In [1]:
```
import numpy as np
```

Problem 1:

The machine epsilon is found to be $2.220446049250313 \times 10^{-16}$. The program used to find this value of machine epsilon is provided below.

In [2]:
```
# 1. Finding the machine epsilon

eps = 1
while 1+eps>1 and 1-eps<1:
    eps = eps/2
machine_epsilon = 2*eps

print("The machine epsilon is", machine_epsilon)
```

```
The machine epsilon is 2.220446049250313e-16
```

Problem 2:

We want to write a program to solve $Ax = b$ for

$$A = \begin{bmatrix} 54 & 14 & -11 & 2 \\ 14 & 50 & -4 & 29 \\ -11 & -4 & 55 & 22 \\ 2 & 29 & 22 & 95 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

a) L-U Decomposition:

We will first solve it using the L-U decomposition. The following provides the code for the L-U decomposition of a non-singular square matrix A.

In [3]:
```
def lu_factor(A):
    # Returns the LU factorization of a non singular square matrix A
    U = A.copy()
    n = A.shape[0]
    L = np.identity(n)
    for i in range(n-1):
        L_n = np.identity(n)
        for j in range(n-(i+1)):
            l = U[j+(i+1), i]/U[i, i]
            L_n[j+(i+1), i] = -l
        inv_L_n = np.linalg.inv(L_n)
        U = np.matmul(L_n, U)
        L = np.matmul(L, inv_L_n)
    return L, U
```

The above code only computes the L-U factorization but does not solve the linear problem. We need to use back-substitution to be able to solve for $Ax = b$. We have to be careful because we want the backward substitution to work for both lower and upper

triangular matrices. The following code contains a backsubstitution algorithm for both lower and upper triangular matrices.

In [4]:
```
def backsub(A,b):
    n = A.shape[0]
    x = np.zeros(n)
    # Check if matrix is lower or upper triangular (assuming we only input lower
    lower = False
    above_diag = 0
    for i in range(n-1):
        above_diag += np.sum(A[i,i+1:n])
    if above_diag == 0:
        lower = True

    # If matrix is lower triangular
    if lower == True:
        x[0] = b[0]/A[0,0]
        for k in range(1,n):
            a = A[k, 0:k]
            x_vec = x[0:k]
            val = np.dot(a, x_vec)
            x[k] = (b[k]-val)/A[k,k]

    # If matrix is upper triangular
    else:
        x[-1] = b[-1]/A[-1,-1]
        len = list(range(n-1))
        len.reverse()
        for i in len:
            a = A[i,i+1:]
            x_vec = x[i+1:]
            val = np.dot(a,x_vec)
            x[i]= (b[i]-val)/A[i,i]
    return x
```

We are now fully equipped to solve the system $Ax = b$. We will first compute the L-U factorization of $A : A = LU$. Then, we solve the following system for y using back-substitution: $Ly = b$. Finally, we solve $Ux = y$ for x using backsubstituion to get the solution to $Ax = b$. The code for it is provided below. Using the code, we find that:

$$x_{lu} = \begin{bmatrix} 0.01893441 \\ 0.01680508 \\ 0.02335523 \\ -0.00041085 \end{bmatrix}$$

In [5]:
```
A = np.array([[54., 14., -11., 2.],
[14., 50., -4., 29.],
[-11., -4., 55., 22.],
[2., 29., 22., 95.]])
n = A.shape[0]
b = np.ones(n)

# 2a) Solving the problem using LU decomposition

L, U = lu_factor(A)
```

```
y = backsub(L,b)
x_lu = backsub(U,y)
print("The solution to the problem using LU factorization is", x_lu)
```

```
The solution to the problem using LU factorization is [ 0.01893441  0.01680508
0.02335523 -0.00041085]
```

b) Gauss-Jacobi Iteration:

We want to solve $Ax = b$ using Gauss-Jacobi iteration. Since, the solution to this is not exact we stop our iteration when the solution from the iteration agrees with the L-U decomposition solution to four significant digits. The following code creates a function to conduct Gauss-Jacobi iteration and it returns the number of iterations needed to reach agreement to four significant digits and the solution x. After feeding in matrix A and vector b from our problem, we find that the solution to the problem using Gauss-Jacobi iteration is

$$x_{gj} = \begin{bmatrix} 0.01893441 \\ 0.01680507 \\ 0.02335523 \\ -0.00041085 \end{bmatrix}$$

and it required 25 iterations to agree with LU decomposition solution to 4 s.d.

In [6]:

```
def gauss_jacobi(A, b, x_true):
    n = A.shape[0]
    # initial guess
    x_guess = np.zeros(n)
    counter = 0
    while np.max(np.abs(x_guess-x_true))>0.00000001:
        new_x_guess = np.zeros(n)
        counter += 1
        for i in range(n):
            a = A[i,:]
            a = np.delete(a,i)
            x_guess_copy = np.delete(x_guess, i)
            val = np.dot(a,x_guess_copy)
            new_x_guess[i] = (1/A[i,i])*(b[i]-val)
        x_guess = new_x_guess
    return x_guess, counter


# 2b) Gauss-Jacobi

x_gj, counter_gj = gauss_jacobi(A,b,x_lu)

print("The solution to the problem using Gauss-Jacobi iteration is", x_gj, "and
```

```
The solution to the problem using Gauss-Jacobi iteration is [ 0.01893441  0.0168
0507  0.02335523 -0.00041085] and it required 25 iterations to agree with LU dec
omposition solution to 4 s.d.
```

c) Gauss-Seidel Iteration

Finally, we will solve the system $Ax = b$ using Gauss-Seidel iteration. The solution, again, is not going to be exact and so we stop iterating when we have agreement to four significant digits with the solution from L-U decomposition. The following code contains the code for the Gauss-Seidel iteration and it returns the number of iterations needed to reach desired level of tolerance along with the solution x. The solution to our system $Ax = b$ is

$$x_{gs} = \begin{bmatrix} 0.01893441 \\ 0.01680507 \\ 0.02335523 \\ -0.00041085 \end{bmatrix}$$

and it required 12 iterations to agree with LU decomposition solution to 4 s.d. This makes sense because it should take fewer iterations for Gauss-Seidel because it uses new information more than Gauss-Jacobi.

In [7]:

```
def gauss_seidel(A, b, x_true):
    n = A.shape[0]
    # initial guess
    x_guess = np.zeros(n)
    counter = 0
    while np.max(np.abs(x_guess-x_true))>0.00000001:
        new_x_guess = np.zeros(n)
        counter += 1
        for i in range(n):
            if i ==0:
                a = A[i, 1:]
                x_guess_copy = x_guess[1:]
                val = np.dot(a, x_guess_copy)
                new_x_guess[i] = (1/A[i,i])*(b[i]-val)
            else:
                a_new = A[i,0:i]
                a_old = A[i, i+1:]
                new_x_copy = new_x_guess[0:i]
                x_guess_copy = x_guess[i+1:]
                val_new = np.dot(a_new, new_x_copy)
                val_old = np.dot(a_old, x_guess_copy)
                new_x_guess[i] = (1/A[i,i])*(b[i]-val_new-val_old)
        x_guess = new_x_guess
    return x_guess, counter

# 2c) Gauss-Seidel

x_gs, counter_gs = gauss_seidel(A,b,x_lu)

print("The solution to the problem using Gauss-Seidel iteration is", x_gs, "and
```

```
The solution to the problem using Gauss-Seidel iteration is [ 0.01893441  0.0168
0507  0.02335523 -0.00041085] and it required 12 iterations to agree with LU dec
omposition solution to 4 s.d.
```

Problem 3:

We want to compute the solution to the following non-linear problem:

$$f_1(x_1, x_2) = x_1^{0.2} + x_2^{0.2} - 2 = 0$$

$$f_2(x_1, x_2) = x_1^{0.1} + x_2^{0.4} - 2 = 0$$

Clearly, a solution to the problem is $(x_1, x_2) = (1, 1)$.

In the following piece of code, we define the functions $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$. We also compute the partial derivatives: $\frac{\partial f_1(x_1,x_2)}{\partial x_1}$, $\frac{\partial f_1(x_1,x_2)}{\partial x_2}$, $\frac{\partial f_2(x_1,x_2)}{\partial x_1}$ and $\frac{\partial f_2(x_1,x_2)}{\partial x_2}$. This will help us compute the Jacobian.

In [8]:

```
def f1(x1,x2):
    return x1**(0.2) +x2**(0.2) -2

def f2(x1,x2):
    return x1**(0.1) +x2**(0.4)-2

def f1_1(x1,x2):
    return 0.2*(x1**(-0.8))

def f1_2(x1,x2):
    return 0.2*(x2**(-0.8))

def f2_1(x1,x2):
    return 0.1*(x1**(-0.9))

def f2_2(x1,x2):
    return 0.4*(x2**(-0.6))
```

Gauss-Jacobi:

We will solve the non-linear problem using the Gauss-Jacobi algorithm. The following piece of code contains the Gauss-Jacobi algorithm:

In [9]:

```
def gauss_jacobi_nonlinear(x_0):
    max_iterations = 500
    x_guess = x_0
    tol = 1e-4
    counter = 0
    diff = 1000
    while diff > tol:
        counter +=1
        if counter > max_iterations:
            print("We passed the maximum number of iterations and the method did
            break
        else:
            x1 = x_guess[0]
            x2 = x_guess[1]
            new_guess = np.zeros(2)
            new_guess[0] = x1-(f1(x1,x2)/f1_1(x1,x2))
            new_guess[1] = x2 - (f2(x1,x2)/f2_2(x1,x2))
            diff = np.max(np.abs(x_guess-new_guess))
            x_guess = new_guess
```

```
    return x_guess, counter
```

We will now solve our problem using Gauss-Jacobi algorithm and two sets of initial points: $(x_1^0, x_2^0) = (2, 2)$ and $(x_1^0, x_2^0) = (3, 3)$.

In [10]:

```
x_guess_1 = np.array([2,2])
x_guess_2 = np.array([3,3])

x_gj_1, counter_gj_1 = gauss_jacobi_nonlinear(x_guess_1)
x_gj_2, counter_gj_2 = gauss_jacobi_nonlinear(x_guess_2)
```

```
/var/folders/xd/3hdyv6y13hsbb53tkk_tz62r0000gn/T/ipykernel_96384/3391074956.py:
2: RuntimeWarning: invalid value encountered in double_scalars
  return x1**(0.2) +x2**(0.2) -2
/var/folders/xd/3hdyv6y13hsbb53tkk_tz62r0000gn/T/ipykernel_96384/3391074956.py:
8: RuntimeWarning: invalid value encountered in double_scalars
  return 0.2*(x1**(-0.8))
/var/folders/xd/3hdyv6y13hsbb53tkk_tz62r0000gn/T/ipykernel_96384/3391074956.py:
5: RuntimeWarning: invalid value encountered in double_scalars
  return x1**(0.1) +x2**(0.4)-2
/var/folders/xd/3hdyv6y13hsbb53tkk_tz62r0000gn/T/ipykernel_96384/3391074956.py:1
7: RuntimeWarning: invalid value encountered in double_scalars
  return 0.4*(x2**(-0.6))
```

We run into some runtime errors above. This maybe due to the fact that we get complex roots in the process. For example, when we compute $0.5^{\frac{3}{5}}$ we can get five possible roots out of which four are complex. We have to find a way to only get positive roots. The following piece of code redefines the functions so as to pick the positive square root.

In [11]:

```
def f1(x1,x2):
    sign_1 = np.sign(x1)
    sign_2 = np.sign(x2)
    return(np.abs(x1)**.2 * sign_1 + np.abs(x2)**.2 * sign_2 - 2)

def f2(x1,x2):
    sign_1 = np.sign(x1)
    sign_2 = np.sign(x2)
    return(np.abs(x1)**.1 * sign_1 + np.abs(x2)**.4 * sign_2 - 2)

def f1_1(x1,x2):
    return(.2*np.abs(x1)**(-.8))

def f2_2(x1,x2):
    sign_1 = np.sign(x1)
    return(.4*np.abs(x1)**(-.6) * sign_1)

def f1_2(x1,x2):
    return(.2*np.abs(x2)**(-.8))

def f2_1(x1,x2):
    sign_2 = np.sign(x2)
    return(.1*np.abs(x2)**(-.9) * sign_2)
```

We will use the newly defined functions to solve for the problem using Gauss-Jacobi

iterations with the two sets of initial points. The iterations did not converge for both the intial points and we were not able to find a solution.

In [12]:
```
x_gj_1, counter_gj_1 = gauss_jacobi_nonlinear(x_guess_1)

x_gj_2, counter_gj_2 = gauss_jacobi_nonlinear(x_guess_2)
```

```
We passed the maximum number of iterations and the method did not converge.
We passed the maximum number of iterations and the method did not converge.
```

We did not converge to a solution using Gauss-Jacobi algorithm with the two starting values provied. We instead try using $(x_1^0, x_2^0) = (0.5, 0.5)$ and $(x_1^0, x_2^0) = (1.5, 1.5)$ as intial guesses. Our algorithm converges and the solution is summarized below:

The solution to the problem using Gauss-Jacobi algorithm and initial guess of $(x_1^0, x_2^0) = (0.5, 0.5)$ is

$$x = \begin{bmatrix} 0.99998944 \\ 0.99998397 \end{bmatrix}$$

and it converged in 14 iterations.

The solution to the problem using Gauss-Jacobi algorithm and initial guess of $(x_1^0, x_2^0) = (1.5, 1.5)$ is

$$x = \begin{bmatrix} 0.99995378 \\ 1.0000159 \end{bmatrix}$$

and it converged in 14 iterations.

In [13]:
```
x_guess_3 = np.array([0.5, 0.5])
x_guess_4 = np.array([1.5, 1.5])

x_gj_3, counter_gj_3 = gauss_jacobi_nonlinear(x_guess_3)

print("The solution to the problem using Gauss-Jacobi algorithm and initial gues

x_gj_4, counter_gj_4 = gauss_jacobi_nonlinear(x_guess_4)

print("The solution to the problem using Gauss-Jacobi algorithm and initial gues
```

```
The solution to the problem using Gauss-Jacobi algorithm and initial guess of
(0.5,0.5) is [0.99998944 0.99998397] and it converged in 14 iterations
The solution to the problem using Gauss-Jacobi algorithm and initial guess of
(1.5,1.5) is [0.99995378 1.0000159 ] and it converged in 14 iterations
```

Gauss-Seidel:

We will solve the non-linear problem using the Gauss-Seidel algorithm. The following piece of code contains the Gauss-Seidel algorithm:

In [14]:
```
def gauss_seidel_nonlinear(x_0):
    max_iterations = 500
```

```
    x_guess = x_0
    tol = 1e-4
    counter = 0
    diff = 1000
    while diff > tol:
        counter +=1
        if counter > max_iterations:
            print("We passed the maximum number of iterations and the method did
            break
        else:
            x1 = x_guess[0]
            x2 = x_guess[1]
            new_guess = np.zeros(2)
            new_guess[0] = x1-(f1(x1,x2)/f1_1(x1,x2))
            new_guess[1] = x2 - (f2(new_guess[0],x2)/f2_2(new_guess[0],x2))
            diff = np.max(np.abs(new_guess-x_guess))
            x_guess = new_guess
    return x_guess, counter
```

We will now solve our problem using Gauss-Seidel algorithm and two sets of initial points: $(x_1^0, x_2^0) = (2, 2)$ and $(x_1^0, x_2^0) = (3, 3)$. We will still use the newly defined functions to ensure we do not run into a problem of encountering complex roots. The Gauss-Seidel iteration also fails to converge to a solution with the two intial points given in the problem.

In [15]:
```
x_gs_1, counter_gs_1 = gauss_seidel_nonlinear(x_guess_1)

x_gs_2, counter_gs_2 = gauss_seidel_nonlinear(x_guess_2)
```

```
We passed the maximum number of iterations and the method did not converge.
We passed the maximum number of iterations and the method did not converge.
```

We did not converge to a solution using Gauss-Seidel algorithm with the two starting values provied. We instead try using $(x_1^0, x_2^0) = (0.5, 0.5)$ and $(x_1^0, x_2^0) = (1.5, 1.5)$ as intial guesses. Our algorithm converges and the solution is summarized below:

The solution to the problem using Gauss-Seidel algorithm and initial guess of $(x_1^0, x_2^0) = (0.5, 0.5)$ is

$$x = \begin{bmatrix} 0.99998008 \\ 1.00000498 \end{bmatrix}$$

and it converged in 9 iterations.

The solution to the problem using Gauss-Seidel algorithm and initial guess of $(x_1^0, x_2^0) = (1.5, 1.5)$ is

$$x = \begin{bmatrix} 0.99999022 \\ 1.00000244 \end{bmatrix}$$

and it converged in 11 iterations.

In [16]:
```
x_gs_3, counter_gs_3 = gauss_seidel_nonlinear(x_guess_3)

print("The solution to the problem using Gauss-Seidel algorithm and initial gues
```

```
x_gs_4, counter_gs_4 = gauss_seidel_nonlinear(x_guess_4)

print("The solution to the problem using Gauss-Seidel algorithm and initial gues
```

```
The solution to the problem using Gauss-Seidel algorithm and initial guess of
(0.5,0.5) is [0.99998008 1.00000498] and it converged in 9 iterations
The solution to the problem using Gauss-Seidel algorithm and initial guess of
(1.5,1.5) is [0.99999022 1.00000244] and it converged in 11 iterations
```

Newton's Method:

We will now solve the problem using Newton's method. We will need to compute the Jacobian for Newton's method. The following piece of code contains the algorithm for Newton's method and a function to calculate the Jacobian using the functions we had defined earlier. We will continue using the newly defined functions to ensure we are only accounting for positive roots.

In [17]:

```
def jacobian(x1, x2):
    J = np.eye(2)
    J[0,0] = f1_1(x1, x2)
    J[0,1] = f1_2(x1, x2)
    J[1,0] = f2_1(x1, x2)
    J[1,1] = f2_2(x1, x2)
    return J

def newton(x_0):
    max_iterations = 500
    eps = 1e-4
    delta = 1e-4
    diff = 1000
    counter = 0
    while diff > eps:
        counter +=1
        if counter > max_iterations:
            print("The maximum number of iterations has been reached and we did
            break
        else:
            x1 = x_0[0]
            x2 = x_0[1]
            J = jacobian(x1, x2)
            J_inv = np.linalg.inv(J)
            f = np.zeros((2,))
            f[0] = f1(x1,x2)
            f[1] = f2(x1,x2)
            val = np.matmul(J_inv, f)
            x_new = x_0-val
            diff = np.max(np.abs(x_new-x_0))-eps*(np.max(np.abs(x_new)))
            x_0 = x_new
    f = np.array([f1(x_0[0], x_0[1]),f2(x_0[0], x_0[1]) ])
    if np.max(np.abs(f)) > delta:
        print("Failure")
    else:
        print("Success")

    return x_0, counter
```

We will now solve the problem using Newton's method and two sets of initial points: $(x_1^0, x_2^0) = (2, 2)$ and $(x_1^0, x_2^0) = (3, 3)$.

$(x_1^0, x_2^0) = (2, 2)$: Newton's method gives the following solution in 5 iterations:

$$x = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

$(x_1^0, x_2^0) = (3, 3)$: Newton's method exceeds the maximum number of iterations without converging. This is consistent with what was found in the textbook.

In [18]:

```
x_newton1,iterations_newt1 = newton(x_guess_1)

print("The solution from Newton's method with the first set of initial values is
```

```
Success
The solution from Newton's method with the first set of initial values is [1.
1.] and it converged in 5 iterations.
```

In [19]:

```
x_newton2,iterations_newt2 = newton(x_guess_2)

print("The solution from Newton's method with the second set of initial values i
```

```
The maximum number of iterations has been reached and we did not converge.
Failure
The solution from Newton's method with the second set of initial values is [-3.7
8039133e+267  3.78039133e+267] and it converged in 501 iterations.
```

Broyden's method:

Finally, we will now solve the problem using Broyden's method. The following piece of code contains the algorithm for Broyden's method:

In [20]:

```
def f(x):
    x1 = x[0]
    x2 = x[1]
    return np.array([f1(x1, x2), f2(x1, x2)])


def broyden(x_0):
    max_iterations = 500
    eps = 1e-4
    delta = 1e-4
    diff = 1000
    counter = 0
    # Initial guess of Jacobian
    J_guess = jacobian(x_0[0], x_0[1])
    while diff > eps:
        counter +=1
        if counter > max_iterations:
            print("The maximum number of iterations has been reached and we did
            break
        else:
            J_guess_inv = np.linalg.inv(J_guess)
            s_k = -np.matmul(J_guess_inv, f(x_0))
```

```
                new_guess = x_0 + s_k
                diff = np.max(np.abs(x_0-new_guess))-eps*np.max(np.abs(new_guess))
                y_k = f(new_guess)-f(x_0)
                val_k = np.matmul(J_guess, s_k)
                num_k = np.matmul((y_k-val_k), np.transpose(s_k))
                denom_k = np.dot(s_k, s_k)
                J_guess = J_guess + num_k/denom_k
                x_0 = new_guess
        f_x = np.array([f1(x_0[0], x_0[1]),f2(x_0[0], x_0[1]) ])
        if np.max(np.abs(f_x)) > delta:
            print("Failure")
        else:
            print("Success")
        return x_0, counter
```

We will now solve the problem using Broyden's method and two sets of initial points: $(x_1^0, x_2^0) = (2, 2)$ and $(x_1^0, x_2^0) = (3, 3)$.

$(x_1^0, x_2^0) = (2, 2)$**: Broyden's method gives the following solution in 10 iterations:**

$$x = \begin{bmatrix} 1.00007144 \\ 0.99995617 \end{bmatrix}.$$

$(x_1^0, x_2^0) = (3, 3)$**: Broyden's method gives the following solution in 25 iterations:**

$$x = \begin{bmatrix} 1.000082 \\ 0.99991826 \end{bmatrix}.$$

In [21]:

```
x_broyd1,iterations_broyd1 = broyden(x_guess_1)

print("The solution from Broyden's method with the first set of initial values i
```

```
Success
The solution from Broyden's method with the first set of initial values is [1.00
007144 0.99995617] and it converged in 10 iterations.
```

In [22]:

```
x_broyd2,iterations_broyd2 = broyden(x_guess_2)

print("The solution from Broyden's method with the second set of initial values
```

```
Success
The solution from Broyden's method with the second set of initial values is [1.0
00082   0.99991826] and it converged in 25 iterations.
```

In [ ]: