 # relative error for pass tolerance
```

Start iteration with Jacobi method,

```
k = 1  # index for jacobi
while True:
    (start_vec2, relex) = jacobi_iterative_method(3,aug_mat2,start_vec2)
    print("Iteration {}".format(k))
    print("x_{}".format(k))
    print(start_vec2)
    print("Relative Error: {}".format(relex))

    relex_tolerance = relex
    k += 1
    if abs(relex_tolerance) < tolerance:
        break
```

The last three iterations are

```
Iteration 125
x_125
[[1.00047153]
 [0.99926452]
 [0.99899155]]
Relative Error: 0.001206441469564971
Iteration 126
x_126
[[1.00100845]
 [1.00047153]
 [0.99966685]]
Relative Error: 0.0012057943571702778
Iteration 127
x_127
[[1.00033315]
 [1.00100845]
 [1.0006505 ]]
Relative Error: 0.0009826566026146842
```

Relative error of last iteration(127) is under the 10^-3. Therefore, Jacobi method converges in this equation.

For the Gauss-Seidel method:

```
j = 1
while True:
    (start_vec2, relex) = GaussSeidel_iterative_method(3,aug_mat2,start_vec2)
    print("Iteration {}".format(j))
    print("x_{}".format(j))
    print(start_vec2)
    print("Relative Error: {}".format(relex))

    relex_tolerance = relex
    j += 1
    if abs(relex_tolerance) < tolerance:
        break

    if j == 1000:
        break
```

We added new escape code which means after 1000 iterations, iteration break since Gauss-Seidel method does not converge

```
Iteration 9996
x_9996
[[0.]
 [0.]
 [0.]]
Relative Error: inf
Iteration 9997
x_9997
[[2.]
 [2.]
 [2.]]
Relative Error: 1.0
Iteration 9998
x_9998
[[0.]
 [0.]
 [0.]]
Relative Error: inf
Iteration 9999
x_9999
[[2.]
 [2.]
 [2.]]
Relative Error: 1.0
```

Only vectors we can find are [0 0 0] and [2 2 2] after 1000 iterations.

Therefore, Jacobi method converges to solution [1 1 1] but Gauss-Seidel method does not converge to solution.