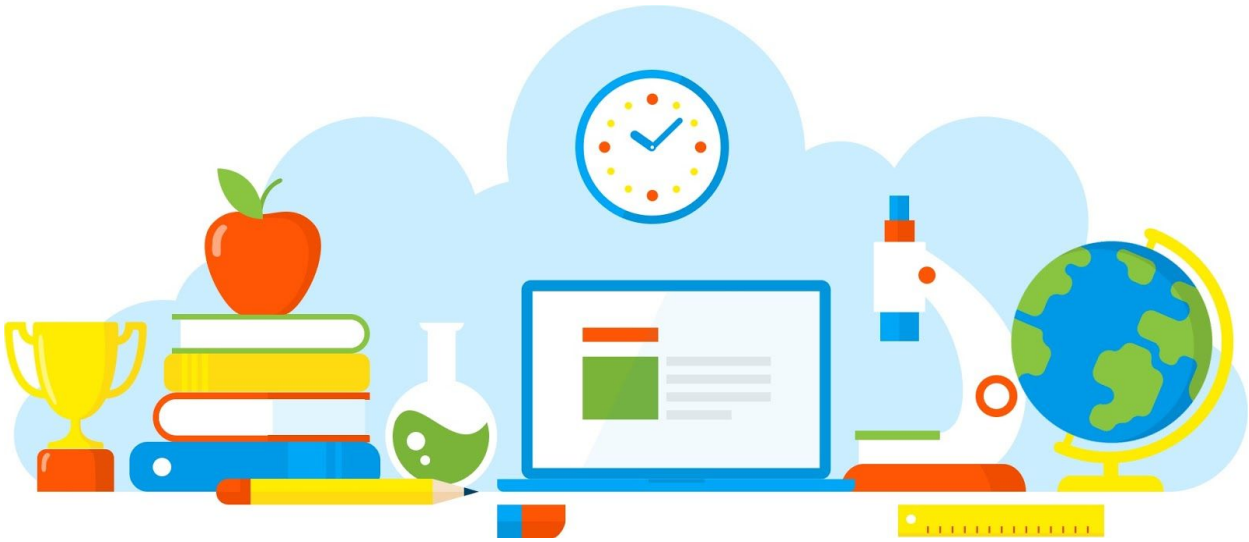

Math Systems HW1 (Optional Question)

Teacher: Dr. Ebrahimi

Mahdi Sadeghi
830598035

Jacobi

To solve linear equation system using jacobi method, we first convert A and b Matrices to the form of C and d form where we have:

$$Ax = b \Rightarrow x = Cx + d$$

Where $C_{ii} = 0$

```
def generate_cd_matrices(a, b):
    """
    generates c and d matrices from a and b
    :param a:
    :param b:
    :return: c, d
    """
    n = a.shape[0]
    c = a.copy()
    d = b.copy()
    for i in range(n):
        denom = a[i, i] if a[i, i] > 0 else EPSILON
        c[i, i] = 0
        c[i, :] *= 1 / denom
        d[i, :] *= 1 / denom

    return c, d
```

Before converting to C and d we should rearrange rows of A in such a way that it becomes diagonally dominant. The code for doing it is as follows:

```
def rearrange_ab(a, b):
    """
    rearranges a and b to become diagonally dominent
    :param a:
    :param b:
    :return:
    """
    n = a.shape[0]
    permutation = [-1] * n
    for i in range(n):
        row = a[i, :]
        for j in range(n):
            summation = row.sum() - row[j]
            if row[j] >= summation:
                if j not in permutation:
                    permutation[i] = j

    if -1 in permutation:
        return None, None

    rearranged_a = np.zeros(a.shape)
    rearranged_b = np.zeros(b.shape)

    for i, p in enumerate(permutation):
        rearranged_a[p, :] = a[i, :]
        rearranged_b[p, :] = b[i, :]

    return rearranged_a, rearranged_b
```

No for jacobi method we use following formula:

$$
\begin{cases}
x_1^{new} = (b_1 - a_{12}x_2^{old} - a_{13}x_3^{old} - a_{14}x_4^{old}) / a_{11} \\
x_2^{new} = (b_2 - a_{21}x_1^{old} - a_{23}x_3^{old} - a_{24}x_4^{old}) / a_{22} \\
x_3^{new} = (b_3 - a_{31}x_1^{old} - a_{32}x_2^{old} - a_{34}x_4^{old}) / a_{33} \\
x_4^{new} = (b_4 - a_{41}x_1^{old} - a_{42}x_2^{old} - a_{43}x_3^{old}) / a_{44}
\end{cases}
$$

```python
def jacobi(c, d):
    """
    solve equation with jacobi method
    :param n:
    :param c:
    :param d:
    :return:
    """
    n = c.shape[0]
    x_old = np.zeros(shape=(n, 1))
    x_new = np.zeros(shape=(n, 1))

    while True:
        for i in range(n):
            current_x = d[i, 0]

            for j in range(n):
                current_x = current_x - (c[i, j] * x_old[j, 0])

            x_new[i, 0] = current_x

        if check_convergence(x_old, x_new):
            break

        x_old = x_new.copy()

    return x_new
```

Gauss-Seidel

The only difference between gauss-seidel is that in each iteration we used only the old x values from previous iteration in jacobi where as in gauss we should use the new x values obtained in the current iteration if they are available:

$$\begin{cases} x_1^{new} = (b_1 - a_{12}x_2^{old} - a_{13}x_3^{old} - a_{14}x_4^{old}) / a_{11} \\ x_2^{new} = (b_2 - a_{21}x_1^{new} - a_{23}x_3^{old} - a_{24}x_4^{old}) / a_{22} \\ x_3^{new} = (b_3 - a_{31}x_1^{new} - a_{32}x_2^{new} - a_{34}x_4^{old}) / a_{33} \\ x_4^{new} = (b_4 - a_{41}x_1^{new} - a_{42}x_2^{new} - a_{43}x_3^{new}) / a_{44} \end{cases}$$

```
def gauss_seidel(c, d):
    """
    solve equation with gauss-seidel method
    :param c:
    :param d:
    :return:
    """
    n = c.shape[0]
    x_old = np.zeros(shape=(n, 1))
    x_new = np.zeros(shape=(n, 1))

    while True:
        for i in range(n):
            current_x = d[i, 0]

            for j in range(n):
                if j <= i:
                    current_x = current_x - (c[i, j] * x_new[j, 0])
                else:
                    current_x = current_x - (c[i, j]) * x_old[j, 0]

            x_new[i, 0] = current_x
```

```
        if check_convergence(x_old, x_new):
            break

        x_old = x_new.copy()

    return x_new
```

SOR

In SOR we use following formula:

$$\begin{cases} x_1^{new} = (1-\lambda)x_1^{old} + \lambda(b_1 - a_{12}x_2^{old} - a_{13}x_3^{old} - a_{14}x_4^{old})/a_{11} \\ x_2^{new} = (1-\lambda)x_2^{old} + \lambda(b_2 - a_{21}x_1^{new} - a_{23}x_3^{old} - a_{24}x_4^{old})/a_{22} \\ x_3^{new} = (1-\lambda)x_3^{old} + \lambda(b_3 - a_{31}x_1^{new} - a_{32}x_2^{new} - a_{34}x_4^{old})/a_{33} \\ x_4^{new} = (1-\lambda)x_4^{old} + \lambda(b_4 - a_{41}x_1^{new} - a_{42}x_2^{new} - a_{43}x_3^{new})/a_{44} \end{cases}$$

```
def sor(c, d, landa=SOR_LANDA):
    """
    solve equation with sor method
    :param landa:
    :param c:
    :param d:
    :return:
    """
    n = c.shape[0]
    x_old = np.zeros(shape=(n, 1))
    x_new = np.zeros(shape=(n, 1))

    while True:
        for i in range(n):
            current_x = ((1 - landa) * x_old[i, 0]) + (landa * d[i,
0])

            for j in range(n):
                if j <= i:
                    current_x = current_x - (landa * c[i, j] *
x_new[j, 0])
                else:
                    current_x = current_x - (landa * c[i, j]) *
x_old[j, 0]

            x_new[i, 0] = current_x
```

```
        if check_convergence(x_old, x_new):
            break

        x_old = x_new.copy()

    return x_new
```