Solving Systems Of Linear Equations

by Loïc Quertenmont, PhD

LINFO1113 - 2019-2020

Programme

Cours	1	Librairies mathématiques, représentation des nombres en Python et erreurs liées
Cours	**2,3,4**	**Résolution des systèmes linéaires**
Cours	5,6	Interpolation et Régression Linéaires
Cours	7	Zéro d'équation
Cours	8,9	Différentiation numérique
Cours	10	Intégration numérique
Cours	11,12	Introduction à l'optimisation
Cours	13	Rappel / Répétition

Outline

- **Matrix Algebra (Reminder)**
- **Introduction**
- **Gauss Elimination Method**
- **LU Decomposition Methods**
- **Symmetric and Banded Coefficient Matrices**
- **Iterative Methods**
- **Other Remarks**

LU Decomposition

LU Decomposition

- LU decomposition is a different way to solve $\mathbf{AX} = \mathbf{B}$
- Matrix $\mathbf{A}$ is first decomposed into a lower triangular matrix $\mathbf{L}$ and an upper triangular matrix $\mathbf{U}$ with $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$:

$$A = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{pmatrix} \cdot \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

- Why is that useful? Let's assume for a moment that we know how to find $\mathbf{L}$ and $\mathbf{U}$…

LU Decomposition (2)

- Instead of solving $\mathbf{AX} = \mathbf{B}$ with $O(n^3)$ we solve:

$$\mathbf{L\,U\,X} = \mathbf{B}$$

- We can solve $\mathbf{L\,Y} = \mathbf{B}$ to find $\mathbf{Y} = \mathbf{UX}$
  - But $\mathbf{L}$ is a lower triangular matrix!
    → We get $\mathbf{Y}$ just by forward substitution with $O(n^2)$
- When we have $\mathbf{Y}$ we can solve $\mathbf{UX} = \mathbf{Y}$ to find $\mathbf{X}$
  - But $\mathbf{U}$ is an upper triangular matrix!
    → We get $\mathbf{X}$ just by back substitution with $O(n^2)$
-

  Total complexity: $O(n^2)$ instead of $O(n^3)$

LU Decomposition (3)

- There is an easy method to find $\mathbf{L}$ and $\mathbf{U}$ for $\mathbf{A}$
  - Discussed in the coming slides
  - Complexity as GE: $O(n^3)$
- What is the advantage of LU decomposition over Gaussian Elimination with $O(n^3)$?
  - We only have to calculate $\mathbf{L}$ and $\mathbf{U}$ for $\mathbf{A}$ once
  - Then we can solve $\mathbf{AX} = \mathbf{B}$ for any $\mathbf{B}$ in $O(n^2)$
  - Good for problems where $\mathbf{B}$ changes over time and $\mathbf{AX} = \mathbf{B}$ has to be solved repeatedly
- But: How can we find the decomposition ?

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \, \mathbf{?}$$

LU Decomposition (4)

- Decomposition of **A** in $\mathbf{L} \cdot \mathbf{U}$ is not unique

- Endless possibilities unless some constraints are placed on **L** or **U**

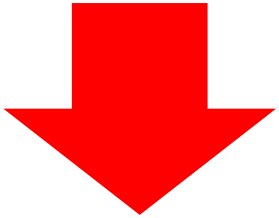
Name	Constraints
Doolittle's decomposition	$L_{ii} = 1, \quad i = 1, 2, \ldots, n$
Crout's decomposition	$U_{ii} = 1, \quad i = 1, 2, \ldots, n$
Choleski's decomposition	$\mathbf{L} = \mathbf{U}^T$

Table 2.2. Three Commonly Used Decompositions

Doolitle's Decomposition (1)

- Example for 3x3

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$



A = LU

$$\mathbf{A} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ U_{11}L_{21} & U_{12}L_{21} + U_{22} & U_{13}L_{21} + U_{23} \\ U_{11}L_{31} & U_{12}L_{31} + U_{22}L_{32} & U_{13}L_{31} + U_{23}L_{32} + U_{33} \end{bmatrix}$$

Doolitle's Decomposition (2)

- Apply Gauss Elimination
  - **Pivot is A_{11}**
    - row 2 ← row 2 − L_{21} × row 1 (eliminatesA_{21})
    - row 3 ← row 3 − L_{31} × row 1 (eliminatesA_{31})

$$\mathbf{A}' = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & U_{22}L_{32} & U_{23}L_{32} + U_{33} \end{bmatrix}$$

  - **Pivot is A_{22}**
    - row 3 ← row 3 − L_{32} × row 2 (eliminatesA_{32})

$$\mathbf{A}'' = \mathbf{U} = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

Doolitle's Decomposition (3)

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -1 \\ 3 & -1 & 1 \end{pmatrix} \qquad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{pmatrix}$$

R2→R2 - 0xR1

$$\begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -1 \\ 3 & -1 & 1 \end{pmatrix} \qquad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ \mathbf{0} & 1 & 0 \\ ? & ? & 1 \end{pmatrix}$$

R3→R3 - 3xR1

$$\begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -1 \\ 0 & -4 & 7 \end{pmatrix} \qquad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathbf{3} & ? & 1 \end{pmatrix}$$

R3→R3 – (-4)xR2

$$\mathbf{U} = \begin{pmatrix} 1 & 1 & -2 \\ 0 & 1 & -1 \\ 0 & 0 & 3 \end{pmatrix} \qquad \mathbf{L} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & \mathbf{-4} & 1 \end{pmatrix}$$

Doolitle's Decomposition (4)

- The matrix U is identical to the upper triangular matrix that results from Gauss elimination.

- The off-diagonal elements of L are the pivot equation multipliers used during Gauss elimination; that is,
  - Lij is the multiplier that eliminated Aij .

Doolitle's Decomposition (5)

- Memory optimization
  - L and U matrices can be stored together
  - It is understood that L diagonal is always 1

$$[\mathbf{L}\backslash\mathbf{U}] = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{bmatrix}$$

  - Actually, the resulting matrix L\U can even be the input matrix A (same memory)
    - L_{ij} replace A_{ij} during gauss elimination

Doolitle's Decomposition (6)

- The algorithm for Doolittle's decomposition is thus identical to the Gauss elimination

- Except that each multiplier λ is now stored in the lower triangular portion of A

```
def LUdecomp(a):
    n = len(a)
    for k in range(0,n-1):
        for i in range(k+1,n):
           if a[i,k] != 0.0:
               lam = a [i,k]/a[k,k]
               a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
               a[i,k] = lam
    return a
```

```
def gaussElimin(a,b):
    n = len(b)
  # Elimination Phase
    for k in range(0,n-1): #iterate on rows
        for i in range(k+1,n): #iterate on rows below pivot
           if a[i,k] != 0.0:
               lam = a [i,k]/a[k,k]
               a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
```

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

executed in 16ms, finished 10:22:58 2019-09-19

```
array([[ 1,  1, -2],
       [ 0,  1, -1],
       [ 3, -4,  3]])
```

Doolitle's Decomposition (7)

- Solving the system after decomposition

$$\mathbf{L\,U\,X = B}$$

1. $\mathbf{L\,Y = B}$ (forward substitution)

$$\begin{aligned} y_1 &= b_1 \\ L_{21}y_1 + y_2 &= b_2 \\ &\vdots \\ L_{k1}y_1 + L_{k2}y_2 + \cdots + L_{k,k-1}y_{k-1} + y_k &= b_k \end{aligned}$$

$$y_k = b_k - \sum_{j=1}^{k-1} L_{kj}y_j, \quad k = 2, 3, \ldots, n$$

```
y[0] = b[0]
for k in range(1,n):
    y[k] = b[k] - dot(a[k,0:k],y[0:k])
```

2. $\mathbf{UX = Y}$ (backward substitution)
   - Identical to what was used in gauss substitution

Doolitle's Decomposition (8)

```
def LUdecomp(a):
    n = len(a)
    for k in range(0,n-1):
        for i in range(k+1,n):
           if a[i,k] != 0.0:
               lam = a [i,k]/a[k,k]
               a[i,k+1:n] = a[i,k+1:n] - lam*a[k,k+1:n]
               a[i,k] = lam
    return a #Return the combined matrix LU

def LUsolve(a,b):
    #a contains the matrix LU
    n = len(a)
    #LY=B forward substitution (b is replaced with the solution of Y)
    for k in range(1,n):
        b[k] = b[k] - np.dot(a[k,0:k],b[0:k])
    #UX=Y backward substitution (b=Y is replaced with the solution of X)
    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
```

LU Decomposition

Name	Constraints
Doolittle's decomposition	$L_{ii} = 1, \quad i = 1, 2, \ldots, n$
Crout's decomposition	$U_{ii} = 1, \quad i = 1, 2, \ldots, n$
Choleski's decomposition	$\mathbf{L} = \mathbf{U}^T$

Table 2.2. Three Commonly Used Decompositions

- **Crout's** and **Doolittle's** are very similar
  - Performances are identical
  - Not much interest to explore Crout's further

- **Choleski's**
  - ~ 2x faster than Doolittle's by exploiting symmetry

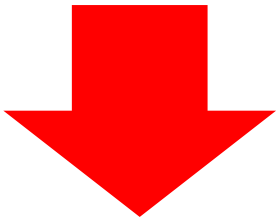
Choleski's Decomposition (1)

- **Choleski's:** $\boldsymbol{A = LL^T}$
  - **A must be <u>symmetric</u>**
    - Because LL^T is always symmetric s
  - **A must be <u>positive definite</u>**
    - Because decomposition requires to take square root of some elements of A
- By exploiting the symmetry, Choleski's achieve a complexity 2x smaller than Doolitle's

Choleski's Decomposition (2)

- Example for 3x3

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 \\ L_{21} & L_{22} & 0 \\ L_{31} & L_{32} & L_{33} \end{bmatrix} \begin{bmatrix} L_{11} & L_{21} & L_{31} \\ 0 & L_{22} & L_{32} \\ 0 & 0 & L_{33} \end{bmatrix}$$



$$\mathbf{A = LL^T}$$

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11}^2 & L_{11}L_{21} & L_{11}L_{31} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 & L_{21}L_{31} + L_{22}L_{32} \\ L_{11}L_{31} & L_{21}L_{31} + L_{22}L_{32} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{bmatrix}$$

- **Note: A is indeed symmetric,**

Choleski's Decomposition (2)

- Solving the equations

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} L_{11}^2 & L_{11}L_{21} & L_{11}L_{31} \\ L_{11}L_{21} & L_{21}^2 + L_{22}^2 & L_{21}L_{31} + L_{22}L_{32} \\ L_{11}L_{31} & L_{21}L_{31} + L_{22}L_{32} & L_{31}^2 + L_{32}^2 + L_{33}^2 \end{bmatrix}$$

$$A_{11} = L_{11}^2 \qquad L_{11} = \sqrt{A_{11}}$$

$$A_{21} = L_{11}L_{21} \qquad L_{21} = A_{21}/L_{11}$$

$$A_{31} = L_{11}L_{31} \qquad L_{31} = A_{31}/L_{11}$$

$$A_{22} = L_{21}^2 + L_{22}^2 \qquad L_{22} = \sqrt{A_{22} - L_{21}^2}$$

$$A_{32} = L_{21}L_{31} + L_{22}L_{32} \qquad L_{32} = (A_{32} - L_{21}L_{31})/L_{22}$$

$$A_{33} = L_{31}^2 + L_{32}^2 + L_{33}^2 \qquad L_{33} = \sqrt{A_{33} - L_{31}^2 - L_{32}^2}$$

Generalization: $A_{ij} = (\mathbf{L}\mathbf{L}^T)_{ij} = L_{i1}L_{j1} + L_{i2}L_{j2} + \cdots + L_{ij}L_{jj} \qquad i \geq j$

Choleski's Decomposition (3)

Example:

$$\mathbf{A} = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{pmatrix} \qquad \mathbf{L} = \begin{pmatrix} ? & 0 & 0 \\ ? & ? & 0 \\ ? & ? & ? \end{pmatrix}$$

$$L_{11} = \sqrt{A_{11}} = \sqrt{2}$$

$$L_{21} = A_{21}/L_{11} = -1/\sqrt{2}$$

$$L_{22} = \sqrt{A_{22} - L_{21}^2} = \sqrt{2 - (1/2)} = \sqrt{3/2}$$

$$L_{31} = A_{31}/L_{11} = 0/\sqrt{2}$$

$$L_{32} = (A_{32} - L_{21}L_{31})/L_{22} = \frac{-1-0}{\sqrt{3/2}} = -\sqrt{2/3}$$

$$L_{33} = \sqrt{A_{33} - L_{31}^2 - L_{32}^2} = \sqrt{2 - 0 - 2/3} = \sqrt{4/3}$$

Choleski's Decomposition (4)

$$(\mathbf{L}\mathbf{L}^T)_{ij} = L_{i1}L_{j1} + L_{i2}L_{j2} + \cdots + L_{ij}L_{jj} = \sum_{k=1}^{j} L_{ik}L_{jk}, \quad i \geq j$$

$$A_{ij} = \sum_{k=1}^{j} L_{ik}L_{jk}, \quad i = j, j+1, \ldots, n, \quad j = 1, 2, \ldots, n$$

- Solving for the first column (j=1)

$$L_{11} = \sqrt{A_{11}} \qquad L_{i1} = A_{i1}/L_{11}, \quad i = 2, 3, \ldots, n$$

- Solving for other columns:
  - Unknowns are $\mathbf{L_{ij}}$ (with i ≥ j) the other elements have already been computed
  - Running the sum only up to j-1 → $A_{ij} = \sum_{k=1}^{j-1} L_{ik}L_{jk} + L_{ij}L_{jj}$
  - **i==j → diagonal:** $L_{jj} = \sqrt{A_{jj} - \sum_{k=1}^{j-1} L_{jk}^2}, \quad j = 2, 3, \ldots, n$
  - **i!=j → off-diagonal:** $L_{ij} = \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik}L_{jk}\right)/L_{jj} \quad \begin{matrix} j = 2, 3, \ldots, n-1 \\ i = j+1, j+2, \ldots, n \end{matrix}$

Choleski's Decomposition (5)

- Algorithm
  - Remark, $\mathbf{A_{ij}}$ only appear in the formula for $\mathbf{L_{ij}}$
  - We can store the value of $\mathbf{L_{ij}}$ in place of $\mathbf{A_{ij}}$ **(lower triangle)**
    - A upper triangle remains untouched → should be set to 0

```python
import numpy as np
import math

def choleski(a):
    n = len(a)
    for k in range(n):
        try:
            a[k,k] = math.sqrt(a[k,k] - np.dot(a[k,0:k],a[k,0:k]))
        except ValueError:
            raise Exception('Matrix is not positive definite')
        for i in range(k+1,n):
            a[i,k] = (a[i,k] - np.dot(a[i,0:k],a[k,0:k]))/a[k,k]
    for k in range(1,n): a[0:k,k] = 0.0 #erase upper triangle
    return a
```

```python
def choleskiSol(L,b):
    n = len(b)
  # Solution of [L]{y} = {b}
    for k in range(n):
        b[k] = (b[k] - np.dot(L[k,0:k],b[0:k]))/L[k,k]
  # Solution of [L_transpose]{x} = {y}
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - np.dot(L[k+1:n,k],b[k+1:n]))/L[k,k]
    return b
```

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

b = np.array([-3,-1,4])
display(b)

L = choleski(A)
display(L)

x = choleskiSol(L,b)
display(x)
```

executed in 20ms, finished 12:05:28 2019-09-19

```
array([[ 2, -1,  0],
       [-1,  2, -1],
       [ 0, -1,  2]])

array([-3, -1,  4])

array([[ 1,  0,  0],
       [-1,  1,  0],
       [ 0, -1,  1]])

array([-7, -4,  0])
```

Additional Remarks on LU

- LU decomposition
  - Row Pivoting is also possible, but we need to
    - keep track of row inversions made during LU decomposition
    - apply same transformation on B when we solve the system
  - Better performance (CPU/memory) can be achieved by exploiting further specific symmetry/properties of the A matrix

    - For example symmetric matrix,
      Can be decomposed into:

$$\mathbf{A} = \mathbf{LU} = \mathbf{LDL}^T$$

      - **In the case D=I we get choleski decomposition**

- Matrix inversion → loss of symmetry

Tri-diagonal Matrices (1)

- Useful example of banded matrix (tri-diagonal)
  - X means none zero element.
  - Most of the elements are null.

- If A is decomposed in LU,
  Both L and U retained the banded structure

$$\mathbf{A} = \begin{bmatrix} X & X & 0 & 0 & 0 \\ X & X & X & 0 & 0 \\ 0 & X & X & X & 0 \\ 0 & 0 & X & X & X \\ 0 & 0 & 0 & X & X \end{bmatrix} \quad \mathbf{L} = \begin{bmatrix} X & 0 & 0 & 0 & 0 \\ X & X & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 \\ 0 & 0 & X & X & 0 \\ 0 & 0 & 0 & X & X \end{bmatrix} \quad \mathbf{U} = \begin{bmatrix} X & X & 0 & 0 & 0 \\ 0 & X & X & 0 & 0 \\ 0 & 0 & X & X & 0 \\ 0 & 0 & 0 & X & X \\ 0 & 0 & 0 & 0 & X \end{bmatrix}$$

Tri-diagonal Matrices (2)

- Usage of the symmetry
  - Can save both **CPU** and **Memory**
- Memory space of a n x n matrix of dimension ?
  - n^2 * 4 (float32) bytes
- Memory space of a tri-diagonal matrix of n x n size ?
  - (3n – 2) * 4 (float32) bytes
- For n=100 → 3% of the memory, for n=1000 → 0.3%

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & 0 & \cdots & 0 \\ c_1 & d_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_2 & d_3 & e_3 & \cdots & 0 \\ 0 & 0 & c_3 & d_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n-1} & d_n \end{bmatrix} \rightarrow \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} \quad \mathbf{d} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_{n-1} \\ d_n \end{bmatrix} \quad \mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_{n-1} \end{bmatrix}$$

Tri-diagonal Matrices (3)

- LU decomposition via Doolitle's

$$\mathbf{A} = \begin{bmatrix} d_1 & e_1 & 0 & 0 & \cdots & 0 \\ c_1 & d_2 & e_2 & 0 & \cdots & 0 \\ 0 & c_2 & d_3 & e_3 & \cdots & 0 \\ 0 & 0 & c_3 & d_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n-1} & d_n \end{bmatrix}$$

$$\begin{bmatrix} d_1 & e_1 & 0 & 0 & \cdots & 0 \\ 0 & d_2' & e_2' & 0 & \cdots & 0 \\ 0 & c_2 & d_3 & e_3 & \cdots & 0 \\ 0 & 0 & c_3 & d_4 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & c_{n-1} & d_n \end{bmatrix}$$

$$row_2 \rightarrow row_2 - \frac{c_1}{d_1} row_1$$

$$d_2' = d_2 - \frac{c_1}{d_1} e_1$$

$$e_2' = e_2$$

Generalisation

$$\text{row } k \leftarrow \text{row } k - (c_{k-1}/d_{k-1}) \times \text{row } (k-1),$$

$$d_k \leftarrow d_k - (c_{k-1}/d_{k-1}) e_{k-1}$$

$$\lambda = c_{k-1}/d_{k-1} \quad \text{Stored in} \rightarrow L_{k,k-1} = c_{k-1}$$

Tri-diagonal Matrices (4)

- Solving the system **LU x = b**
  - **L y = b** with **U x = y**
- **forward substitution**

L y = b

$$\left[\mathbf{L} \mid \mathbf{b}\right] = \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & \cdots & 0 & b_1 \\ c_1 & 1 & 0 & 0 & \cdots & 0 & b_2 \\ 0 & c_2 & 1 & 0 & \cdots & 0 & b_3 \\ 0 & 0 & c_3 & 1 & \ldots & 0 & b_4 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & 1 & b_n \end{array}\right]$$

- **backward substitution**

U x = y

$$\left[\mathbf{U} \mid \mathbf{y}\right] = \left[\begin{array}{cccccc|c} d_1 & e_1 & 0 & \cdots & 0 & 0 & y_1 \\ 0 & d_2 & e_2 & \cdots & 0 & 0 & y_2 \\ 0 & 0 & d_3 & \cdots & 0 & 0 & y_3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & d_{n-1} & e_{n-1} & y_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & d_n & y_n \end{array}\right]$$

Tri-diagonal Matrices (5)

- Solving the system **LU x = b**
  - **L y = b** with **U x = y**
- **forward substitution**

L y = b

$$\left[\mathbf{L} \mid \mathbf{b}\right] = \left[\begin{array}{cccccc|c} 1 & 0 & 0 & 0 & \cdots & 0 & b_1 \\ c_1 & 1 & 0 & 0 & \cdots & 0 & b_2 \\ 0 & c_2 & 1 & 0 & \cdots & 0 & b_3 \\ 0 & 0 & c_3 & 1 & \ldots & 0 & b_4 \\ \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & c_{n-1} & 1 & b_n \end{array}\right]$$

- **backward substitution**

U x = y

$$\left[\mathbf{U} \mid \mathbf{y}\right] = \left[\begin{array}{cccccc|c} d_1 & e_1 & 0 & \cdots & 0 & 0 & y_1 \\ 0 & d_2 & e_2 & \cdots & 0 & 0 & y_2 \\ 0 & 0 & d_3 & \cdots & 0 & 0 & y_3 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & d_{n-1} & e_{n-1} & y_{n-1} \\ 0 & 0 & 0 & \cdots & 0 & d_n & y_n \end{array}\right]$$

Tri-diagonal Matrices (6)

- Resulting Algortihm

```
def LUdecomp3(c,d,e):
    n = len(d)
    for k in range(1,n):
        lam = c[k-1]/d[k-1]
        d[k] = d[k] - lam*e[k-1]
        c[k-1] = lam
    return c,d,e
```

```
def LUsolve3(c,d,e,b):
    n = len(d)
    for k in range(1,n):
        b[k] = b[k] - c[k-1]*b[k-1]
    b[n-1] = b[n-1]/d[n-1]
    for k in range(n-2,-1,-1):
        b[k] = (b[k] - e[k]*b[k+1])/d[k]
    return b
```

```
LUc,LUd,LUe = LUdecomp3(c,d,e)
x3 = LUsolve3(LUc,LUd,LUe,b.copy())
x3
```

executed in 9ms, finished 11:11:42 2019-10-03

```
array([-0.16326511,  1.1632651 , -0.08163255,  0.30612248,  0.755102  ],
      dtype=float32)
```

verifying solution

```
np.matmul(A,x3)
```

executed in 9ms, finished 11:12:00 2019-10-03

```
array([1., 2., 3., 4., 5.], dtype=float32)
```

```
c = np.array([1, 2, 3, 4], np.float32)
d = np.array([1, 2, 3, 4, 5], np.float32)
e = np.array([1, 2, 3, 4], np.float32)
display("c", c)
display("d", d)
display("e", e)
```

```
'c'

array([1., 2., 3., 4.], dtype=float32)

'd'

array([1., 2., 3., 4., 5.], dtype=float32)

'e'

array([1., 2., 3., 4.], dtype=float32)
```

```
from scipy.sparse import diags
A = diags([c,d,e], [-1, 0, 1]).toarray().copy()
display("A", A)

b = np.array([1,2,3,4,5], np.float32)
display("b", b)
```

```
'A'

array([[1., 1., 0., 0., 0.],
       [1., 2., 2., 0., 0.],
       [0., 2., 3., 3., 0.],
       [0., 0., 3., 4., 4.],
       [0., 0., 0., 4., 5.]], dtype=float32)

'b'

array([1., 2., 3., 4., 5.], dtype=float32)
```

Final Remarks on LU decomposition

- Pivoting
  - Row swaps should be “remember” in LU decomposition and applied to b in the solving phase.
  - Draw backs for diagonal, banded, symmetric matrices. Pivoting would break the symmetry and it’s therefore not always helpful.

Iterative Methods

Iterative Methods

- Gaussian Elimination computes an exact solution for $AX = B$ if a solution exists (provided there are no floating-point errors)

- Iterative methods start with an initial approximation $X^{(0)}$ of the solution and then try to improve the approximation iteratively:

$$X^{(0)} \quad := \text{initial approximation}$$
$$\mathrm{X}^{(\mathrm{k}+1)} := \mathrm{f}(\mathrm{X}^{(\mathrm{k})})$$

- Of course, one hopes that the approximation $\mathrm{X}^{(\mathrm{k}+1)}$ converges to the exact solution X with increasing k

Jacobi Method

- We split the matrix A in two matrices D and R:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \underbrace{\begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}}_{\text{diagonal matrix D}} + \underbrace{\begin{pmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{pmatrix}}_{\text{R}}$$

Jacobi Method (2)

Transform the problem **Ax=b** by replacing **A=(D+R)**:

- $(\mathbf{D}+\mathbf{R})\,\mathbf{x}=\mathbf{b}$
- $\mathbf{Dx}+\mathbf{Rx}=\mathbf{b}$
- $\mathbf{Dx}=\mathbf{b}-\mathbf{Rx}$
- $\mathrm{x}=\mathrm{D}^{-1}\,(\mathbf{b}-\mathbf{Rx})$

The Jacobi Method uses the iteration formula:

$\mathbf{x}^{(0)}$:= initial approximation

$\mathbf{x}^{(k+1)} := \mathbf{D}^{-1}(\mathbf{b}-\mathbf{R}\mathbf{x}^{(k)})$

Jacobi Method (3)

- Note that the inverse of a diagonal matrix is

$$\mathbf{D}^{-1} = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix}^{-1} = \begin{pmatrix} 1/a_{11} & 0 & \cdots & 0 \\ 0 & 1/a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1/a_{nn} \end{pmatrix}$$

- Therefore, we can write $\mathbf{x}^{(\mathbf{k+1})} := \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(\mathbf{k})})$ as:

$$\mathrm{x}_\mathrm{i}^{(\mathrm{k+1})} := \frac{1}{\mathrm{a}_{\mathrm{ii}}}\left(\mathrm{b}_\mathrm{i} - \sum_{\mathrm{j}\neq\mathrm{i}} \mathrm{a}_{\mathrm{ij}}\mathrm{x}_\mathrm{j}^{(\mathrm{k})}\right) , \qquad i = 1,2,\ldots,n$$

Jacobi Method: Implementation

1. Choose initial guess $\mathbf{x}^{(\mathbf{0})}$, k:=0

2. Using current guess $\mathbf{x}^{(\boldsymbol{k})}$ calculate new guess $\mathbf{x}^{(\boldsymbol{k+1})}$:

$$x_i^{(k+1)} := \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), \qquad i = 1,2,\ldots,n$$

3. Repeat step 2 with k:=k+1 until difference$(\mathbf{x}^{(\mathbf{k+1})}, \mathbf{x}^{(\mathbf{k})}) < \varepsilon$

4. Convergence Error $:= \mathbf{A}\,\mathbf{x}^{(\mathbf{k+1})} - \mathbf{b}$

Example (Wikipedia)

$$10x_1 - x_2 + 2x_3 = 6$$
$$-x_1 + 11x_2 - x_3 + 3x_4 = 25$$
$$2x_1 - x_2 + 10x_3 - x_4 = -11$$
$$3x_2 - x_3 + 8x_4 = 15$$

Jacobi results:

$X^{(0)}$ = [0.0 0.0 0.0 0.0] ← our initial guess

$X^{(1)}$ = [0.6 2.27272727 -1.1 1.875]

$X^{(2)}$ = [1.04727273 1.71590909 -0.80522727 0.88522727]

…

$X^{(21)}$ = [0.99999999 2.00000002 -1.00000001 1.00000002]

$X^{(22)}$ = [1.0 1.99999999 -0.99999999 0.99999999]

Jacobi Method: Implementation

…
3. Repeat step 2 with k:=k+1 until difference$(\mathbf{x}^{(k+1)}, \mathbf{x}^{(k)}) < \varepsilon$
…

What does "difference(X,Y)" mean?

- There are different ways to define it, for example:

$$\text{difference(X,Y)} = \max_i |x_i - y_i|$$

$$\text{difference(X,Y)} = \sum_i (x_i - y_i)^2$$

- Sometimes, one stops too early because, for example, $X^{(6)}$ and $X^{(7)}$ are very close although $X^{(8)}$ is much better.
  We can use this instead:

$$\text{difference}(X^{(k+1)}, X^{(k-1)})$$

Jacobi Method: Convergence

- Gaussian Elimination works always if a solution exists, but the Jacobi method does not always converge!
- Convergence depends on the *spectral radius* of $\mathbf{A}$ which is quite expensive to calculate.
- We will not discuss that here… but:
- Sufficient (but not necessary condition) for convergence:

$$\textbf{A is strictly diagionally dominant: } |a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

Jacobi Method: Properties

- Complexity of one iteration: $O(n^2)$
- Why using Jacobi?
  - For matrices with $|a_{ii}| \gg \sum_{j \neq i} |a_{ij}|$ Jacobi can converge much faster than $O(n^3)$
  - For large sparse matrices, we can save only the non-zero elements. Make it possible to deal with very large matrices
  - Iterative method can be self-correcting regarding round-off errors. Errors in one cycle, can be corrected by the following iterations
  - $\mathrm{x}_i^{(k+1)} := \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right)$ can be parallelized. Give each computer one $x_i^{(k+1)}$ to calculate

- But
  - needs a little bit more space than Gaussian Elimination: two vectors $X^{(k)}$ and $X^{(k+1)}$ to store in memory (if not sparse)
  - Convergence is not always guarantee and can be slow

Gauss-Seidel Method

- This is a variation of the Jacobi method
- Instead of

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

- Use the already computed $x_{1..i-1}^{(k+1)}$ to compute $x_i^{(k+1)}$:

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

Gauss-Seidel Method: Properties

Comparison with Jacobi Method:

- Sometimes faster/slower than Jacobi, depending on $\mathbf{A}$
- Good:
  Only one vector $X^{(k)}$ needed to store. It will be overwritten by $X^{(k+1)}$ during the computation.

- Bad:
  Cannot be parallelized because the computation of

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

  needs $x_{i-1}^{(k+1)}, x_{i-2}^{(k+1)}, \ldots$

Successive Over-Relaxation (SOR)

- SOR tries to achieve faster conversion:

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

  with $1 < \omega < 2$

- Note that $\omega = 1$ is Gauss-Seidel
- Optimal value for the relaxation parameter ω depends again on the spectral properties of A. In practice, one chooses a value between 1.5 and 2
- **Under-Relaxation** $0 < \omega < 1$ can be used to stabilize a solution that slightly diverges

Optimal value for ω

- Taken from the reference book (not demonstrated here)

> Carry out k iterations with $\omega = 1$ ($k = 10$ is reasonable).
> Record $\Delta x^{(k)}$.
> Perform additional p iterations.
> Record $\Delta x^{(k+p)}$.
> Compute ω_{opt} from Eq. (2.36).
> Perform all subsequent iterations with $\omega = \omega_{\text{opt}}$.

- With:

$$\Delta x^{(k)} = \left| \mathbf{x}^{(k-1)} - \mathbf{x}^{(k)} \right|$$

- And

$$\omega_{\text{opt}} \approx \frac{2}{1 + \sqrt{1 - \left(\Delta x^{(k+p)} / \Delta x^{(k)} \right)^{1/p}}}$$

Algorithm

```
def gaussSeidel(iterEqs,x,tol = 1.0e-9):
    omega = 1.0
    k = 10
    p = 1
    for i in range(1,501):
        xOld = x.copy()
        x = iterEqs(x,omega)
        dx = np.sqrt(np.dot(x-xOld,x-xOld))
        print("running iteration %3d  dx=%6.2e" % (i,dx) )
        if dx < tol: return x,i,omega
      # Compute relaxation factor after k+p iterations
        if i == k: dx1 = dx
        if i == k + p:
            dx2 = dx
            omega = 2.0/(1.0 + math.sqrt(1.0 - (dx2/dx1)**(1.0/p)))
    print('Gauss-Seidel failed to converge')
```

```
def iterEqs(x,omega):
    n = len(x)
    for i in range(n):
        new  = b[i]
        for j in range(n):
            if(i==j):continue
            new -= A[i,j]*x[j]
        x[i] = ((1-omega)*x[i]) + ((omega/A[i,i]) * new)
    return x
```

```
array([[ 5.,  1., -2.],
       [ 0.,  5., -1.],
       [ 3., -1.,  5.]])

array([-3., -1.,  4.])

running iteration   1  dx=1.29e+00
running iteration   2  dx=5.91e-01
running iteration   3  dx=1.70e-01
running iteration   4  dx=4.82e-02
running iteration   5  dx=1.37e-02
running iteration   6  dx=3.90e-03
running iteration   7  dx=1.11e-03
running iteration   8  dx=3.16e-04
running iteration   9  dx=8.98e-05
running iteration  10  dx=2.55e-05
running iteration  11  dx=7.25e-06
running iteration  12  dx=2.28e-06
running iteration  13  dx=9.86e-07
running iteration  14  dx=4.86e-07
running iteration  15  dx=2.74e-07
running iteration  16  dx=1.74e-07
running iteration  17  dx=1.00e-07
running iteration  18  dx=4.74e-08
running iteration  19  dx=1.50e-08
running iteration  20  dx=0.00e+00
x [-0.2244898  -0.01360544  0.9319728 ]
i 20
omega 1.0834399612526664
```

Other Methods

Other methods

- A matrix can be decomposed in several ways
- **QR factorization**

$$\mathbf{A} = \mathbf{QR}$$

  - With Q an orthogonal Matrix: $\mathbf{Q}^{-1} = \mathbf{Q}^{T}$
  - With R an upper triangular Matrix (similar to U)
  - → $\boldsymbol{Rx = Q^{T}b}$.
  - Generally slower than LU factorization, but could be more stable
- **Singular Value Decomposition (SVD)**

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{V}^{T}$$

  - With U and V orthogonal matrices
  - If A is symmetric and postive define
    λ are the eigen value

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & 0 & \cdots \\ 0 & \lambda_2 & 0 & \cdots \\ 0 & 0 & \lambda_3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$