Chapter-09 Numerical Linear Algebra

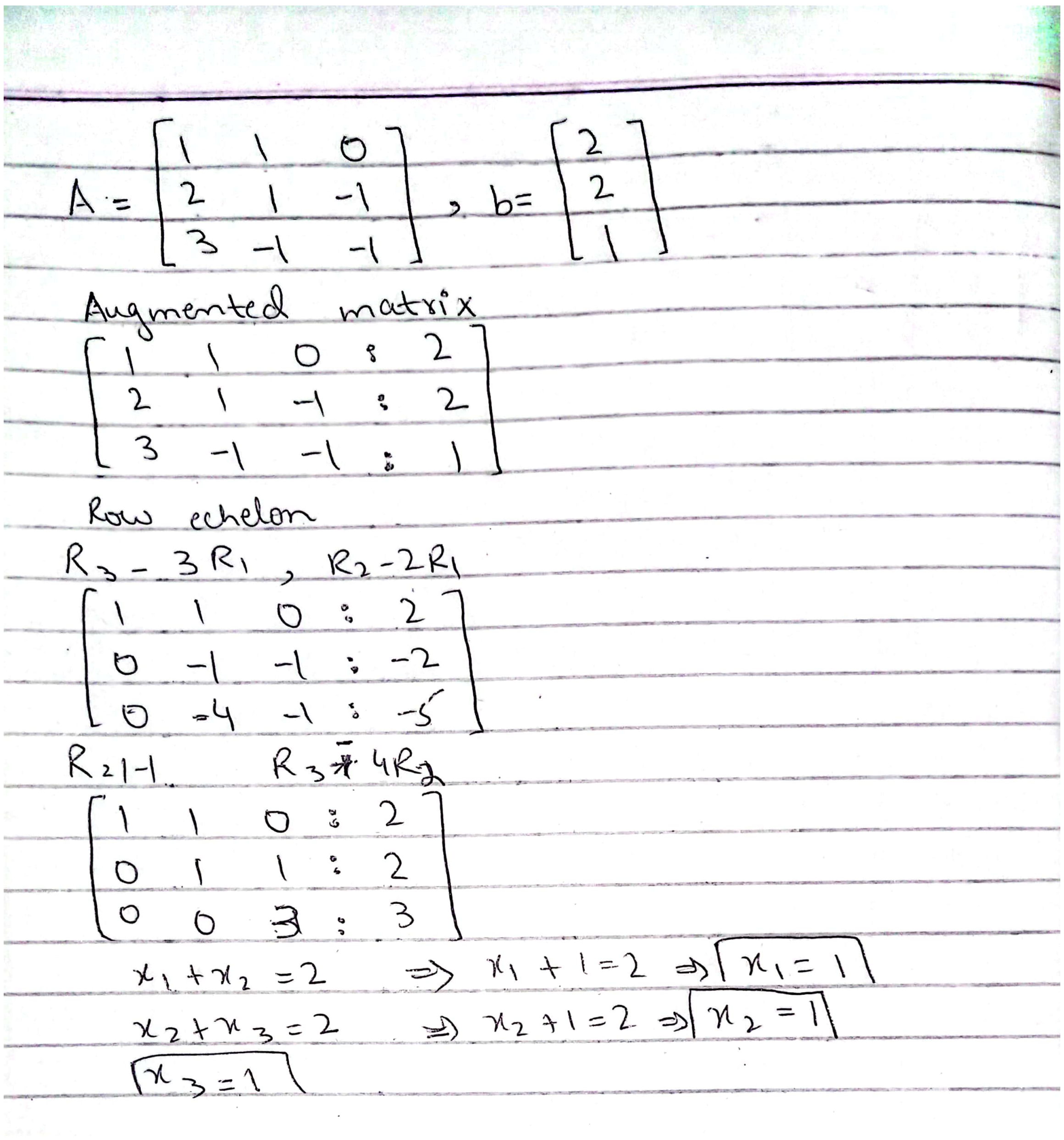
Date	@December 11, 2024
tag	

Numerical Solution of Linear Systems

Iterative Methods

▼ Numerical Solution of Linear Systems

▼ Gaussian Elimination



$$A = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix}, \quad b = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$$

Augmented matrix

$$\left[\begin{array}{ccc:c} 1 & 1 & 0 & 2 \\ 2 & 1 & -1 & 2 \\ 3 & -1 & -1 & 1 \end{array}\right]$$

Row echelon

$R_3 - 3R_1$, $R_2 - 2R_1$

$$\left[\begin{array}{ccc:c} 1 & 1 & 0 & 2 \\ 0 & -1 & -1 & -2 \\ 0 & -4 & -1 & -5 \end{array}\right]$$

$R_2 | -1$ $\qquad$ $R_3 \overset{-}{\cancel{+}} 4R_2$

$$\left[\begin{array}{ccc:c} 1 & 1 & 0 & 2 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 3 & 3 \end{array}\right]$$

$x_1 + x_2 = 2 \Rightarrow x_1 + 1 = 2 \Rightarrow \boxed{x_1 = 1}$

$x_2 + x_3 = 2 \Rightarrow x_2 + 1 = 2 \Rightarrow \boxed{x_2 = 1}$

$\boxed{x_3 = 1}$

```
def Gauss_elimination(A,b):
    n = len(b)
    # Gaussian elimination
    for k in range(n-1):
        for i in range(k+1,n):
            if A[k,k]!=0.0:
                mik=A[i,k]/A[k,k]
                A[i,k+1:n] = A[i,k+1:n] - mik * A[k,k+1:
n]
                b[i] = b[i] - mik * b[k]
    # backward substitution
    for k in range(n-1,-1,-1):
        b[k] = (b[k]-np.dot(A[k,k+1:n],b[k+1:n]))/A[k,k]
    return b
```

```
A = np.array([[ 1., 1., 0.],
              [ 2., 1.,-1.],
              [ 3.,-1.,-1.]])
b = np.array([ 2., 2., 1.])
x = Gauss_elimination( A.copy(), b.copy() )
print(x)
```

▼ Code explanation

Gaussian Elimination

Gaussian elimination is a two-step process:

1. **Forward Elimination**: Transform the system Ax=b into an **upper triangular system** Ux=b′.
2. **Backward Substitution**: Solve the upper triangular system for x.

Code Walk-through

Forward Elimination

```
for k in range(n-1):
    for i in range(k+1,n):
        if A[k,k]!=0.0:
            mik=A[i,k]/A[k,k]
            A[i,k+1:n] = A[i,k+1:n] - mik * A[k,k+1:n]
            b[i] = b[i] - mik * b[k]
```

1. **Outer Loop (`k`)**:
    - k is the current pivot row.
    - The algorithm eliminates elements below the diagonal in column k.
2. **Inner Loop (`i`)**:
    - Iterates over rows i below the pivot row k.
3. **Pivot Normalization (`mik`)**:

- Compute the **multiplier** $m_{ik} = \frac{A[i,k]}{A[k,k]}$
- This multiplier is used to eliminate A[i,k], making it 0.

4. **Row Updates**:
    - The entries in row i (columns k+1 to n−1) are updated using:

    $$A[i,j] \leftarrow A[i,j] - m_{ik} \cdot A[k,j]$$

    - The corresponding entry in the right-hand side vector b is updated:

$$b[i] \leftarrow b[i] - m_{ik} \cdot b[k]$$

Relation to Manual Method

In the manual Gaussian elimination process, this corresponds to the **row operations** used to form the **echelon form** of the matrix A. Each step eliminates elements below the pivot in the current column.

Backward Substitution

```
for k in range(n-1,-1,-1):
    b[k] = (b[k]-np.dot(A[k,k+1:n],b[k+1:n]))/A[k,k]
```

1. **Solve for x_k**:
    - Starting from the last row and moving upwards (reverse order).
    - Use the equation:

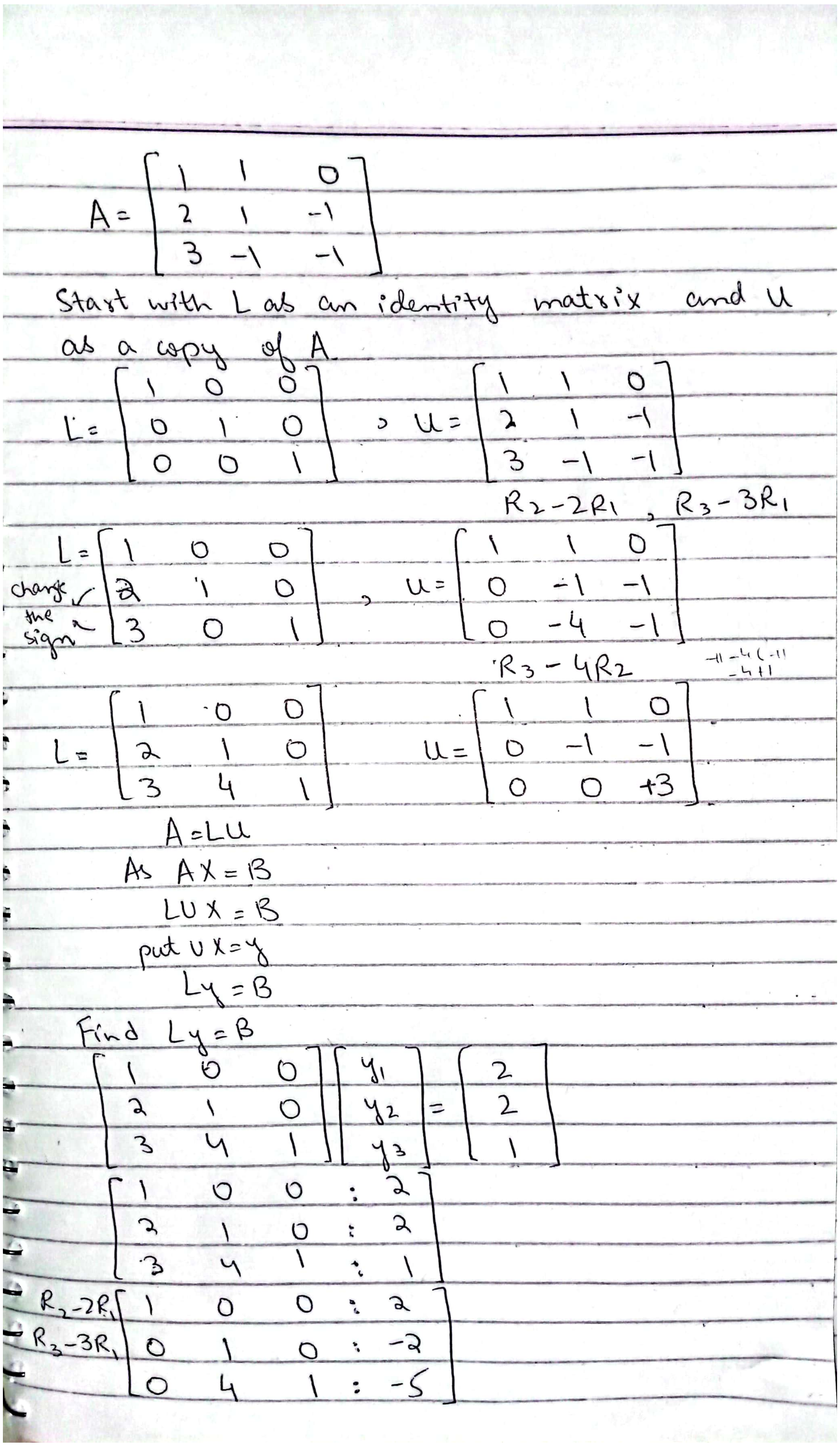
    $$x_k = \frac{b[k] - \sum_{j=k+1}^{n-1} A[k,j] \cdot x_j}{A[k,k]}$$

    - The term $\sum_{j=k+1}^{n-1} A[k,j] \cdot x_j$ is computed using `np.dot(A[k,k+1:n], b[k+1:n])`.
2. **Update b[k]**:
    - The solution vector x is stored in b.

Relation to Manual Method

Backward substitution is the manual process of solving the equations starting from the last row in the echelon form matrix (now upper triangular).

▼ LU Factorization



$$A = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix}$$

Start with L as an identity matrix and U as a copy of A.

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & -1 \\ 3 & -1 & -1 \end{bmatrix}$$

$R_2 - 2R_1$, $R_3 - 3R_1$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 0 & 1 \end{bmatrix}, \quad U = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & -4 & -1 \end{bmatrix}$$

change the sign

$R_3 - 4R_2$ $\quad -1-4(-1)$ $\quad -4+1$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & +3 \end{bmatrix}$$

$A = LU$

As $AX = B$

$LUX = B$

put $UX = y$

$Ly = B$

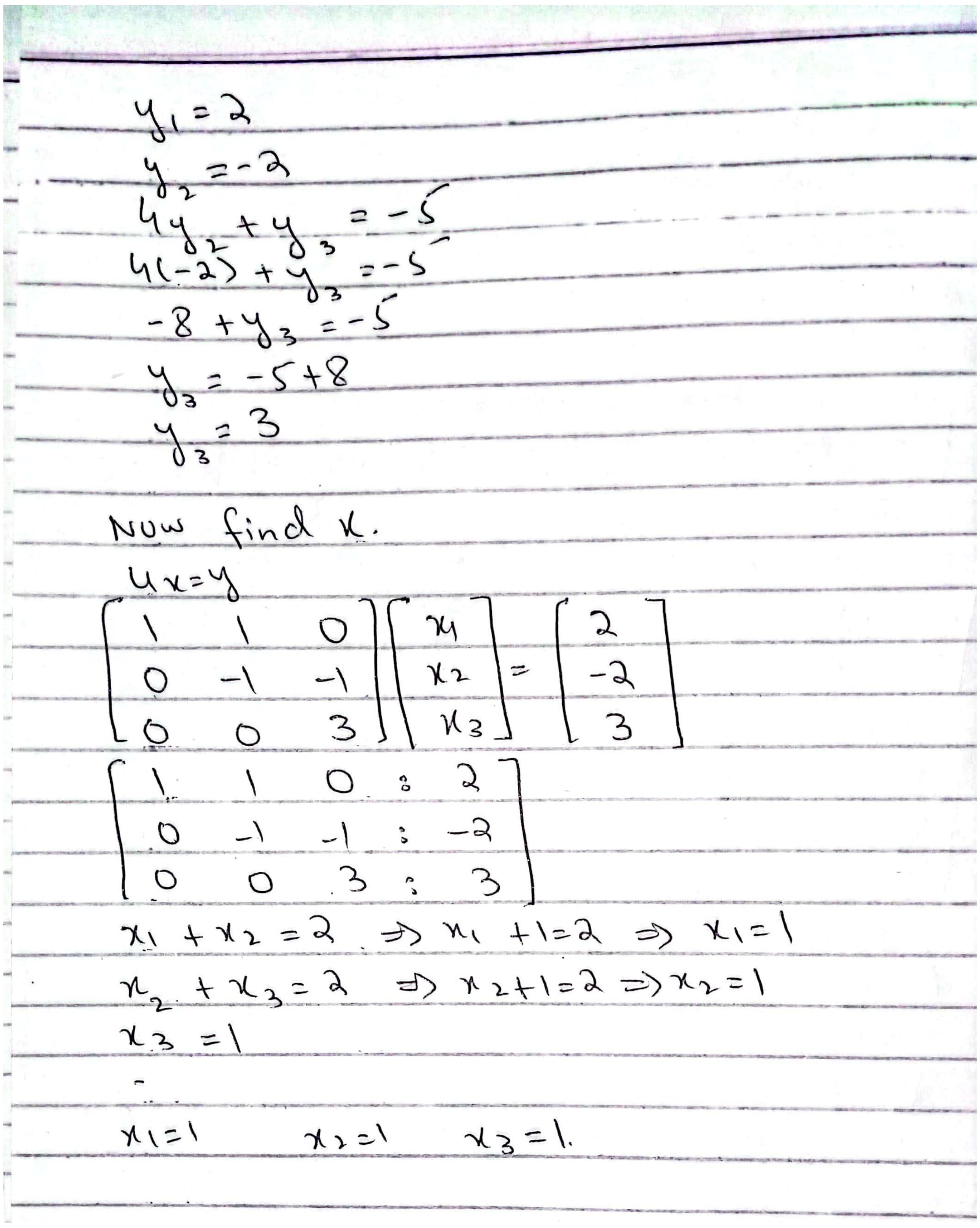
Find $Ly = B$

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 4 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \\ 1 \end{bmatrix}$$

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 2 & 1 & 0 & 2 \\ 3 & 4 & 1 & 1 \end{array}\right]$$

$R_2 - 2R_1$
$R_3 - 3R_1$

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & -2 \\ 0 & 4 & 1 & -5 \end{array}\right]$$



$y_1 = 2$

$y_2 = -2$

$4y_2 + y_3 = -5$

$4(-2) + y_3 = -5$

$-8 + y_3 = -5$

$y_3 = -5+8$

$y_3 = 3$

Now find x.

$Ux=y$

$$\begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & -1 \\ 0 & 0 & 3 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ -2 \\ 3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 0 & : & 2 \\ 0 & -1 & -1 & : & -2 \\ 0 & 0 & 3 & : & 3 \end{bmatrix}$$

$x_1 + x_2 = 2 \Rightarrow x_1 + 1 = 2 \Rightarrow x_1 = 1$

$x_2 + x_3 = 2 \Rightarrow x_2 + 1 = 2 \Rightarrow x_2 = 1$

$x_3 = 1$

$x_1 = 1 \qquad x_2 = 1 \qquad x_3 = 1.$

```
def LU(A):
    # LU factorization of matrix A
    n = len(A)
    for k in range(n-1):
        for i in range(k+1,n):
            if A[k,k] != 0.0:
                mik = A[i,k]/A[k,k]
                A[i,k+1:n] = A[i,k+1:n] - mik*A[k,k+1:n]
                A[i,k] = mik
    return A
```

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

```
B = LU(A)

L = np.tril(B)
U = np.triu(B)

print('L =', L)
print('U =', U)
print(L@U)
```

```
def solveLU(A,b):
    # Solution of the linear system LUx=b
    # matrix A contains the LU factorization of A
    n = len(A)
    # Solve the low triangular system Ly=b, note that L_ii
    for k in range(1,n):
        b[k] = b[k] - np.dot(A[k,0:k],b[0:k])
    # Solve the upper triangular system Ux=b
    b[n-1] = b[n-1]/A[n-1,n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - np.dot(A[k,k+1:n],b[k+1:n]))/A[k,k]
    return b
```

```
A = np.array([[ 1., 1., 0.],
              [ 2., 1.,-1.],
              [ 3.,-1.,-1.]])
b1 = np.array([ 2., 2., 1.])
b2 = np.array([ 4., 4., 2.])
# Perform LU factorization once
A = LU(A)
# Solve the first system using forward/backward substituti
x1 = solveLU(A,b1)
print(x1)
#Solve the second system using forward/backward substituti
x2 = solveLU(A,b2)
print(x2)
```

▼ Pivoting Strategies

Gaussian elimination with partial pivoting

swap the rows

There are some linear systems where Gaussian elimination can fail in practice due to the the finite precision arithmetic of the computers.

For example consider a linear system with one of the matrices

$$\mathbf{A}_1 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \quad \text{or} \quad \mathbf{A}_2 = \begin{pmatrix} 10^{-17} & 1 \\ 1 & 1 \end{pmatrix}$$

In the first case the first multiplier is 0 and thus we cannot perform Gaussian elimination. In the second case, with right-hand side (1,2), although the exact solution is close to (1,1) we compute (0,1) which is totally wrong:

```
def rowSwap(v,i,j):
    if len(v.shape) == 1:
        v[i],v[j] = v[j],v[i]
    else:
        v[[i,j],:] = v[[j,i],:]

def pGauss_elimination(A,b,tol=1.0e-15):
    n = len(b)
    for k in range(n-1):
        # swap rows if necessary
        p = np.argmax(np.abs(A[k:n,k])) + k
        if np.abs(A[p,k]) < tol:
            error.err('singular matrix has been detecte
d')
        if p != k:
            rowSwap(b,k,p)
            rowSwap(A,k,p)
        # perform Gauss elimination
        for i in range(k+1,n):
            if A[k,k] != 0.0:
                mik = A[i,k]/A[k,k]
                A[i,k+1:n] = A[i,k+1:n] - mik*A[k,k+1:n]
                b[i] = b[i] - mik*b[k]
    # check if the matrix is singular
    if np.abs(A[n-1,n-1]) < tol:
        error.err('singular matrix has been detected')
    # perform backward substitution
    b[n-1] = b[n-1]/A[n-1,n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] -np.dot(A[k,k+1:n],b[k+1:n]))/A[k,
k]
    return b
```

▼ Code Explanation

Explanation:

1. **Purpose of Row Swapping (Partial Pivoting)**:
   In Gauss elimination, at each step k, the algorithm tries to eliminate the entries below the pivot element (the diagonal element A[k,k]).

- If the pivot element is very small or zero, it can cause numerical instability or division by zero.
- To avoid this, we swap the current row k with a row p below it where A[p,k] (the entry in column k) has the largest absolute value. This process is called **partial pivoting**.

2. **Finding the Row with the Largest Pivot**:

   ```
   p = np.argmax(np.abs(A[k:n,k])) + k
   ```

   - `np.abs(A[k:n,k])`: Extracts the absolute values of the entries in column k from row k to n−1.
   - `np.argmax(...)`: Finds the index of the maximum value in this sub-array. Since indexing is relative to the sliced array starting at k, we add k to get the actual row index p.
   - p is now the index of the row with the largest absolute value in column k.

3. **Checking for a Singular Matrix**:

   ```
   if np.abs(A[p,k]) < tol:
       error.err('singular matrix has been detected')
   ```

   - If the largest pivot value is less than a very small tolerance (`tol`), the matrix is effectively singular (non-invertible). At this point, the function raises an error and stops execution.

4. **Swapping Rows if Needed**:

   ```
   if p != k:
       rowSwap(b,k,p)
       rowSwap(A,k,p)
   ```

   - If p˜I=k, it means the row with the largest pivot is not already the current row k. In that case:

     $p \neq k$

     - `rowSwap(b, k, p)`: Swaps the corresponding entries in the right-hand side vector b.

       bb

     - `rowSwap(A, k, p)`: Swaps rows k and p in the matrix A.
   - Swapping rows ensures that the largest possible pivot is used for the elimination step.

LU decomposition with pivoting

First, we define the permutation matrix. A permutation matrix is an identity matrix with its rows or columns interchanged.

For example, if we want to interchange rows 1 and 3, then the permutation matrix is:

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

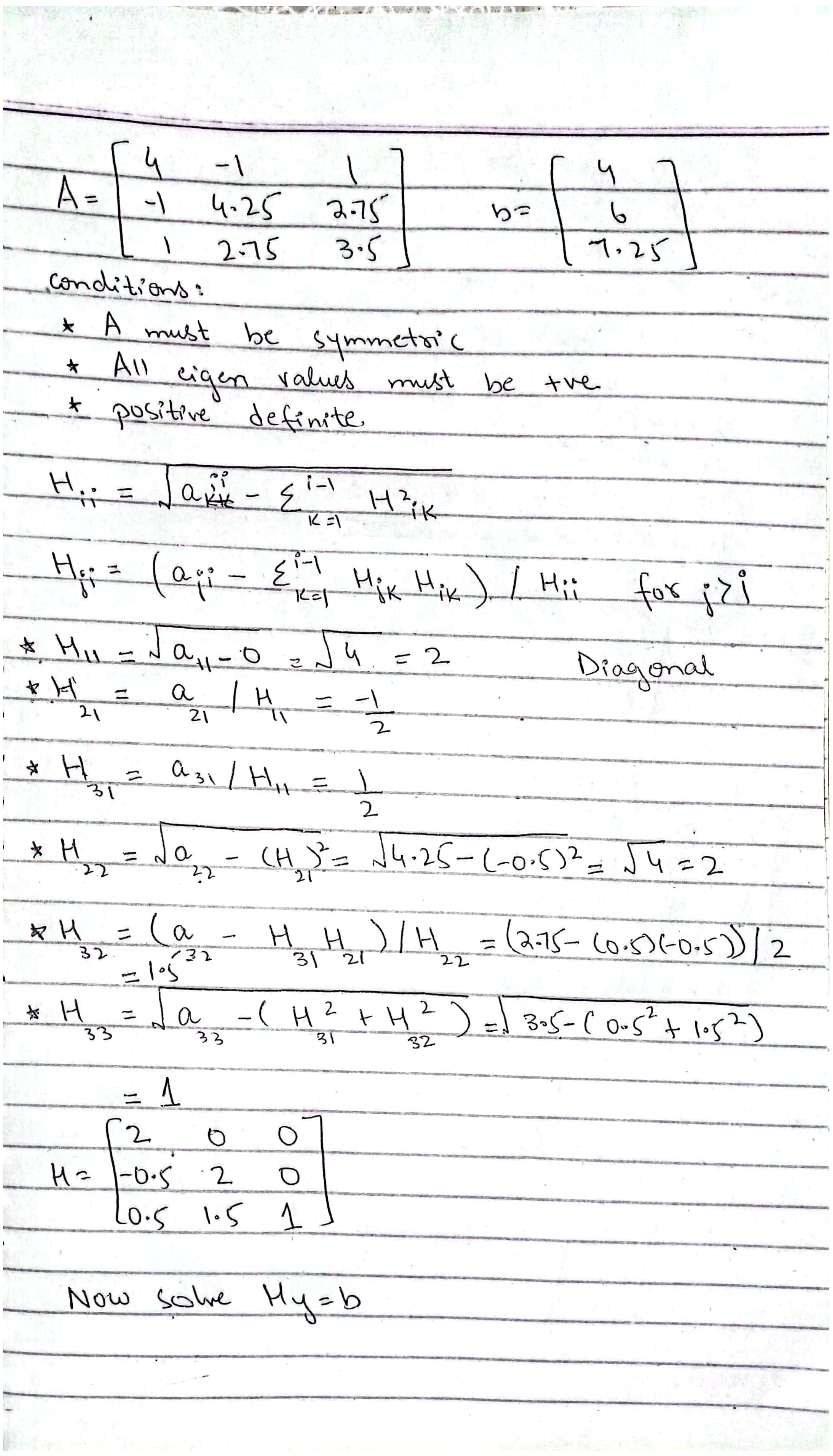
```
def LUpivot(A,tol=1.0e-15):
    n = len(A)
    # define permutation vector
    perm = np.array(range(n))
    for k in range(0,n-1):
        # perform row interchange if necessary
        p = np.argmax(np.abs(A[k:n,k])) + k
        if np.abs(A[p,k]) < tol:
            error.err('singular matrix has been detecte
d')
        if p != k:
            rowSwap(A,k,p)
            rowSwap(perm,k,p)
        # perform Gauss elimination
        for i in range(k+1,n):
            if A[i,k] != 0.0:
                mik = A[i,k]/A[k,k]
                A[i,k+1:n] = A[i,k+1:n] - mik*A[k,k+1:n]
                A[i,k] = mik
    return A, perm
```

```
def solveLUpivot(A,b,perm):
    n = len(A)
    # Store right-hand side in solution vector x
    x = b.copy()
    for i in range(n):
        x[i]=b[perm[i]]
    # Forward-backward substitution
    for k in range(1,n):
        x[k] = x[k] - np.dot(A[k,0:k],x[0:k])
    x[n-1] = x[n-1]/A[n-1,n-1]
    for k in range(n-2,-1,-1):
        x[k] = (x[k] - np.dot(A[k,k+1:n],x[k+1:n]))/A[k,
k]
    return x
```

```
A = np.array([[ 3., 17., 10.],
              [ 2., 4.,-2.],
              [ 6.,18.,-12.]])
b = np.array([ 30., 4., 12.])
[A,perm] = LUpivot(A)
x = solveLUpivot(A,b,perm)
print(x)

# construct lower triangular matrix L
L = np.tril(A,-1)+np.eye(3)
# construct upper triangular matrix U
U = np.triu(A)
# construct permutation matrix P
P = np.eye(3)
P = P[perm,:]
print('P='); print(P)
print('L='); print(L)
print('U='); print(U)
```

▼ Cholesky Factorization



$$A = \begin{bmatrix} 4 & -1 & 1 \\ -1 & 4.25 & 2.75 \\ 1 & 2.75 & 3.5 \end{bmatrix} \qquad b = \begin{bmatrix} 4 \\ 6 \\ 7.25 \end{bmatrix}$$

conditions:

* A must be symmetric
* All eigen values must be +ve
* positive definite

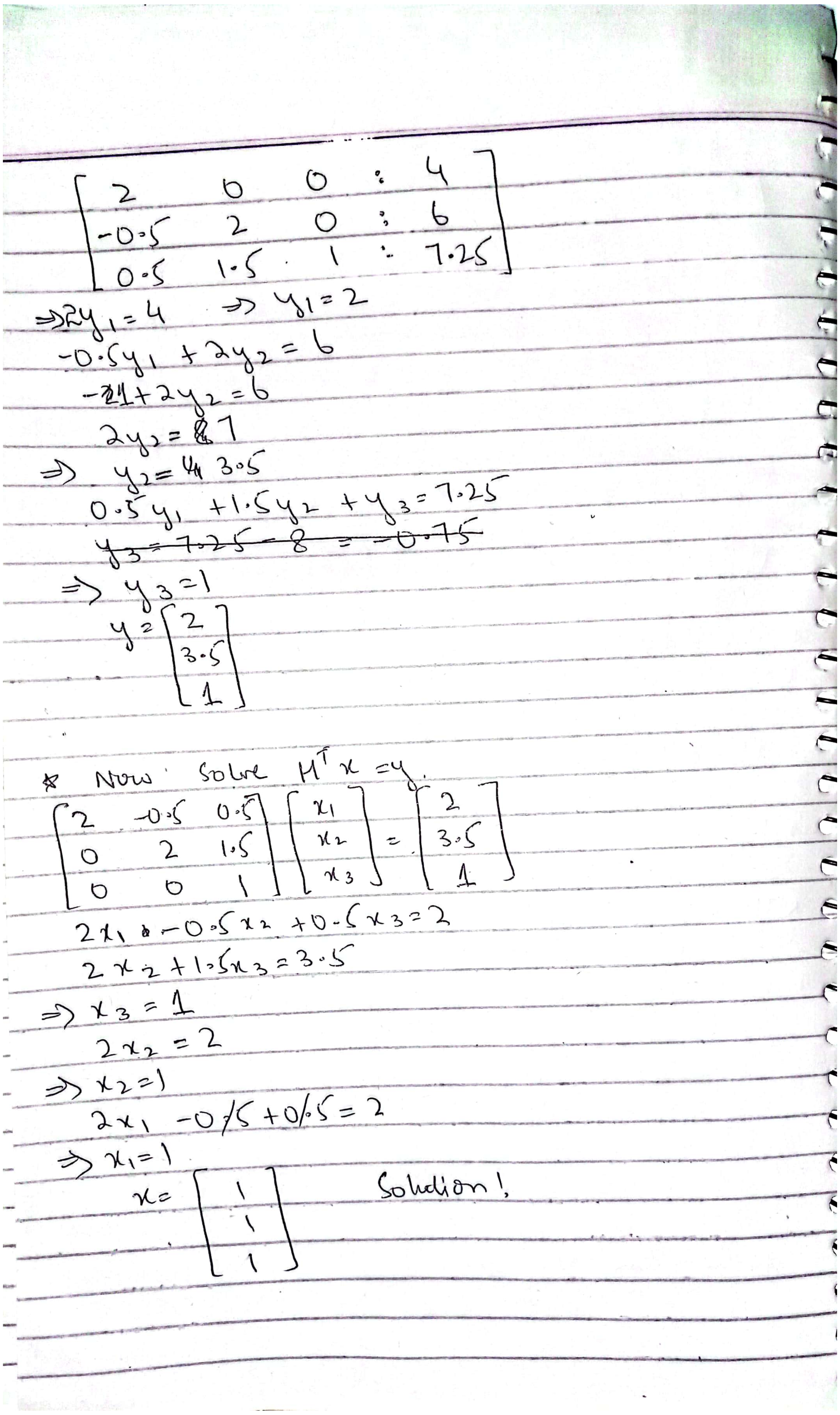
$$H_{ii} = \sqrt{a^{ii}_{\cancel{kk}} - \sum_{k=1}^{i-1} H^2_{ik}}$$

$$H_{ji} = \left(a_{ji} - \sum_{k=1}^{i-1} H_{jk} H_{ik}\right) / H_{ii} \quad \text{for } j > i$$

* $H_{11} = \sqrt{a_{11} - 0} = \sqrt{4} = 2$ Diagonal
* $H_{21} = a_{21} / H_{11} = \frac{-1}{2}$
* $H_{31} = a_{31} / H_{11} = \frac{1}{2}$
* $H_{22} = \sqrt{a_{22} - (H_{21})^2} = \sqrt{4.25 - (-0.5)^2} = \sqrt{4} = 2$
* $H_{32} = (a_{32} - H_{31} H_{21}) / H_{22} = (2.75 - (0.5)(-0.5)) / 2$
  $= 1.5$
* $H_{33} = \sqrt{a_{33} - (H_{31}^2 + H_{32}^2)} = \sqrt{3.5 - (0.5^2 + 1.5^2)}$
  $= 1$

$$H = \begin{bmatrix} 2 & 0 & 0 \\ -0.5 & 2 & 0 \\ 0.5 & 1.5 & 1 \end{bmatrix}$$

Now solve $Hy = b$



$$\left[\begin{array}{ccc:c} 2 & 0 & 0 & 4 \\ -0.5 & 2 & 0 & 6 \\ 0.5 & 1.5 & 1 & 7.25 \end{array}\right]$$

$\Rightarrow 2y_1 = 4 \quad \Rightarrow y_1 = 2$

$-0.5y_1 + 2y_2 = 6$

$-\cancel{2}1 + 2y_2 = 6$

$2y_2 = \cancel{8}7$

$\Rightarrow \quad y_2 = \cancel{4}\, 3.5$

$0.5y_1 + 1.5y_2 + y_3 = 7.25$

~~$y_3 = 7.25 - 8 = -0.75$~~

$\Rightarrow y_3 = 1$

$y = \begin{bmatrix} 2 \\ 3.5 \\ 1 \end{bmatrix}$

* Now solve $M^T x = y$.

$$\begin{bmatrix} 2 & -0.5 & 0.5 \\ 0 & 2 & 1.5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 3.5 \\ 1 \end{bmatrix}$$

$2x_1 - 0.5x_2 + 0.5x_3 = 2$

$2x_2 + 1.5x_3 = 3.5$

$\Rightarrow x_3 = 1$

$2x_2 = 2$

$\Rightarrow x_2 = 1$

$2x_1 - \cancel{0.5} + \cancel{0.5} = 2$

$\Rightarrow x_1 = 1$

$x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$ Solution!

```
def Cholesky(A):
    n = len(A)
    for i in range(n):
        try:
            A[i,i]=np.sqrt(A[i,i] - np.dot(A[i,0:i],A[i,
0:i]))
        except ValueError:
            error.err('Matrix is not positive definite')
        for j in range(i+1,n):
            A[j,i] = (A[j,i]-np.dot(A[j,0:i],A[i,0:i]))/
A[i,i]
    for k in range(1,n):
        A[0:k,k]=0.0
    return A
```

```
def solveCholesky(H,b):
    n = len(b)
    # Solve Hy=b
    for k in range(n):
        b[k] = (b[k] - np.dot(H[k,0:k],b[0:k]))/H[k,k]
    # Solve H^T x =y
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - np.dot(H[k+1:n,k],b[k+1:n]))/H[k,
k]
    return b
```

```
A = np.array([[4.0, -1.0, 1.0],
              [-1.0, 4.25, 2.75],
              [1.0, 2.75, 3.5]])
H = Cholesky(A)
print(H)
b = np.array([4.0, 6.0, 7.25])
x = solveCholesky(H,b)
print(x)
```

▼ Scipy

LU

```
import scipy.linalg as spl
A = np.array([[ 3., 17., 10.],
              [ 2., 4.,-2.],
              [ 6.,18.,-12.]])
(P,L,U) = spl.lu(A)
print(P)
print(L)
print(U)

A = np.array([[ 3., 17., 10.],
              [ 2., 4.,-2.],
              [ 6.,18.,-12.]])
```

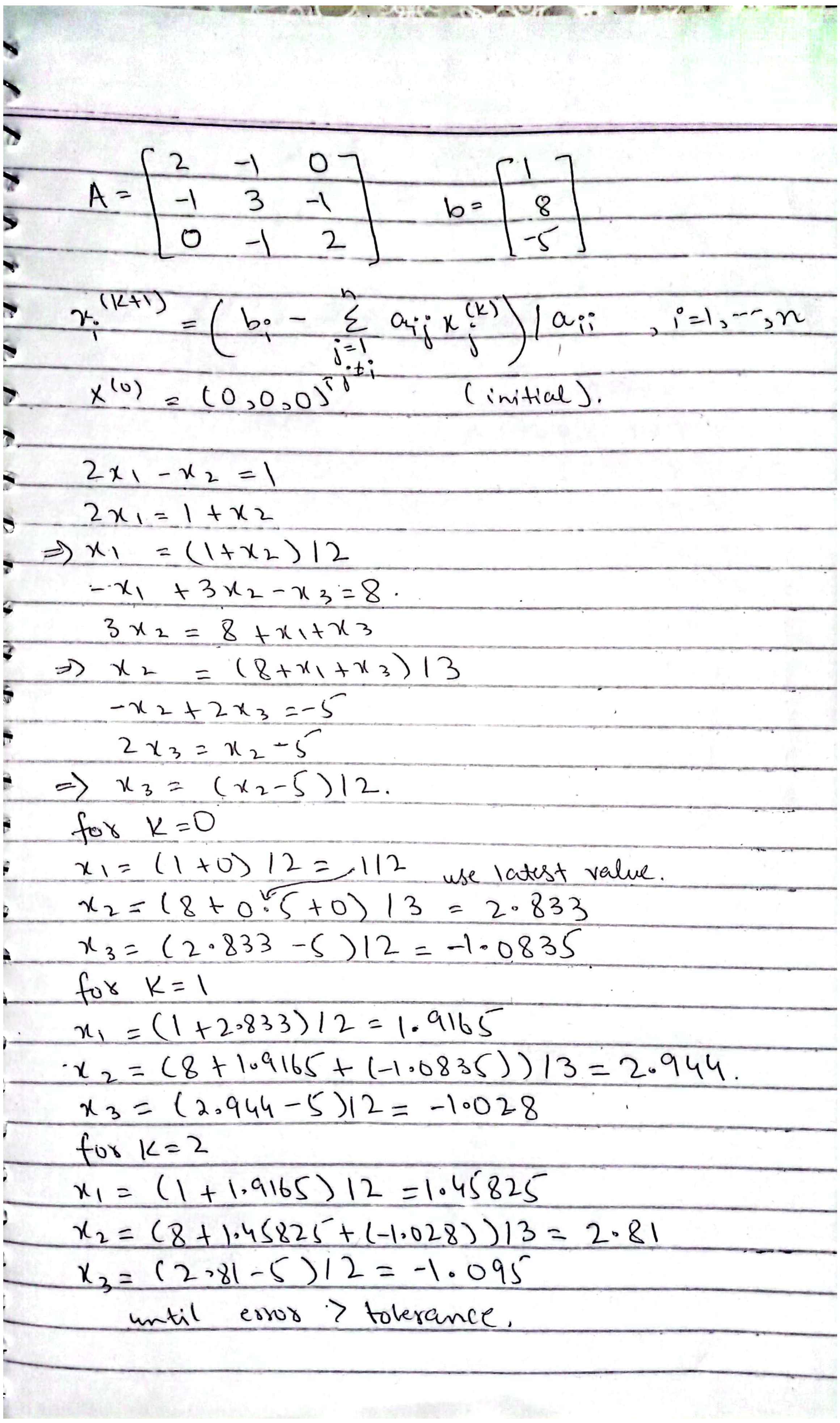
```
b = np.array([30., 4., 12.])
(LU,P) = spl.lu_factor(A)
x = spl.lu_solve((LU,P),b)
print(x)
```

Cholesky

```
A = np.array([[4.0, -1.0, 1.0],
              [-1.0, 4.25, 2.75],
              [1.0, 2.75, 3.5]])
b = np.array([4.0, 6.0, 7.25])
H = spl.cholesky(A)
print(H)
# We use the option lower=True since H is lower triangul
ar matrix
y = spl.solve_triangular(H, b, lower=True)
# We use the option trans=1 to solve the system H^Tx=y
x = spl.solve_triangular(H, y, trans=1, lower=True)
print(x)
```

▼ Iterative Methods

▼ Gauss Seidal Method



$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 8 \\ -5 \end{bmatrix}$$

$$x_i^{(k+1)} = \left(b_i - \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k)} \right) / a_{ii} \quad , i = 1, \dots, n$$

$x^{(0)} = (0, 0, 0)^T$ (initial).

$2x_1 - x_2 = 1$

$2x_1 = 1 + x_2$

$\Rightarrow x_1 = (1 + x_2)/2$

$-x_1 + 3x_2 - x_3 = 8.$

$3x_2 = 8 + x_1 + x_3$

$\Rightarrow x_2 = (8 + x_1 + x_3)/3$

$-x_2 + 2x_3 = -5$

$2x_3 = x_2 - 5$

$\Rightarrow x_3 = (x_2 - 5)/2.$

for $k = 0$

$x_1 = (1 + 0)/2 = 1/2$ use latest value.

$x_2 = (8 + 0.5 + 0)/3 = 2.833$

$x_3 = (2.833 - 5)/2 = -1.0835$

for $k = 1$

$x_1 = (1 + 2.833)/2 = 1.9165$

$x_2 = (8 + 1.9165 + (-1.0835))/3 = 2.944.$

$x_3 = (2.944 - 5)/2 = -1.028$

for $k = 2$

$x_1 = (1 + 1.9165)/2 = 1.45825$

$x_2 = (8 + 1.45825 + (-1.028))/3 = 2.81$

$x_3 = (2.81 - 5)/2 = -1.095$

until error > tolerance.

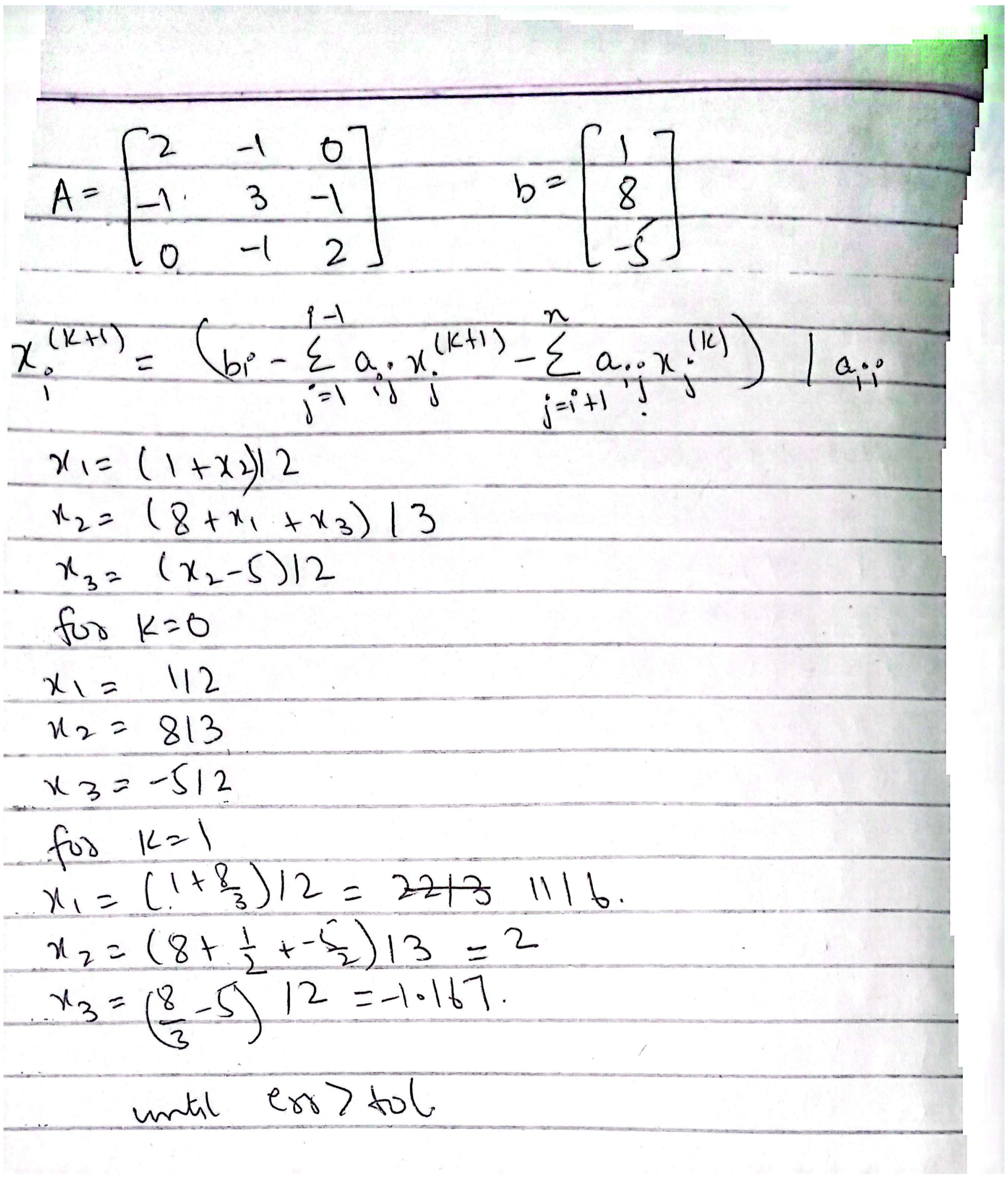
```
def gauss_seidel(A, b, x, tol = 1.e-5, maxit = 100):
    n = len(b)
    err = 1.0
    iters = 0
    # Initialize the solution with the initial guess
    xnew = np.zeros_like(x)
    # Extract the lower triangular part of A
    M = np.tril(A)
    # Construct the upper triangular part of A
    U = A - M
    while (err > tol and iters < maxit):
        iters += 1
        # Compute the new approximation
        xnew = np.dot(npl.inv(M), b - np.dot(U, x))
        # Estimate convergence
        err = npl.norm(xnew-x,np.inf)
        x = np.copy(xnew)
    print('iterations required for convergence:', iters)
    return x
```

```
A = np.array([[2.0, -1.0,  0.0],
              [-1.0, 3.0, -1.0],
              [0.0, -1.0,  2.0]])
b = np.array([1.0, 8.0, -5.0])
x = np.zeros(3)
x = gauss_seidel(A, b, x, 1.e-5, 100)
print(x)
```

Second implementation

```
def gauss_seidel(A, b, x, tol = 1.e-5, maxit = 100):
    n = len(b)
    err = 1.0
    iters = 0
    xnew = np.zeros_like(x)
    while (err > tol and iters < maxit):
        iters += 1
        for i in range(n):
            s1 = np.dot(A[i, :i], xnew[:i])
            s2 = np.dot(A[i, i + 1:], x[i + 1:])
            xnew[i] = (b[i] - s1 - s2) / A[i, i]
        err = npl.norm(xnew-x,np.inf)
        x = np.copy(xnew)
    print('iterations required for convergence', iters)
    return x
```

▼ Jacobi Method



$$A = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 8 \\ -5 \end{bmatrix}$$

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}\right) / a_{ii}$$

$x_1 = (1 + x_2)/2$

$x_2 = (8 + x_1 + x_3)/3$

$x_3 = (x_2 - 5)/2$

for $k=0$

$x_1 = 1/2$

$x_2 = 8/3$

$x_3 = -5/2$

for $k=1$

$x_1 = (1 + \frac{8}{3})/2 = \cancel{22/3}\ 11/6.$

$x_2 = (8 + \frac{1}{2} + -\frac{5}{2})/3 = 2$

$x_3 = (\frac{8}{3} - 5)/2 = -1.167.$

until err > tol

```
import numpy.linalg as npl


def jacobi(A, b, x, tol = 1.e-5, maxit = 100):
    d = np.copy(np.diag(A))
    np.fill_diagonal(A,0.0)
    err = 1.0
    iters = 0
    while (err > tol and iters < maxit):
        iters += 1
        xnew = (b - np.dot(A,x)) / d
        err = npl.norm(xnew-x,np.inf)
        x = np.copy(xnew)
    print('iterations required for convergence:', iters)
    return x
```

```
A = np.array([[2.0, -1.0,  0.0],
              [-1.0, 3.0, -1.0],
              [0.0, -1.0,  2.0]])
b = np.array([1.0, 8.0, -5.0])
x = np.zeros(3)
x = jacobi(A, b, x, 1.e-5, 100)
print(x)
```

▼ Sparse Gauss-Seidal Implementation

Sparse Matrices

```
from scipy import sparse

S = np.array([[ 2.0,-1.0, 0.0, 0.0],
           [-1.0, 2.0,-1.0, 0.0],
           [ 0.0,-1.0, 2.0,-1.0],
           [ 0.0, 0.0,-1.0, 2.0]])

A = sparse.coo_matrix(S)

print(A)
```

```
import numpy as np

A = np.array([[10, 20,  0,  0,  0,  0],
              [ 0, 30,  0, 40,  0,  0],
              [ 0,  0, 50, 60, 70,  0],
              [ 0,  0,  0,  0,  0, 80]])
S = sparse.csr_matrix(A)
print(S.data[:])
print(S.indptr[:])
print(S.indices[:])
```

```
def sp_gauss_seidel(S, b, x, tol = 1.e-5, maxit = 100):
    n = len(b)
    xnew = np.zeros_like(x) # create new vector
    D = sparse.spdiags(S, 0, n, n)
    L = sparse.tril(S, 0, format = 'csc')
    U = sparse.triu(S, 1, format = 'csc')
    G = -(sparse.linalg.inv(L)).dot(U)
    c = (sparse.linalg.inv(L)).dot(b)
    iters = 0
    err = 1.0
    while (err > tol and iters < maxit):
        iters += 1
        xnew = G*x + c
        err = npl.norm(xnew-x, np.inf)
        x = xnew
    print('iterations required for convergence:', iters)
    return x


x = np.zeros(len(b))
x = sp_gauss_seidel(S, b, x, tol = 1.e-5, maxit = 100)
print(x)
```

Avoiding inverse matrix:

```
def sp_gauss_seidel(S, b, x, tol = 1.e-5, maxit = 100):
    n = len(b)
    err = 1.0
    iters = 0
    D = S.diagonal()
    xnew = np.zeros_like(x)
    while (err > tol and iters < maxit):
        iters += 1
        for i in range(n):
            rowstart = S.indptr[i]
            rowend = S.indptr[i+1]
            z = np.copy(x)
            z[:i] = xnew[:i]
            z[i]=0.0
            s = np.dot(S.data[rowstart:rowend], z[S.indi
ces[rowstart:rowend]])
            xnew[i] = (b[i] - s) / D[i]
        err = np.linalg.norm(xnew-x,np.inf)
        x = np.copy(xnew)
    print('iterations required for convergence:', iters)
    return x
```

```
A = np.array([[ 2.0,-1.0, 0.0, 0.0],
              [-1.0, 2.0,-1.0, 0.0],
              [ 0.0,-1.0, 2.0,-1.0],
              [ 0.0, 0.0,-1.0, 2.0]])
S = sparse.coo_matrix(A)
x = np.ones(4)
S = S.tocsr()
b = S.dot(x)
x = np.zeros(len(b))
x = sp_gauss_seidel(S, b, x, tol = 1.e-5, maxit = 100)
```

```
print(x)



import matplotlib.pyplot as plt

plt.spy(S)
plt.show()
```

```
#A conventional way to characterize a matrix as sparse i
s to consider the sparsity index I_s=N_s/N where N_s is
the number of zero entries of $S$. The closer this index
is to 1 the more sparse is the matrix S.
#In our example
Is = 1-S.count_nonzero()/np.size(S.toarray())
print(Is)
```

Application in electric circuits

Kirchoff's Law

```
A = np.array([[ 1.0,-1.0, 0.0, 0.0,-1.0, 0.0],
              [ 0.0, 1.0,-1.0,-1.0, 0.0, 0.0],
              [ 0.0, 0.0, 0.0, 1.0, 1.0,-1.0],
              [ 0.0, 0.0, 0.0, 0.0, 1.0, 2.0],
              [ 0.0, 2.0, 0.0, 2.0,-1.0, 0.0],
              [ 0.0, 0.0, 1.0,-2.0, 0.0,-2.0]])
b = np.array([0, 0, 0, 10, 0, 0])
S = sparse.csc_matrix(A)
I = linalg.spsolve(S,b)
print('I =', I)
```