Análisis numérico

Facultad de Ciencias, UNAM
Solución de sistemas de ecuaciones lineales

Jorge Zavaleta Sánchez

Semestre 2022-1

Índice

Preliminares 1

Sistemas triángulares **3**
Sistemas triangulares inferiores 3
Sistemas triangulares superiores 5

Sistemas lineales **7**
Eliminación Gaussiana 7
Factorización LU 9
Sistemas tridiagonales 13

Preliminares

[1]:
```
# Modulos necesarios
import numpy as np

# Excepciones
class slsException(Exception):

    def __init__(self,valor):
        self.valor = str(valor)

    def __str__(self):
        errores = {"0":"Las dimensiones son diferentes. La matriz debe ser␣
 ↪cuadrada.",
                   "1":"Los elementos de la diagonal son cero. La matriz no es␣
 ↪invertible.",
                   "2":"Los elementos de la diagonal son cero."}
        if self.valor in errores:
            return f"Error {self.valor}. {errores[self.valor]}"
        else:
            return "Error no clasificado"
```

[2]:
```python
# Ejemplos
print("Sistema triangular inferior")
L = np.array([[1,0,0],[4,1,0],[4,0.5,1]])
bl = np.array([3,6,10])
print(f"L = \n{L}",f"b = \n{bl}",sep="\n\n")

print("\nSistema triangular superior")
U = np.array([[1,2,2],[0,-4,-6],[0,0,-1]])
bu = np.array([3,-6,1])
print(f"U = \n{U}",f"b = \n{bu}",sep="\n\n")

print("\nSistema triangular superior")
A = np.array([[1,2,2],[4,4,2],[4,6,4]])
b = np.array([3,6,10])
print(f"A = \n{A}",f"b = \n{b}",sep="\n\n")
```

```
Sistema triangular inferior
L =
[[1.  0.  0. ]
 [4.  1.  0. ]
 [4.  0.5 1. ]]

b =
[ 3  6 10]

Sistema triangular superior
U =
[[ 1  2  2]
 [ 0 -4 -6]
 [ 0  0 -1]]

b =
[ 3 -6  1]

Sistema triangular superior
A =
[[1 2 2]
 [4 4 2]
 [4 6 4]]

b =
[ 3  6 10]
```

Sistemas triángulares

Sistemas triangulares inferiores

Se considera $A \in \mathbb{R}^{n\times n}$ una matriz **triangular inferior**, *i.e.*, con sus entradas todas ceros por encima de la diagonal principal ($a_{ij} = 0\ \forall i < j$) y el sistema

$$A\boldsymbol{x} = \begin{pmatrix} a_{11} & 0 & \cdots & 0 \\ a_{21} & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \boldsymbol{b}.$$

Suponiendo que $a_{ii} \neq 0\ \forall i = 1, \ldots, n$, la fómula para determinar el vector solución $\boldsymbol{x}$ está dada por

$$x_1 = \frac{b_1}{a_{11}}$$

$$x_i = \frac{b_i - \displaystyle\sum_{k=1}^{i-1} a_{ik}x_k}{a_{ii}} \quad i = 2, \ldots, n$$

que se conoce como **sustitución hacia adelante** (*forward substitution*), ya que la solución se va calculando de x_1 a x_n.

[3]:
```
# Implementacion 1
def fsubs(L,b):
    n = L.shape[0]
    x = np.empty(b.shape,dtype = "float64")
    for i in range(n):
        s = 0
        for k in range(i):
            s += L[i,k]*x[k]
        x[i] = (b[i] - s)/L[i,i]
    return x
```

[4]:
```
#Prueba
x = fsubs(L,bl)
print(x,np.allclose(L@x,bl))
```

```
[ 3. -6.  1.] True
```

Para la modificación se usará el lado derecho para ir acumulando los términos en la suma que se puedan calcular conforme se determinen los x_i. En la primera iteración, es decir $i = 1$ se tiene que:

$$x_1 = \frac{b_1}{a_{11}}$$

y para $2 \leq i \leq n$, ya conociendo x_1 se tiene que

$$x_i = \frac{b_i - a_{i2}x_1 - \sum_{k=2}^{i-1} a_{ik}x_k}{a_{ii}}$$

donde podemos reescribir

$$b_i^{(2)} = b_i^{(1)} - a_{i2}x_1 \quad \text{considerando} \quad b_i^{(1)} = b_i$$

Así en la segunda iteración ($i = 2$)

$$x_2 = \frac{b_2^{(2)}}{a_{22}} \quad y \quad x_i = \frac{b_i^{(2)} - \sum_{k=2}^{i-1} a_{ik}x_k}{a_{ii}} \quad \text{para} \quad 3 \leq i \leq n$$

En general, en la iteración $j-1$ ya conociendo x_{j-1}, si se toma $b_i^{(j)} = b_i^{(j-1)} - a_{ij}x_{j-1}$ se tendrá que en la iteración j

$$x_j = \frac{b_j^{(j)}}{a_{jj}} \quad y \quad x_i = \frac{b_i^{(j)} - \sum_{k=j}^{i-1} a_{ik}x_k}{a_{ii}} \quad \text{para} \quad j < i \leq n$$

De esta forma, tomando $b_j^{(j)} = b_j$ y $b_i^{(j)} = b_i$, para $j = 1, \ldots, n$. 1. Se calcula

$$x_j = \frac{b_j}{a_{jj}}$$

. 2. Para $i = j+1, \ldots, n$ se actualiza el lado derecho con

$$b_i = b_i - a_{ij}x_j$$

.

Este es el *algoritmo 1* que se implementa a continuación.

[5]:
```
# Implemetacion 2
def STI(L,rhs):
    m,n = L.shape
    if m != n:
        raise slsException(0)
    b = rhs.copy()
    x = np.empty((n,),dtype = "float64")
    for j in range(n):
        if L[j,j] == 0:
            raise slsException(1)
        x[j] = b[j]/L[j,j]
        for i in range(j+1,n):
```

```
            b[i] = b[i] - L[i,j]*x[j]
    return x
```

[6]:
```
#Prueba
x = STI(L,bl)
print(x,np.allclose(L@x,bl))
```

```
[ 3. -6.  1.] True
```

Sistemas triangulares superiores

Se considera $A \in \mathbb{R}^{n\times n}$ una matriz **triangular superior**, es decir, que las entradas de A debajo de la diagonal principal son cero ($a_{ij} = 0 \ \forall i > j$). De esta forma el sistema queda de la forma

$$A\boldsymbol{x} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ 0 & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} = \boldsymbol{b}.$$

Suponiendo que $a_{ii} \neq 0 \ \forall i = 1, \ldots, n$, se obtienen la fórmula

$$x_n = \frac{b_n}{a_{nn}}$$

$$x_i = \frac{b_i - \displaystyle\sum_{k=i+1}^{n} a_{ik}x_k}{a_{ii}} \quad i = n-1, \ldots, 1$$

la cual se conoce como **sustitución hacia atrás** (*backward subtitution*), ya que la solución se va calculando de x_n a x_1. Un análisis similar se puede realizar para obtener el psudocódigo descrito en el *algoritmo 2* que es el que se implementa a continuación.

[7]:
```
#Implementacion
def STS(U,rhs):
    m,n = U.shape
    if m != n:
        raise slsException(0)
    b = rhs.copy()
    x = np.zeros(b.shape,dtype = "float64")
    for j in reversed(range(n)):
        if U[j,j] == 0:
            raise slsException(1)
        else:
            x[j] = b[j]/U[j,j]
            for i in range(j):
                b[i] = b[i] - U[i,j]*x[j]
    return x
```

[8]:
```python
#Prueba
x = STS(U,bu)
print(x,np.allclose(U@x,bu))
```

```
[-1.  3. -1.] True
```

Sistemas lineales

Eliminación Gaussiana

Ahora se considera que A es una matriz de $n \times n$, $\boldsymbol{x}$ y $\boldsymbol{b}$ son vectores columna con n entradas, dando lugar al sistema

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

El algoritmo estándar para resolver un sistema de ecuaciones lineales se basa en la eliminación gaussiana con algunas modificaciones.

La eliminación gaussiana consiste en el proceso de reducción de filas utiliza operaciones elementales de filas y se puede dividir en dos partes. La primera parte (a veces llamada eliminación hacia adelante) reduce un sistema dado a la forma escalonada por filas, a partir de la cual se puede saber si no hay soluciones, una solución única o infinitas soluciones. La segunda parte consiste en aplicar el algoritmo de sustitución hacia atrás, ya que la primera parte transforma la matriz A en una matriz triangular superior al hacer ceros debajo de la diagonal principal.

Algo importante que hay que notar, es que la reducción de filas produce una descomposición de la matriz original. Las operaciones elementales de fila pueden verse como la multiplicación por la izquierda de la matriz original por matrices elementales. Luego, la primera parte del algoritmo calcula una descomposición de LU, con L una matriz triangular inferior y U una matriz triangular superior. Esto es, la eliminación gaussiana se puede ver como un procedimiento para factorizar una matriz A en su factorización LU, la cual se logra multiplicando por la izquierda a la matriz A por una sucesión de matrices $L_{m-1} \cdots L_2 L_1 A = U$ hasta que U sea una matriz triangular superior y L es una matriz triangular inferior, donde $L \equiv L_1^{-1} L_2^{-1} \cdots L_{m-1}^{-1}$.

Hay tres tipos de *operaciones elementales de fila* que se pueden realizar sobre las filas de una matriz:

1. Intercambiar las posiciones de dos filas.
2. Multiplicar una fila por un escalar distinto de cero.
3. Agregar a una fila un múltiplo escalar de otra.

Si la matriz está asociada a un sistema de ecuaciones lineales, estas operaciones no cambian el conjunto de soluciones. Por lo tanto, si el objetivo es resolver un sistema de ecuaciones lineales, el uso de estas operaciones de fila podría facilitar el problema.

Ejemplo Supongamos que el objetivo es encontrar y describir el conjunto de soluciones para el siguiente sistema de ecuaciones lineales:

$$\begin{aligned} 1x + 2y + 2z &= 3 \quad (\ell_1) \\ 4x + 4y + 2z &= 6 \quad (\ell_2) \\ 4x + 6y + 4z &= 10 \quad (\ell_3) \end{aligned}$$

A continuación mostraremos el proceso de reducción de filas aplicado simultáneamente al sistema de ecuaciones y su matriz aumentada asociada. En la práctica, no se suele tratar los sistemas en términos de ecuaciones ni de la matriz aumentada, sino que se manipulan directamente los arreglos donde se encuentran almacenados A y $\boldsymbol{b}$. El procedimiento de reducción de filas puede resumirse de la siguiente manera: eliminar x de todas las ecuaciones por debajo de (ℓ_1, y luego eliminar y de todas las ecuaciones por debajo de (ℓ_2). Esto pondrá el sistema en forma triangular superior. Luego, utilizando la sustitución hacia atrás, cada incógnita puede resolverse.

- Paso 0

$$\left(\begin{array}{ccc|c} 1 & 2 & 2 & 3 \\ 4 & 4 & 2 & 6 \\ 4 & 6 & 4 & 10 \end{array}\right)$$

- Paso 1
  - Hacemos $\ell_2 + \frac{-4}{1}\ell_1 \to \ell_2$
  - Hacemos $\ell_3 + \frac{-4}{1}\ell_1 \to \ell_3$

$$\left(\begin{array}{ccc|c} 1 & 2 & 2 & 3 \\ 0 & -4 & -6 & -6 \\ 0 & -2 & -4 & -2 \end{array}\right) \quad \text{entonces} \quad L_1 = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -4 & 0 & 1 \end{pmatrix}$$

- Paso 2
  - Hacemos $\ell_3 + \frac{-(-2)}{-4} \to \ell_3$

$$\left(\begin{array}{ccc|c} 1 & 2 & 2 & 3 \\ 0 & -4 & -6 & -6 \\ 0 & 0 & -1 & 1 \end{array}\right) \quad \text{entonces} \quad L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -\frac{1}{2} & 1 \end{pmatrix}$$

De aquí, aplicando sustitución hacia atrás podemos ver que $z = -1$, $y = 3$ y $x = -1$. Además se verifica que $L_2L_1A = U$ con

$$U = \begin{pmatrix} 1 & 2 & 2 \\ 0 & -4 & -6 \\ 0 & 0 & -1 \end{pmatrix}$$

Otro punto importante que se debe notar de las matrices L_i $i = 1, \ldots, n-1$ es que son matrices triangulares que tienen unos en la diagonal principal y los únicos elementos diferentes de cero son sus elementos de la i-ésima columna, los cuales son los elementos de la i-ésima columna de A por debajo de la diagonal divididos por $-a_{ii}$. Esto es, si l_{ij} denota la entrada ij de la matriz L, entonces, para L_i con i fijo, $l_{ij} = -a_{ij}/a_{ii}$ para $i > j$. Esto nos da los valores por los cuales debemos multiplicar la fila i-ésima de A para hacer ceros dedajo de la diagonal de la i-ésima columna.

El algoritmo se puede resumir a continuación:

1. **Para** $j = 1 \to n-1$ **hacer**
   - A. **Si** $a_{jj} == 0$ **entonces**
     - a. **detener**
   - B. **Si no**
     - a. **Para** $k = j+1 \to n$ **hacer**
       - I. 1. $\ell_{kj} = -a_{kj}/a_{jj}$

II. 2. **Para** $i = j+1 \to n$ **hacer**

1) 1.$a_{ki} = a_{ki} + \ell_{kj} a_{ji}$

III. 3. $b_k = b_k + \ell_{kj} b_j$

2. Resolver el sistema triangular $A\boldsymbol{x} = \boldsymbol{b}$
3. **Regresar** $\boldsymbol{x}$

[9]:
```python
# implementacion
def Gauss(M,rhs):
    A,b = M.copy(),rhs.copy()
    m,n = A.shape
    if m != n:
        raise slsException(0)
    for j in range(n-1):
        if A[j,j] == 0:
            raise slsException(2)
        for k in range(j+1,n):
            l = -A[k,j]/A[j,j]
            for i in range(j+1,n):
                A[k,i] = A[k,i] + l*A[j,i]
            b[k] = b[k] + l*b[j]
    return STS(A,b)
```

[10]:
```python
x = Gauss(A,b)
print(x,np.allclose(A@x,b))
```

```
[-1.  3. -1.] True
```

Factorización LU

Se quieren encontrar matrices triangulares L inferior y U superior tales que:

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

El primer algoritmo para encontrar la descomposición se basa en garantizar que el producto de las matrices L y U es A. Sea M una matriz, se denotará por $M_{[k]}$ a la k-esima fila de M y $M^{[k]}$ la k-ésima columna de M. Primero observe que la primera fila de A se obtiene al multiplicar la primera fila de L por U, esto es,

$$L_{[1]}U = A_{[1]}$$

pero, dado que solo $\ell_{11} = 1$ y $\ell_{1j} = 0$ para $1 < j < n$, $L_{[1]}U = U_{[1]}$, por tanto

$$U_{[1]} = A_{[1]}$$

Una vez determinada la primera fila de U, se puede proceder a determinar la primera columna de L, esto ya que

$$LU^{[1]} = A^{[1]}$$

pero, dado que solo $u_{11} \neq 0$ y $u_{j1} = 0$ para $1 < j < n$, $LU^{[1]} = u_{11}L^{[1]}$, por tanto

$$L^{[1]} = \frac{1}{u_{11}} A^{[1]}$$

En el paso dos se puede determinar de manera análoga la segunda fila de U al considerar que $L_{[2]}U = A_{[2]}$ y la segunda columna de L al considerar que $LU^{[2]} = A^{[2]}$ se tiene que

$$U_{[2]} = A_{[2]} - \ell_{21} U_{[1]} \quad \text{y} \quad L^{[2]} = \frac{1}{u_{22}} (A^{[2]} - u_{12} L^{[1]})$$

Así, en la j-ésima iteración al considerar que $L_{[j]}U = A_{[j]}$ y que $LU^{[j]} = A^{[j]}$ se tiene que

$$U_{[j]} = A_{[j]} - \sum_{i=1}^{j-1} \ell_{ji} U_{[i]} \quad \text{y} \quad L^{[j]} = \frac{1}{u_{jj}} \left(A^{[j]} - \sum_{i=1}^{j-1} u_{ij} L^{[i]} \right)$$

para $1 \leq j \leq n$.

Considerando que hay entradas cero el algoritmo queda de la siguiente manera

Método de Doolittle

1. **Para** $j = 1 \to n$ **hacer**
   - A. **Para** $k = j \to n$ **hacer**
     - a. $u_{jk} = a_{jk} - \sum_{i=1}^{j-1} \ell_{ji} u_{ik}$
   - B. $\ell_{jj} = 1$
   - C. **Para** $k = j+1 \to n$ **hacer**
     - a. $\ell_{kj} = \frac{1}{u_{jj}} \left(a_{kj} - \sum_{i=1}^{j-1} \ell_{ki} u_{ij} \right)$
2. **Regresar** L y U

[11]:
```
# Implementacion
def flu(A):
    m,n = A.shape
    if m!= n:
        raise slsException(0)
```

```
    L,U = np.eye(n),np.zeros((n,n))
    for j in range(n):
        for k in range(j,n):
            s = 0
            for i in range(j):
                s += L[j,i]*U[i,k]
            U[j,k] = A[j,k] - s
        for k in range(j+1,n):
            s = 0
            for i in range(j):
                s += L[k,i]*U[i,j]
            L[k,j] = (A[k,j] - s)/U[j,j]
    return L,U
```

[12]:
```
L,U = flu(A);
print(L@U,np.allclose(L@U,A))
```

```
[[1. 2. 2.]
 [4. 4. 2.]
 [4. 6. 4.]] True
```

[13]:
```
print(L,U,sep="\n\n")
```

```
[[1.  0.  0. ]
 [4.  1.  0. ]
 [4.  0.5 1. ]]

[[ 1.  2.  2.]
 [ 0. -4. -6.]
 [ 0.  0. -1.]]
```

En este caso sólo es necesario ver que pasa en cada iteración del algoritmo anterior y comparar con las operaciones realizadas en el método de Gauss, ya que al final de ese proceso obtenemos la matriz triangular superior U. Definamos

$$\begin{aligned} a_{ij}^{(1)} &= a_{ij} & i,j &= 1,\dots,n \\ a_{ij}^{(k+1)} &= a_{ij}^{(k)} - \ell_{ik} a_{kj}^{(k)} & k &\geq 1,\ 1 \leq i,j \leq n-k \end{aligned}$$

donde ℓ_{ik} está dada por el método de Doolittle, esto es,

$$\ell_{ik} = \frac{1}{u_{kk}} \left(a_{ik} - \sum_{j=1}^{i-1} \ell_{kj} u_{jk} \right)$$

Entonces de acuerdo con el algoritmo de la factorización LU, los elementos de la primera fila de U y de la primera columna de L están dados por

$$u_{1j} = a_{1j}^{(1)}, \quad \ell_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}}$$

Para la iteración 2, se tiene que

$$\begin{aligned} u_{2j} &= a_{2j} - \ell_{21}u_{1j} = a_{2j}^{(1)} - \ell_{21}a_{1j}^{(1)} = a_{2j}^{(2)} \\ \ell_{i2} &= \frac{a_{i2} - \ell_{i1}u_{12}}{u_{22}} = \frac{a_{i2}^{(1)} - \ell_{i1}a_{12}^{(1)}}{a_{22}^{(2)}} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}} \end{aligned}$$

Para la iteración 3, se tiene que

$$\begin{aligned} u_{3j} &= a_{3j} - \ell_{31}u_{1j} - \ell_{32}u_{2j} = a_{3j}^{(1)} - \ell_{31}a_{1j}^{(1)} - \ell_{32}a_{2j}^{(2)} = a_{3j}^{(2)} - \ell_{32}a_{2j}^{(2)} = a_{3j}^{(3)} \\ \ell_{i3} &= \frac{a_{i3} - \ell_{i1}u_{13} - \ell_{i2}u_{23}}{u_{33}} = \frac{a_{i3}^{(1)} - \ell_{i1}a_{13}^{(1)} - \ell_{i2}a_{23}^{(2)}}{a_{33}^{(3)}} = \frac{a_{i3}^{(2)} - \ell_{i2}a_{23}^{(2)}}{a_{33}^{(3)}} = \frac{a_{i3}^{(3)}}{a_{33}^{(3)}} \end{aligned}$$

Si se continua de esta manera, se tiene que

$$u_{ij} = a_{ij}^{(i)}, \quad i \leq j, \qquad \ell_{ij} = \frac{a_{ij}^{(j)}}{a_{jj}^{(j)}} \quad i > j$$

Como se mencionó al principio, la matriz U es la matriz triangular que se obtiene de aplicar el método de Gauss, mientras que L es la matriz formada por los elementos que se utilizan para hacer ceros debajo de la columna.

[14]:
```python
def lu(M):
    m,n = M.shape
    if m != n:
        raise slsException(0)
    U,L = M.copy(),np.eye(n)
    for j in range(n-1):
        if U[j,j] == 0:
            raise slsException(2)
        for k in range(j+1,n):
            L[k,j] = U[k,j]/U[j,j]
            for i in range(j+1,n):
                U[k,i] = U[k,i] - L[k,j]*U[j,i]
        U[j+1:,j] = 0
    return L,U
```

[15]:
```python
L,U = lu(A);
print(L@U,np.allclose(L@U,A))
```

```
[[1. 2. 2.]
 [4. 4. 2.]
 [4. 6. 4.]] True
```

[16]:
```python
print(L,U,sep = "\n\n")
```

```
[[1.  0.  0. ]
 [4.  1.  0. ]
```

```
 [4.  0.5 1. ]]

[[ 1  2  2]
 [ 0 -4 -6]
 [ 0  0 -1]]
```

Sistemas tridiagonales

Se quieren encontrar matrices triangulares L inferior y U superior tales que:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \ell_{21} & 1 & 0 & \cdots & 0 & 0 \\ 0 & \ell_{32} & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & \ell_{n,n-1} & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} u_{11} & u_{12} & 0 & 0 & \cdots & 0 & 0 \\ 0 & u_{22} & u_{23} & 0 & \cdots & 0 & 0 \\ 0 & 0 & u_{33} & u_{34} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & 0 & u_{nn} \end{pmatrix}}_{U} =$$

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & \cdots & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & 0 & 0 & \cdots & a_{n,n-1} & a_{nn} \end{pmatrix}}_{A}$$

Procediendo de la misma manera que antes en la factorización LU, se tiene que $L_{[1]}U = A_{[1]}$, por tanto

$$u_{11} = a_{11} \quad \text{y} \quad u_{12} = a_{12}$$

y al hacer $LU^{[1]} = A^{[1]}$ se tiene que,

$$\ell_{21} = \frac{a_{21}}{u_{11}}$$

En el paso dos se puede determinar de manera análoga la segunda fila de U al considerar que $L_{[2]}U = A_{[2]}$ y la segunda columna de L al considerar que $LU^{[2]} = A^{[2]}$ se tiene que

$$U_{[2]} = A_{[2]} - \ell_{21}U_{[1]} \quad \text{y} \quad L^{[2]} = \frac{1}{u_{22}}(A^{[2]} - u_{12}L^{[1]})$$

que al considerar las entradas diferentes de cero implica que

$$u_{22} = a_{22} - \ell_{21}u_{12}, \quad u_{23} = a_{23} \quad \text{y} \quad \ell_{32} = \frac{a_{32}}{u_{22}}$$

Así, en la j-ésima iteración al considerar que $L_{[j]}U = A_{[j]}$ y que $LU^{[j]} = A^{[j]}$ se tiene que

$$U_{[j]} = A_{[j]} - \sum_{i=1}^{j-1} \ell_{ji}U_{[i]} \quad \text{y} \quad L^{[j]} = \frac{1}{u_{jj}}\left(A^{[j]} - \sum_{i=1}^{j-1} u_{ij}L^{[i]}\right)$$

para $1 < j < n$. Restringiendose a las entradas distintas de cero se tendría que

$$u_{jj} = a_{jj} - \ell_{j,j-1}u_{j-1,j}, \quad u_{j,j+1} = a_{j,j+1} \quad \text{y} \quad \ell_{j+1,j} = \frac{a_{j+1,j}}{u_{jj}}$$

Observando que $u_{11} = a_{11}$, $u_{j,j+1} = a_{j,j+1}$ para $1 < j \leq n$ y reordenando las ecuaciones, se tiene que

$$\ell_{j,j-1} = \frac{a_{j,j-1}}{u_{j-1,j-1}} \quad \text{y} \quad u_{jj} = a_{jj} - \ell_{j,j-1}u_{j-1,j}$$

para $1 < j \leq n$.

Dado que la información sólo se encuentra en la diagonal principal y en las diagonales que están arriba y abajo, entonces para ahorrar espacio se puede guardar esta información en tres vectores. Sean $\boldsymbol{d} \in \mathbb{R}^n$ y $\boldsymbol{e}, \boldsymbol{f} \in \mathbb{R}^{n-1}$ tales que

$$d_j = a_{jj} \quad 1 \leq j \leq n,$$

$$e_{j-1} = a_{j,j-1} \quad 1 < j \leq n$$

y

$$f_j = a_{j,j+1} \quad 1 \leq j < n$$

de esta manera la diagonal debajo de la diagonal principal de L y la diagonal principal de U se pueden guardar en un vector $\boldsymbol{l} \in \mathbb{R}^{n-1}$ y $\boldsymbol{u} \in \mathbb{R}^n$, respectivamente, donde sus entradas se pueden calcular como:

$$l_{j-1} = \frac{e_{j-1}}{u_{j-1}} \quad \text{y} \quad u_j = d_j - l_{j-1}f_{j-1}$$

para $1 < j \leq n$, y tomando $u_1 = d_1$.

Considerando estás últimas ecuaciones el algoritmo queda de la siguiente manera:

1. **Para** $j = 1 \to n$ **hacer**
    1. $l_{j-1} = \frac{e_{j-1}}{u_{j-1}}$
    2. $u_j = d_j - l_{j-1}f_{j-1}$
2. **Regresar** $\boldsymbol{l}$ y $\boldsymbol{u}$

[17]:
```python
# Implementacion
def triDiaglu(d,e,f):
    n = d.size
    l = np.zeros((n-1,),dtype = "float64")
    u = np.array(d,copy = True,dtype = "float64")
    for j in range(1,n):
        l[j-1] = e[j-1]/u[j-1]
        u[j] = d[j] - l[j-1]*f[j-1]
    return l,u
```

[18]:
```python
# Ejemplo
n = 4
d = 2*np.ones((n,),dtype="float64")
e = -1*np.ones((n-1,),dtype="float64")
f = e.copy()
```

```
M = np.diagflat(d) + np.diagflat(e,-1) + np.diagflat(e,1)
b = np.zeros((n,),dtype="float64")
b[0] = b[-1] = 1.
print(M,b,sep="\n\n")
```

```
[[ 2. -1.  0.  0.]
 [-1.  2. -1.  0.]
 [ 0. -1.  2. -1.]
 [ 0.  0. -1.  2.]]

[1. 0. 0. 1.]
```

[19]:
```
l,u = triDiaglu(d,e,f)
L = np.eye(n) + np.diagflat(l,-1)
U = np.diagflat(u) + np.diagflat(f,1)
print(L,U,L@U,np.allclose(L@U,M),sep="\n\n")
```

```
[[ 1.          0.          0.          0.        ]
 [-0.5         1.          0.          0.        ]
 [ 0.         -0.66666667  1.          0.        ]
 [ 0.          0.         -0.75        1.        ]]

[[ 2.         -1.          0.          0.        ]
 [ 0.          1.5        -1.          0.        ]
 [ 0.          0.          1.33333333 -1.        ]
 [ 0.          0.          0.          1.25      ]]

[[ 2. -1.  0.  0.]
 [-1.  2. -1.  0.]
 [ 0. -1.  2. -1.]
 [ 0.  0. -1.  2.]]

True
```

Dado que tenemos la factorización LU de la matriz tridiagonal A, tenemos que resolver los sistemas

$$L\boldsymbol{y} = \boldsymbol{b} \quad \text{y} \quad U\boldsymbol{x} = \boldsymbol{y}$$

para tener la solución del sistema $A\boldsymbol{x} = \boldsymbol{b}$. Dadas las caracteristicas de las matrices L y U, se tiene que, usando sustitución hacia adelante

$$y_1 = b_1 \quad \text{y} \quad y_j = b_j - l_{j-1}y_{j-1} \quad \text{para } 1 < j \leq n$$

y posteriormente, usando sustitución hacia atrás

$$x_n = \frac{y_n}{u_n} \quad \text{y} \quad x_j = \frac{y_j - f_j x_{j+1}}{u_j} \quad \text{para } 1 \leq j < n$$

[20]:
```
def triDiagluSol(d,e,f,b):
    n = d.size
    l = np.zeros((n-1,),dtype = "float64")
    u = np.array(d,copy = True,dtype = "float64")
    for j in range(1,n):
        l[j-1] = e[j-1]/u[j-1]
        u[j] = d[j] - l[j-1]*f[j-1]
    l = np.append(b[0],l)
    for j in range(1,n):
        l[j] = b[j] - l[j-1]*l[j]
    u[-1] = l[-1]/u[-1]
    for j in reversed(range(n-1)):
        u[j] = (l[j] - f[j]*u[j+1])/u[j]
    return u
```

[21]:
```
x = triDiagluSol(d,e,f,b)
print(x,np.allclose(M@x,b), sep = "\n\n")
```

```
[1. 1. 1. 1.]

True
```