Problem 4

Goals:

- To explore the built-in functions in Matlab used to solve systems of linear equations Ax = b.
- To compare different options in terms of accuracy and efficiency.

First define two matrices: A and b. Make sure the consistency in dimension is maintained.

Method 1: using np.linalg.solve

```
# Solve the system Ax = b using numpy's solve function
x = np.linalg.solve(A, b)

# Print the solution vector x
print("\nSolution vector x:")
print(x)
```

```
# Calculate the residual r = A*x - b
r = np.dot(A, x) - b
# Print the residual vector r
print("\nResidual vector r:")
print(r)
```

Method 2: using np.linalg.lu for A=PLU decomposition method

```python
import scipy.linalg
# Perform LU decomposition of A
P, L, U = scipy.linalg.lu(A)  # Note: Use scipy.linalg.lu for LU decomposition
# Solve the system Ax = b using LU decomposition
x1 = np.linalg.solve(A, b)
# Calculate the error between solutions
err1 = np.dot(A, x1) - b
# Print matrices P, L, U and vectors x1, err1
print("Matrix P (Permutation matrix):")
print(P)
print("\nMatrix L (Lower triangular matrix):")
print(L)
print("\nMatrix U (Upper triangular matrix):")
print(U)
print("\nSolution vector x1:")
print(x1)
print("\nError vector err1:")
print(err1)
```

Now, we can write:
LUx=B
Let, Ux=Y
So, LY=B,
Now solve for Y
Then solve for x using Ux=Y

Method 3: using np.linalg.lstsq (AT.A.x=AT.B)

```
# Solve the system Ax = b using numpy's least squares function
y = np.linalg.lstsq(A, b, rcond=None)[0]
# Print the solution vector y
print("\nSolution vector y:")
print(y)
```

Method 4: Using matrix inverse(only works when A is square. or else use pseudoinverse)

```
# Solve the system Ax = b using matrix inverse
A_inv = np.linalg.inv(A)
x2 = np.dot(A_inv, b)

# Calculate the residual r2 and error err2
r2 = np.dot(A, x2) - b
err2 = x - x2

# Print solution vector x2, residual r2 and error err2
print("\nSolution vector x2 using inverse:")
print(x2)
print("\nResidual vector r2:")
print(r2)
print("\nError vector err2:")
print(err2)
```

Method 5: using Row reduced echelon form

```
def rref(A):
    """
    Computes the reduced row echelon form (RREF) of a matrix A.

    Parameters:
    A : numpy.ndarray
        Input matrix of shape (m, n)

    Returns:
    R : numpy.ndarray
        Reduced row echelon form of matrix A
    """
    A = A.astype(float)
    m, n = A.shape
    lead = 0
    for r in range(m):
        if lead >= n:
            break
        if A[r, lead] == 0:
            for i in range(r + 1, m):
                if A[i, lead] != 0:
                    A[[r, i]] = A[[i, r]]
                    break
        if A[r, lead] != 0:
            A[r] = A[r] / A[r, lead]
        for i in range(m):
            if i != r:
                A[i] = A[i] - A[i, lead] * A[r]
        lead += 1
    return A
```

```
# Compute reduced row echelon form of [A | b]
print(A)
print(b)
C = np.hstack((A, b.reshape(-1, 1)))
print("[A:B] vector", C)
R = rref(C)

# Extract solution vector x3 from the RREF matrix
x3 = R[:, -1]

# Calculate residuals
r3 = np.dot(A, x3) - b
err3 = x - x3

print("\nSolution vector x3 (from RREF):")
print(x3)
print("\nResidual vector r3 (from RREF):")
print(r3)
print("\nError vector err3 (difference between x and x3):")
print(err3)
```

Comparing the computational time taken by each method:

```
# Method 1: Solve using solve operator
start_time = time.time()
x1 = np.linalg.solve(A, b)
end_time = time.time()
time_backslash = end_time - start_time

# Method 2: Solve using matrix inverse
start_time = time.time()
x2 = np.linalg.inv(A) @ b
end_time = time.time()
time_inv = end_time - start_time

# Method 3: Solve using reduced row echelon form (rref)
start_time = time.time()
C = np.hstack((A, b))
R = rref(C)
x3 = R[:, -1]
end_time = time.time()
time_rref = end_time - start_time

# Print the solution vectors and computational times
print("\nSolution vector x1 (using backslash operator):")
print(x1)

print("\nSolution vector x2 (using matrix inverse):")
print(x2)

print("\nSolution vector x3 (using reduced row echelon form):")
print(x3)

print("\nComputational times:")
print(f"Backslash operator: {time_backslash} seconds")
print(f"Matrix inverse: {time_inv} seconds")
print(f"Reduced row echelon form (rref): {time_rref} seconds")
```

For overdetermined system (more equations than number of variables)

```python
import numpy as np

# Define matrix A and vector b for the underdetermined system
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9],
              [10, 11, 12]])

b = np.array([1, 3, 5, 7]).reshape(-1, 1)  # Remove the reshape, as it's unnecessary

# Solve the system Ax = b using the backslash operator
x = np.linalg.lstsq(A, b, rcond=None)[0] #rcond: This parameter is used to determine the cutoff
#for small singular values in the computation of the least squares solution.

#[0] takes the first value of answer amongs all possible answers. Result is a tuple.

# Calculate the residual r1 = Ax - b
r1 = np.dot(A, x) - b

# Print the solution vector x
print("\nSolution vector x:")
print(x)

# Print the residual vector r1
print("\nResidual vector r1:")
print(r1)

# Obtain another particular solution using the pseudoinverse pinv(A)
A_pinv = np.linalg.pinv(A)
y = np.dot(A_pinv, b)

# Calculate the residual r2 = Ax - b for solution y
r2 = np.dot(A, y) - b

# Print the solution vector y from pseudoinverse
print("\nSolution vector y (from pseudoinverse):")
print(y)

# Print the residual vector r2
print("\nResidual vector r2:")
print(r2)
```