Iterative Methods for Solving Linear Systems

MATH 310 Project 2

Anny

April 7, 2024

1 Methods

1.1 Method 1

Method 1: $A = L + U$ Decomposition

To solve the system $Ax = b$ using this method, follow these steps:

1. Decompose matrix A into L and U:
   - L is the lower triangular matrix with diagonal entries.
   - U is the upper triangular matrix with 0s on the diagonal.
2. Initialize an initial guess x_0.
3. Iterate to find x:
   - Compute $x_1 = L^{-1}(b - Ux_0)$.
   - Repeat the process with $x_{k+1} = L^{-1}(b - Ux_k)$ until the error given by $||Ax_{k+1} - b||$ is less than the tolerance level.

1.2 Method 2

Method 2: $A = L + D + U$ Decomposition

To solve the system $Ax = b$ using this method, follow these steps:

1. Decompose matrix A into L, D, and U:
   - L is the strictly lower triangular part of A.
   - D is the diagonal part of A.
   - U is the strictly upper triangular part of A.
2. Initialize an initial guess x_0.
3. Iterate to find x:
   - Compute $x_1 = D^{-1}(b - Lx_0 - Ux_0)$.
   - Repeat the process with $x_{k+1} = D^{-1}(b - Lx_k - Ux_k)$ until $||Ax_{k+1} - b < tol||$ is met.

2 Task 2

2.1 A = L + U

Construct a Python function factor_lu() that accepts a square matrix A and then factorize it into $A = L + U$. Given a selected 4×4 matrix $A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 7 & 8 \\ 3 & 3 & 4 & 6 \end{bmatrix}$

```python
import numpy as np
def factor_lu(A):
    A = A.astype(float)
    n = A.shape[0]
    L = np.diag(np.diag(A)) # set diagonal of L is that of A
    U = A # Initialize U
    for i in range(n):
        for j in range(i, n):
            L[j, i] = A[j, i] # lower triangular part
            U[j, i] = 0 # set lower triangular of U is 0
            U[i, i] = 0 # diagonal of U is 0
    return L, U

A = np.array([[1,3,5,7],
              [2,4,6,8],
              [1,3,7,8],
              [3,3,4,6]])
print("LU:")
L, U = factor_lu(A)
print("L:\n", L)
print("U:\n", U)
print("L+U:\n",L+U)
```

```
------------------------------------------------------------
Output:
LU:
L:
 [[1. 0. 0. 0.]
 [2. 4. 0. 0.]
 [1. 3. 7. 0.]
 [3. 3. 4. 6.]]
U:
 [[0. 3. 5. 7.]
 [0. 0. 6. 8.]
 [0. 0. 0. 8.]
 [0. 0. 0. 0.]]
L+U:
 [[1. 3. 5. 7.]
 [2. 4. 6. 8.]
 [1. 3. 7. 8.]
 [3. 3. 4. 6.]]
```

2.2 $\mathbf{A = L + D + U}$

Then, for method 2, construct a Python function factor_ldu() which accepts a square matrix A but factorize it into $A = L + D + U$. Apply the same 4×4 matrix $A = \begin{bmatrix} 1 & 3 & 5 & 7 \\ 2 & 4 & 6 & 8 \\ 1 & 3 & 7 & 8 \\ 3 & 3 & 4 & 6 \end{bmatrix}$ for testing.

```python
def factor_ldu(A):
    A = A.astype(float)
```

```
    n = A.shape[0]
    L = np.zeros_like(A) # Initialize L
    U = A # Initialize U
    D = np.diag(np.diag(A)) # Extract the diagonal of A for D

    for i in range(n):
        for j in range(i, n):
            L[j, i] = A[j, i] # lower triangular part
            L[i, j] = 0 # diagonal of L is 0
            U[j, i] = 0 # set lower triangular of U is 0
            U[i, i] = 0 # diagonal of U is 0
    return L, D, U


A = np.array([[1,3,5,7],
              [2,4,6,8],
              [1,3,7,8],
              [3,3,4,6]])

L, D, U = factor_ldu(A)
print("L:\n", L)
print("D:\n", D)
print("U:\n", U)
print("L+D+U:\n",L+D+U)
-----------------------------------------------------------------------
Output:
L:
 [[0. 0. 0. 0.]
 [2. 0. 0. 0.]
 [1. 3. 0. 0.]
 [3. 3. 4. 0.]]
D:
 [[1. 0. 0. 0.]
 [0. 4. 0. 0.]
 [0. 0. 7. 0.]
 [0. 0. 0. 6.]]
U:
 [[0. 3. 5. 7.]
 [0. 0. 6. 8.]
 [0. 0. 0. 8.]
 [0. 0. 0. 0.]]
L+D+U:
 [[1. 3. 5. 7.]
 [2. 4. 6. 8.]
 [1. 3. 7. 8.]
 [3. 3. 4. 6.]]
```

3 Task 3

Selecting the square matrix $A = \begin{bmatrix} 10 & -1 & 2 & 0 \\ 1 & 11 & -1 & 3 \\ 2 & 1 & 10 & -1 \\ 0 & 3 & 1 & 8 \end{bmatrix}$ and the vector $b = \begin{bmatrix} 6 \\ 25 \\ 11 \\ 15 \end{bmatrix}$

3.1 Method 1

```
import numpy as np
def method1(A, b, tol=1e-5):
    A = A.astype(float)
    x = np.ones_like(b).astype(float) # initialize x

    # Decompose A into L and U, call factor_lu() function
    L, U = factor_lu(A)

    # error is determined by ||Ax - b||
    error = np.linalg.norm(np.dot(A, x) - b)
    pre_error = error # set previous error for checking divergence

    while np.abs(error) > tol:
        x_new = np.linalg.solve(L, b - np.dot(U, x)) # calculate the current approximate solution
        error = np.linalg.norm(np.dot(A, x_new) - b) # update error for stopping criterion

        # check for divergence
        if error > pre_error:
            raise ValueError(f"We detect that the approximation is diverging.")

        x = x_new # update x
    return x

A = np.array([[10, -1, 2, 0],
              [1, 11, -1, 3],
              [2, 1, 10, -1],
              [0, 3, 1, 8]])
b = np.array([[6],
              [25],
              [11],
              [15]])
method1_output = method1(A,b)
print(method1_output)
-----------------------------------------------
Output:
[[0.62733922]
 [2.02010265]
 [0.87335136]
 [1.00829259]]
```

3.2 Method 2

```
def method2(A, b, tol=1e-5):
    A = A.astype(float)
    x = np.zeros_like(b)

    # Decompose A into L, D, and U, call factor_ldu() function
    L, D, U = factor_ldu(A)

    error = np.linalg.norm(np.dot(A, x) - b)
    pre_error = error # for checking divergence

    while np.abs(error) > tol:
        x_new = np.linalg.solve(D, b - np.dot(L, x) - np.dot(U, x)) # calculate the current approximate solution
        error = np.linalg.norm(np.dot(A, x_new) - b) # update error for stopping criterion

        # check for divergence
        if error > pre_error:
            raise ValueError(f"We detect that the approximation is diverging.")

        x = x_new # update x

    return x

A = np.array([[10, -1, 2, 0],
              [1, 11, -1, 3],
              [2, 1, 10, -1],
              [0, 3, 1, 8]])
b = np.array([[6],
              [25],
              [11],
              [15]])
method2_output = method2(A,b)
print(method2_output)
-----------------------------------------------
Output:
[[0.62733995]
 [2.02010315]
 [0.87335084]
 [1.00829283]]
```

Then check for the solution x found by $np.linalg.solve(A, b)$, comparing with the solutions from $method1()$ and $method2()$.

```
Output:
solution by np.linalg.solve(A, b):
 [[0.62734012]
 [2.02010303]
 [0.87335092]
 [1.0082925 ]]
method1 solution absolute difference:
 [[9.00323160e-07]
```

```
 [3.74374609e-07]
 [4.31773823e-07]
 [8.64187506e-08]]
method2 solution absolute  difference:
 [[1.63222890e-07]
 [1.22032436e-07]
 [8.84527613e-08]
 [3.28817580e-07]]
```

4 Task 4

Those iterative methods for solving $Ax = b$ can fail or get inaccurate solutions due to the following scenarios:

- **Singularity Matrix**: If the chosen square matrix A is singular or close to singular, where its determinant is 0, the linear system can bot have a unique solution. This will lead the methods not to converge.
- **Mechanical Precision**: Since we apply Python functions to solve the problem, our computations will fail when round-off errors happen because only a finite number of digits are stored in the computer's floating-point arithmetic system.
- **Divergence**: Divergence will be caused by a weak diagonal dominance of matrix. If the errors given by $||Ax - b||$ are increasing with each iteration, this indicates that the method is moving away from the solution.
- L **has zeros on its diagonal for method1**: Method 1 will fail when L has zero on its diagonal which means it cannot be
- D **has zeros for method2**: Method 2 will fail when D contains zero, which means D^{-1} does not exist.

4.1 Example 1: Singular Matrix

Select a singular matrix $A = \begin{bmatrix} 0 & 2 & 3 & 4 \\ 1 & 5 & 6 & 7 \\ 1 & 8 & 9 & 10 \\ 1 & 11 & 12 & 13 \end{bmatrix}$ and the corresponding vector $b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$.

Then, apply method 1 to solve the system: $L = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 5 & 0 & 0 \\ 1 & 8 & 9 & 0 \\ 1 & 11 & 12 & 13 \end{bmatrix}$ and $U = \begin{bmatrix} 0 & 2 & 3 & 4 \\ 0 & 0 & 6 & 7 \\ 0 & 0 & 0 & 10 \\ 0 & 0 & 0 & 0 \end{bmatrix}$.

We cannot find L^{-1} for solving $x_{k+1} = L^{-1}(b - Ux_k)$ to meet method 1's convergence criterion.

4.2 Example 2: Divergence Detection

Select a square matrix $A = \begin{bmatrix} 1 & 2 & 0 & 1 \\ 2 & 1 & 2 & 0 \\ 0 & 2 & 1 & 2 \\ 1 & 0 & 2 & 1 \end{bmatrix}$ and the corresponding vector $b = \begin{bmatrix} 4 \\ 5 \\ 5 \\ 4 \end{bmatrix}$.

Then, $L = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 1 & 0 & 2 & 0 \end{bmatrix}$, $D = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$, and $U = \begin{bmatrix} 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$.

With these matrices, we will proceed $x_{k+1} = D^{-1}(b - Lx_k - Ux_k)$ iteratively. However, the first 10 iterations give us a diverging result:

```
Iteration 1: [4. 5. 5. 4.]
Iteration 2: [-10. -13. -13. -10.]
Iteration 3: [40. 51. 51. 40.]
Iteration 4: [-138. -177. -177. -138.]
Iteration 5: [496. 635. 635. 496.]
Iteration 6: [-1762. -2257. -2257. -1762.]
Iteration 7: [6280. 8043. 8043. 6280.]
Iteration 8: [-22362. -28641. -28641. -22362.]
Iteration 9: [ 79648. 102011. 102011.  79648.]
Iteration 10: [-283666. -363313. -363313. -283666.]
```

The matrix A has a symmetric structure and non-zero off-diagonal entries that are relatively large compared to the diagonal entries. This structure does not guarantee convergence for Method 2, because its update rule does not favor convergence for such matrices.

5 Task 5

5.1 Dominant Diagonal Entries

From the Task 4, we can assume that the diagonal dominance will be our convergence criterion, this means that the diagonal entries should be relatively larger than the absolute sum of the other entries in the row:

For instance, $A = \begin{bmatrix} -10 & 1 & 2 & 3 \\ 1 & 11 & 2 & 3 \\ 2 & 2 & 10 & 4 \\ 3 & 3 & 6 & 15 \end{bmatrix}$, where $|10| > |1| + |2| + |3|$, $|11| > |1| + |2| + |3|$, $|10| > |2| + |2| + |4|$, and $|15| > |3| + |3| + |6|$. Its corresponding vector $b = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$.

```
Output:
solution by np.linalg.solve(A, b):
 [[0.00115141]
 [0.09873345]
 [0.21588946]
```

```
 [0.16033391]]
method 1 solution:
 [[0.00115119]
 [0.09873394]
 [0.2158888 ]
 [0.16033412]]
method 2 solution:
 [[0.00115144]
 [0.09873324]
 [0.21588887]
 [0.16033425]]
```

Methods fail if change the matrix to be $A = \begin{bmatrix} -1 & 1 & 2 & 3 \\ 1 & 11 & 2 & 3 \\ 2 & 2 & 10 & 4 \\ 3 & 3 & 6 & 15 \end{bmatrix}$, where $|-1| < |1|+|2|+|3|$. Method 1 can work but method 2 raises value error and detects divergence.

5.2 Symmetric Positive Matrix

Assume that we have a symmetric positive matrix $A = \begin{bmatrix} 4 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 0 & 0 & 1 & 10 \end{bmatrix}$ and its corresponding vector $b = \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix}$.

```
Output:
solution by np.linalg.solve(A, b):
 [[0.16568047]
 [0.33727811]
 [0.82248521]
 [0.01775148]]
method 1 solution:
 [[0.16567859]
 [0.33728026]
 [0.8224839 ]
 [0.01775161]]
method 2 solution:
 [[0.16568075]
 [0.33727931]
 [0.82248587]
 [0.01775174]]
```

If we change the matrix to be $A = \begin{bmatrix} 4 & 1 & 0 & 0 \\ 1 & 3 & 1 & 0 \\ 0 & 1 & 2 & 1 \\ 8 & 0 & 1 & 10 \end{bmatrix}$, where now $A^T \neq A$. Method 2 detects divergence and raise its warning.

5.3 Lower Condition Number

A low condition number results in convergence towards the solution since the errors in each iterative step are less amplified.

```
b = np.array([[1], [2], [2], [1]])

# fail
A0 = np.array([[1,3,5,7],
               [2,4,6,8],
               [1,3,7,8],
               [3,3,4,6]])

# converge
A1 = np.array([[10, -1, 2, 0],
               [1, 11, -1, 3],
               [2, 1, 10, -1],
               [0, 3, 1, 8]])
# fail
A2 = np.array([[0, 2, 3, 4],
               [1, 5, 6, 7],
               [1, 8, 9, 10],
               [1, 11, 12, 13]])
# fail
A3 = np.array([[1, 2, 0, 1],
                [2, 1, 2, 0],
                [0, 2, 1, 2],
                [1, 0, 2, 1]])

# converge
A4 = np.array([[-10, 1, 2, 3],
               [1, 11, 2, 3],
               [2, 2, 10, 4],
               [3, 3, 6, 15]])

# method 2 fails
A5 = np.array([[-1, 1, 2, 3],
               [1, 11, 2, 3],
               [2, 2, 10, 4],
               [3, 3, 6, 15]])

# converge
A6 = np.array([[4, 1, 0, 0],
               [1, 3, 1, 0],
               [0, 1, 2, 1],
               [0, 0, 1, 10]])

# method 2 fails
A7 = np.array([[4, 1, 0, 0],
               [1, 3, 1, 0],
               [0, 1, 2, 1],
               [8, 0, 1, 10]])

print("A0:",np.linalg.cond(A0, p=2))
```

```
print("A1:", np.linalg.cond(A1, p=2))
print("A2:", np.linalg.cond(A2, p=2))
print("A3:", np.linalg.cond(A3, p=2))
print("A4:", np.linalg.cond(A4, p=2))
print("A5:", np.linalg.cond(A5, p=2))
print("A6:", np.linalg.cond(A6, p=2))
print("A7:", np.linalg.cond(A7, p=2))
----------------------------------------
Output:
A0: 60.948339884293105
A1: 2.0492445966291517
A2: 1.3270656550699805e+19
A3: 10.403882032022084
A4: 2.89428769673837
A5: 12.195256473167674
A6: 8.46821785953181
A7: 12.809684927643232
```

Both methods will be proceeded when the matrix's condition number is less than 10 from the previous given examples, which indicates that a lower condition number will help to ensure the convergence of iterative methods.

6 Task 6

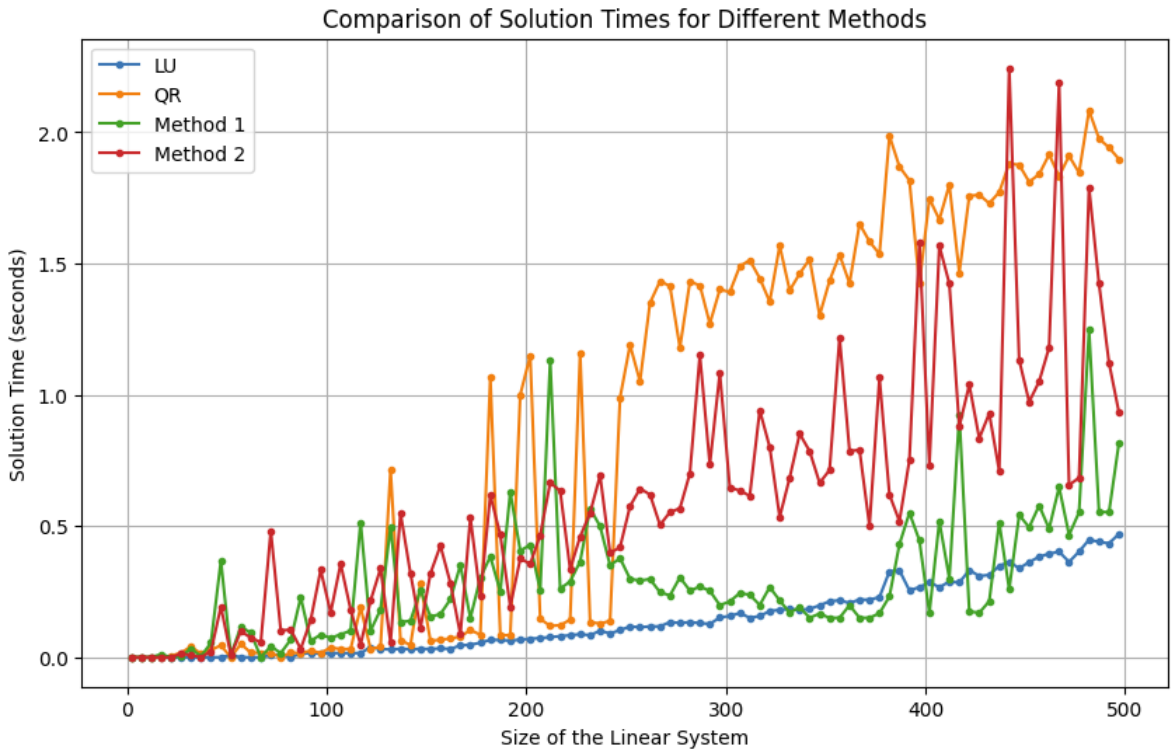

Figure 1: Time taken to solve a random linear system of sizes (2 to 500) using different methods: LU algorithm, QR algorithm, and two iterative methods, method1() and method2().

There is significant variability in the solution times for Method 1 and Method 2, as evidenced by the large spikes. This might suggest that these methods are sensitive to certain matrix characteristics, as we discussed in Task 4 and 5. All methods show a general increase in solution time as the size of the linear system increases. This is expected because larger systems require more computational work. However, the rate at which time increases with system size differs among the methods, with QR after size of 240 and LU appearing more stable and scaling better.

Despite the spikes for Methods 1 and 2, there are system sizes where these methods seem to outperform LU and QR, especially for larger sizes. Method 1 sometimes solve the systems (size of more than 300) even faster than LU. This suggests that under certain conditions or for specific types of matrices, these methods could be more efficient and can be implemented faster.