

1.2 Algebra lineal y sistemas de ecuaciones lineales

Contents

- [Introducción a los sistemas de ecuaciones lineales](#)
- [Métodos de solución directos](#)
- [Métodos iterativos](#)
- [Solución de sistemas de ecuaciones lineales con python](#)
- [Referencias](#)

MEC301 - Metodos Numéricos

Profesor: Francisco Ramírez Cuevas

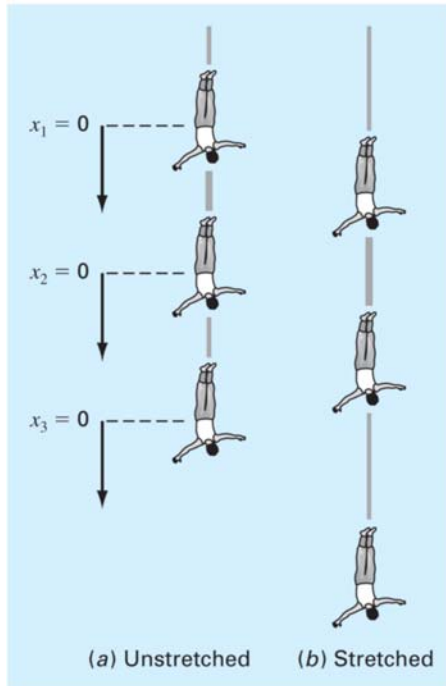
Fecha: 8 de Agosto 2022

Contenidos

- [Introducción a los sistemas de ecuaciones lineales](#)
 - [Definición general](#)
 - [Representación matricial](#)
 - [Repaso de matrices](#)
 - [Representación en python](#)
 - [Caracterización de sistemas de ecuaciones lineales](#)
- [Métodos de solución directos](#)
 - [Eliminación de Gauss](#)
 - [Factorización LU](#)
- [Métodos iterativos](#)
 - [Gauss-Seidel](#)
- [Solución de sistemas de ecuaciones lineales con python](#)
- [Referencias](#)

Introducción a los sistemas de ecuaciones lineales

Consideremos el caso de tres personas conectada por cuerdas elásticas.

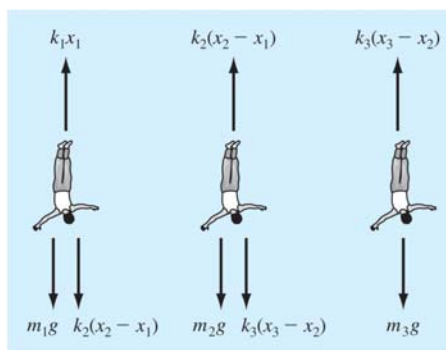


En la primera figura (a), los tres cuerpos están en la posición inicial de forma que los elásticos están totalmente extendidos, **pero no estirados**. Definimos el cambio en la posición inicial de cada persona, como: x_1 , x_2 , x_3 .

Cuando los cuerpos se dejan caer, los elásticos se extienden por la gravedad y cada cuerpo toma la posición indicada en (b).

Analizamos el cambio en la posición de cada persona utilizando la ley de Newton:

Diagrama de cuerpo libre



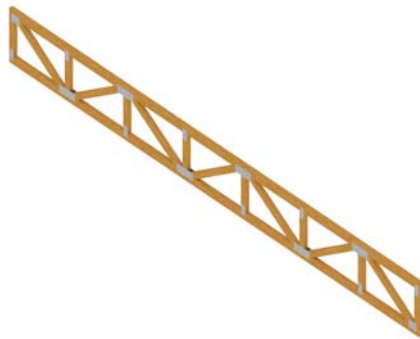
$$\begin{aligned}
m_1 \frac{d^2 x_1}{dt^2} &= m_1 g + k_2(x_2 - x_1) - k_1 x_1 \\
m_2 \frac{d^2 x_2}{dt^2} &= m_2 g + k_3(x_3 - x_2) - k_2(x_1 - x_2) \\
m_3 \frac{d^2 x_3}{dt^2} &= m_3 g + k_3(x_2 - x_3)
\end{aligned}$$

En condiciones de equilibrio:

$$\begin{aligned}
(k_1 + k_2)x_1 - k_2 x_2 &= m_1 g \\
-k_2 x_1 + (k_2 + k_3)x_2 - k_3 x_3 &= m_2 g \\
-k_3 x_2 + k_3 x_3 &= m_3 g
\end{aligned}$$

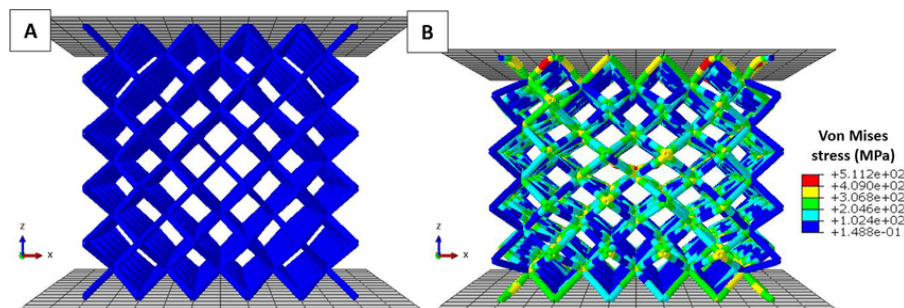
En el ejemplo anterior, derivamos un sistema de ecuaciones lineales con 3 incógnitas el cual podemos resolver con técnicas analíticas.

Sin embargo, si el sistema es más grande, como por ejemplo un reticulado de vigas:



Tenemos un sistema de ecuaciones con un gran número de incógnitas y debemos recurrir a métodos más eficientes para poder resolverlos.

Este es el enfoque que utilizan los software de modelación computacional, tales como: el método de elementos finitos (FEM), métodos de los momentos (MoM), o volúmenes finitos (VEM).



Definición general

Decimos que una ecuación es lineal cuando:

1. Todas sus incógnitas están **únicamente** separadas por sumas o restas
2. El exponente de cada incógnita es 1.

Por ejemplo,

- $3x_1 + 4x_2 - 3 = -5x_3$ (lineal)
- $\frac{-3x_1 + x_2}{x_3} = 2$ (no es lineal, pero podemos transformarla en una ecuación lineal: $-3x_1 + x_2 - 2x_3 = 0$)
- $x_1x_2 + x_3 = 5$ (no lineal)
- $x_1 + 3x_2 + x_3^4 = 3$ (no lineal)

Un sistema de ecuaciones lineales está compuesto por más de una ecuación lineal, tal como en el ejemplo de las personas conectadas por cuerdas elásticas

$$\begin{aligned}(k_1 + k_2)x_1 - k_2x_2 &= m_1g \\ -k_2x_1 + (k_2 + k_3)x_2 - k_3x_3 &= m_2g \\ -k_3x_2 + k_3x_3 &= m_3g\end{aligned}$$

Representación matricial

Para resolver sistemas de ecuaciones lineales se utiliza la representación matricial. Esto permite la implementación computacional de los algoritmos.

Por ejemplo, consideremos una ecuación lineal en su forma general:

$$\begin{aligned}a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,n-1}x_{n-1} + a_{1,n}x_n &= y_1, \\ a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,n-1}x_{n-1} + a_{2,n}x_n &= y_2, \\ &\dots \\ a_{m-1,1}x_1 + a_{m-1,2}x_2 + \dots + a_{m-1,n-1}x_{n-1} + a_{m-1,n}x_n &= y_{m-1}, \\ a_{m,1}x_1 + a_{m,2}x_2 + \dots + a_{m,n-1}x_{n-1} + a_{m,n}x_n &= y_m.\end{aligned}$$

donde $a_{i,j}$ y y_i son números reales.

La forma matricial de esta ecuación tiene la siguiente forma:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{bmatrix}$$

O, similarmente,

$$Ax = y,$$

donde:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix}, \quad y = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_m \end{bmatrix}$$

De igual forma, el problema de las personas sujetas con elásticos,

$$\begin{aligned} (k_1 + k_2)x_1 - k_2x_2 &= m_1g \\ -k_2x_1 + (k_2 + k_3)x_2 - k_3x_3 &= m_2g \\ -k_3x_2 + k_3x_3 &= m_3g, \end{aligned}$$

se puede representar de forma matricial como:

$$\begin{bmatrix} k_1 + k_2 & -k_2 & 0 \\ -k_2 & k_2 + k_3 & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} m_1g \\ m_2g \\ m_3g \end{bmatrix}$$

Repaso de matrices

1. **Norma matricial.** Existen distintos tipos. La más conocida es la *p-norma*:

$$\|M\|_p = \sqrt[p]{\left(\sum_i^m \sum_j^n |a_{ij}|^p\right)}$$

Para $p = 2$, se llama *norma de Frobenius*

1. **Determinante.** Se denota como $\det(M)$, o $|M|$. **Solo se aplica a matrices cuadradas.**

Por ejemplo, para una matriz 2×2 , el determinante es:

$$|M| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc,$$

para una matrix 3×3 :

$$\begin{aligned} |M| &= \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix} - b \begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix} + c \begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix} \\ &= a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= aei - afh + bfg - bdi + cdh - ceg \end{aligned}$$

1. **Matriz identidad (I).** es una matriz cuadarada con 1 en la diagonal, y 0 en el resto de los elementos:

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

1. **Matriz inversa.** Definimos la matriz inversa de M como: M^{-1} .

- Solo existe para matrices cuadradas.
- El producto de la matriz inversa por su diagonal es igual a la matriz identidad $M \cdot M^{-1} = I$
- Para una matriz 2×2 , la matriz inversa está definida por:

$$M^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{|M|} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

La solución analítica para determinar la matriz inversa se vuelve mas complicada a medida que aumentan las dimensiones de la matriz.

Representación en python

Para representar sistemas de ecuaciones lineales en python utilizamos variables del tipo *numpy array* de la libreria *numpy*.

Por ejemplo, para representar el sistema:

$$\begin{aligned} 3x_1 + 1x_2 - 5x_3 &= 2 \\ -2x_1 - 2x_2 + 5x_3 &= 5 \\ 8x_1 + 3x_2 &= -3 \end{aligned}$$

```
import numpy as np

A = np.array([[ 3,  1, -5],
              [-2, -2,  5],
              [ 8,  3,  3]])

y = np.array([2], [5], [-3])

print('A:\n',A)
print('\ny:\n',y)
```

```
A:
[[ 3  1 -5]
 [-2 -2  5]
 [ 8  3  3]]

y:
[[ 2]
 [ 5]
 [-3]]
```

La librería *linalg* de *numpy* tiene funciones predefinidas para calcular la norma, determinante y matriz inversa.

```
from numpy.linalg import norm, det, inv

print('norm(A) = %.4f (Frobenius por defecto)'% norm(A))
print('det(A) = %.4f' % det(A))
print('inv(A):\n',inv(A))
```

```
norm(A) = 12.2474 (Frobenius por defecto)
det(A) = -67.0000
inv(A):
[[ 0.31343284  0.26865672  0.07462687]
 [-0.68656716 -0.73134328  0.07462687]
 [-0.14925373  0.01492537  0.05970149]]
```

```
# comprobamos la identidad A*A^-1 = I
# usamos numpy.dot() para multiplicar matrices
A.dot(inv(A))
```

```
array([[ 1.00000000e+00, -5.20417043e-18,  3.46944695e-17],
       [ 0.00000000e+00,  1.00000000e+00,  2.08166817e-17],
       [ 2.22044605e-16,  9.19403442e-17,  1.00000000e+00]])
```

Para la matriz identidad utilizamos la función `eye` de la librería `numpy`.

```
np.eye(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Caracterización de sistemas de ecuaciones lineales

Un sistema de ecuaciones lineales tiene solución única, si y solo si el número de incógnitas es igual al número de **ecuaciones linealmente independientes** en el sistema

Por ejemplo, el siguiente sistema de ecuaciones lineales:

$$\begin{aligned} 3x_1 + 1x_2 - 5x_3 &= 2 \\ -2x_1 - 2x_2 + 5x_3 &= 5 \\ 4x_1 - 5x_3 &= 9 \end{aligned}$$

No tiene solución única, ya que: (ec. 3) = 2 × (ec. 1) + (ec. 2)

Definimos el rango de la matriz $\text{rank}(A)$, como el número de filas (o columnas) linealmente independientes.

En python, $\text{rank}(A)$ está dado por la función `matrix_rank` de la librería `numpy.linalg`

```
from numpy.linalg import matrix_rank
A = np.array([[ 3,  1, -5],
              [-2, -2,  5],
              [ 4,  0, -5]])
matrix_rank(A)
```

2

Consideremos la matrix aumentada $[A|y]$ como:

$$[A|y] = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} & y_1 \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} & y_2 \\ \dots & \dots & \dots & \dots & \dots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} & y_n \end{bmatrix}$$

donde A es una matriz $m \times n$. Es decir, m ecuaciones y n incógnitas.

- **El sistema tiene solución única** si $\text{rank}([A|y]) = \text{rank}(A)$, y $\text{rank}(A) = n$
- **El sistema tiene infinitas soluciones** si $\text{rank}([A|y]) = \text{rank}(A)$, y $\text{rank}(A) < n$
- **El no tiene soluciones** si $\text{rank}([A|y]) = \text{rank}(A) + 1$

En el caso del ejemplo anterior:

$$\begin{aligned} 3x_1 + 1x_2 - 5x_3 &= 2 \\ -2x_1 - 2x_2 + 5x_3 &= 5 \\ 4x_1 - 5x_3 &= 9 \end{aligned}$$

```
y = np.array([[2], [5], [9]])
Ay_aug = np.concatenate((A,y),axis = 1)
print('[A|y] =\n', Ay_aug)
print('\n')
print('rank(A|b) =', matrix_rank(Ay_aug))
print('rank(A) =', matrix_rank(A))
print('Número de incógnitas, n =', A.shape[1])
```

```
[A|y] =
[[ 3  1 -5  2]
 [-2 -2  5  5]
 [ 4  0 -5  9]]
```

```
rank(A|b) = 2
rank(A) = 2
Número de incógnitas, n = 3
```

$$\begin{aligned} \text{rank}([A|y]) &= \text{rank}(A) \\ \text{rank}(A) &< n \end{aligned}$$

El sistema tiene múltiples soluciones

Si $\det(A) = 0$, decimos que **la matriz es singular** y, por lo tanto, no es invertible.

Por ejemplo, la matriz:

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

es singular.

```
P = np.array([[ 1, 2, -1],
               [ 2, 3,  0],
               [ 1, 1,  1]])
print('det(P) = ', det(P))
```

```
det(P) = 0.0
```

y, por lo tanto, no es invertible:

```
print('inv(P) = ', inv(P))
```

```
-----
-----
LinAlgError                                Traceback (most recent
call last)
/tmp/ipykernel_487/453559162.py in <module>
----> 1 print('inv(P) = ', inv(P))

~/programs/miniconda3/lib/python3.9/site-
packages/numpy/core/overrides.py in inv(*args, **kwargs)

~/programs/miniconda3/lib/python3.9/site-
packages/numpy/linalg/linalg.py in inv(a)
    550     signature = 'D->D' if isComplexType(t) else 'd->d'
    551     extobj =
get_linalg_error_extobj(_raise_linalgerror_singular)
--> 552     ainv = _umath_linalg.inv(a, signature=signature,
extobj=extobj)
    553     return wrap(ainv.astype(result_t, copy=False))
    554

~/programs/miniconda3/lib/python3.9/site-
packages/numpy/linalg/linalg.py in _raise_linalgerror_singular(err,
flag)
    87
    88 def _raise_linalgerror_singular(err, flag):
--> 89     raise LinAlgError("Singular matrix")
    90
    91 def _raise_linalgerror_nonposdef(err, flag):

LinAlgError: Singular matrix
```

Decimos que una matriz A está **mal condicionada**, si $\det(A) \approx 0$.

Si bien las matrices mal condicionadas tienen inversa, son numericamente problemáticas, ya que pueden inducir errores de redondeo, *overflow* o *underflow* como resultado de la división por un número muy pequeño

Para determinar si una matriz está mal condicionada utilizamos el **número de condición**, definido como:

$$\text{Cond}(A) = \|A\| \cdot \|A^{-1}\|$$

Matrices mal condicionadas están caracterizadas por "Cond(A)" altos

En python, $\text{Cond}(A)$ está dado por la función `cond` de la librería `numpy.linalg`

```
from numpy.linalg import cond
print('Cond(P) = ', cond(P))
```

```
Cond(P) = 3.757434988222266e+16
```

$\det(A) = 0$, no necesariamente significa que el sistema no tiene solución

Por ejemplo, en el ejemplo anterior

$$\begin{aligned} 3x_1 + 1x_2 - 5x_3 &= 2 \\ -2x_1 - 2x_2 + 5x_3 &= 5 \\ 4x_1 - 5x_3 &= 9 \end{aligned}$$

```
print('A\n', A)
print('\n')
print('det(A) = ', det(A))
```

```
A
[[ 3  1 -5]
 [-2 -2  5]
 [ 4  0 -5]]
```

```
det(A) = 0.0
```

Sin embargo, como habíamos determinado, el sistema tiene múltiples soluciones.

Métodos de solución directos

Eliminación de Gauss

Es un algoritmo para resolver sistemas ecuaciones lineales basado en convertir la matriz A en una matriz **triangular superior**. El sistema toma la forma:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ 0 & a'_{2,2} & a'_{2,3} & a'_{2,4} \\ 0 & 0 & a'_{3,3} & a'_{3,4} \\ 0 & 0 & 0 & a'_{4,4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} y_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{bmatrix}$$

Esta ecuación puede resolverse fácilmente, comenzando por $x_4 = y'_4/a'_{4,4}$, luego continuamos con $x_3 = \frac{y'_3 - a_{3,4}x_4}{a_{3,4}}$, y así sucesivamente hasta llegar a x_1 . En otras palabras, utilizamos **sustitución hacia atrás**, resolviendo el sistema desde abajo hacia arriba.

Si A es una matriz **triangular inferior**, resolveríamos el problema de arriba hacia abajo utilizando **sustitución hacia adelante**.

La mejor forma de entender el método de eliminación Gauseana es con un ejemplo:

$$\begin{aligned} 4x_1 + 3x_2 - 5x_3 &= 2 \\ -2x_1 - 4x_2 + 5x_3 &= 5 \\ 8x_1 + 8x_2 &= -3 \end{aligned}$$

Paso 1: Transformamos el sistema de ecuaciones en su forma matricial $Ax = y$.

$$\begin{bmatrix} 4 & 3 & -5 \\ -2 & -4 & 5 \\ 8 & 8 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2 \\ 5 \\ -3 \end{bmatrix}$$

Paso 2: Determinar la matriz aumentada $[A, y]$

$$[A, y] = \begin{bmatrix} 4 & 3 & -5 & 2 \\ -2 & -4 & 5 & 5 \\ 8 & 8 & 0 & -3 \end{bmatrix}$$

Paso 3: Determinamos la matriz triangular superior utilizando pivoteo parcial y eliminación.

- Comenzando por la primera columna. Primero, permutamos las filas de manera que el coeficiente con mayor valor absoluto quede en la primera fila:

$$[A, y] = \begin{bmatrix} 8 & 8 & 0 & -3 \\ 4 & 3 & -5 & 2 \\ -2 & -4 & 5 & 5 \end{bmatrix}$$

- Luego, eliminamos los otros coeficientes de la primera columna, comenzando por el segundo. Multiplicamos la primera fila por $1/2$ y la restamos a la segunda fila:

$$[A, y] = \begin{bmatrix} 8 & 8 & 0 & -3 \\ 0 & -1 & -5 & 3.5 \\ -2 & -4 & 5 & 5 \end{bmatrix}$$

- Después, multiplicamos la primera fila por $-1/4$ y la restamos a la tercera fila:

$$[A, y] = \begin{bmatrix} 8 & 8 & 0 & -3 \\ 0 & -1 & -5 & 3.5 \\ 0 & -2 & 5 & 4.25 \end{bmatrix}$$

- Repetimos el proceso con la segunda columna. Primero, permutamos las filas:

$$[A, y] = \begin{bmatrix} 8 & 8 & 0 & -3 \\ 0 & -2 & 5 & 4.25 \\ 0 & -1 & -5 & 3.5 \end{bmatrix}$$

- Luego, eliminamos el coeficiente inferior. Multiplicamos por la segunda fila por $1/2$ y restamos a la tercera fila:

$$[A, y] = \begin{bmatrix} 8 & 8 & 0 & -3 \\ 0 & -2 & 5 & 4.25 \\ 0 & 0 & -7.5 & 1.375 \end{bmatrix}$$

Paso 4. Realizamos sustitución hacia atrás.

$$\begin{aligned} x_3 &= \frac{-1.375}{7.5} = -0.183 \\ x_2 &= \frac{4.25 - (-2)x_3}{5} = -2.583 \\ x_1 &= \frac{-3 - 8x_2 + 0x_3}{8} = 2.208 \end{aligned}$$

El método de eliminación Gaussiana es de complejidad $O(N^3)$

Factorización LU

Es posible demostrar que cualquier matriz cuadrada A puede ser expresada como el producto de una matriz triangular inferior L , y una matriz triangular superior U .

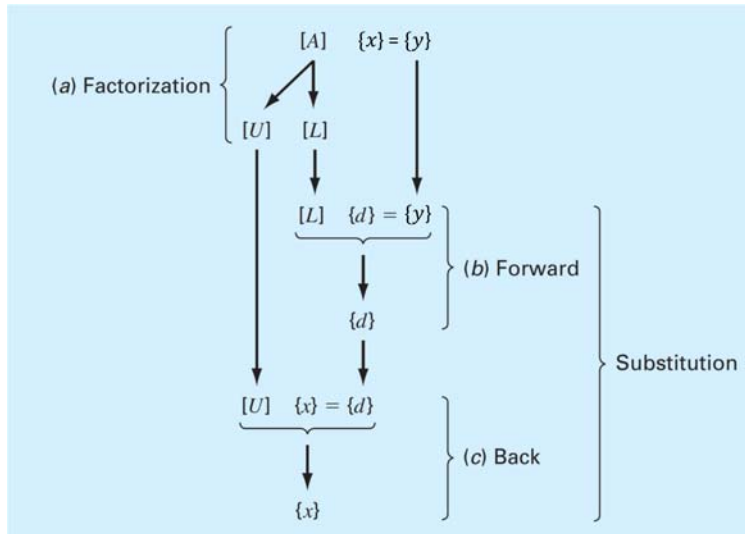
$$A = LU$$

El proceso para obtener L y U es conocido como *descomposición* o *factorización* LU. **Es el método de solución de ecuaciones lineales más confiable y utilizado.**

El tipo de factorización LU no es única, ya que existen múltiples formas de representar L y U para un A dado. Así, definimos tres tipos de factorizaciones comúnmente utilizadas:

Nombre	Condiciones
Doolittle	$L_{ii} = 1, i = 1, 2, \dots, n$
Crout	$U_{ii} = 1, i = 1, 2, \dots, n$
Choleski	$L = U^T$

Una vez ejecutada la factorización, resolvemos el sistema $Ax = y$.



- Primero resolvemos el sistema $Ld = y$, por sustitución hacia adelante.
- Luego, resolvemos el sistema $Ux = d$, por sustitución hacia atrás.

A diferencia del método de eliminación de Gauss, la factorización LU no depende del vector y . Por lo tanto, es conveniente para resolver el sistema $Ax = y$, con múltiples valores de y .

Debido a que la factorización LU está basada en eliminación de Gauss, el orden de complejidad es $O(N^3)$.

Existen diversos métodos para obtener las matrices L y U . Uno de ellos es mediante eliminación Gaussiana.

Como mostramos anteriormente, el método de eliminación de Gauss permite determinar una matriz triangular superior. La matriz triangular inferior, aunque no se mostró de forma explícita esta conformada por "1" en la diagonal, y los múltiplos utilizados para eliminar los elementos de las columnas.

En general, se puede demostrar que para una matriz A , se cumple la siguiente relación:

$$PA = LU$$

donde P es la matriz de permutaciones.

Por ejemplo, en el ejercicio anterior:

$$\begin{aligned} 4x_1 + 3x_2 - 5x_3 &= 2 \\ -2x_1 - 4x_2 + 5x_3 &= 5 \\ 8x_1 + 8x_2 &= -3 \end{aligned}$$

Tenemos: $L = \begin{bmatrix} 1 & 0 & 0 \\ -0.25 & 1 & 0 \\ 0.5 & 0.5 & 1 \end{bmatrix}; U = \begin{bmatrix} 8 & 8 & 0 \\ 0 & -2 & 5 \\ 0 & 0 & -7.5 \end{bmatrix}; P = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$

```

A = np.array([[ 4,  3, -5],
               [-2, -4,  5],
               [ 8,  8,  0]])

L = np.array([[ 1,  0,  0],
               [-0.25,  1,  0],
               [ 0.5,  0.5,  1]])

U = np.array([[ 8,  8,  0],
               [ 0, -2,  5],
               [ 0,  0, -7.5]])

P = np.array([[0, 0, 1],
               [0, 1, 0],
               [1, 0, 0]])

```

```

print('P*A =\n', np.dot(P,A))
print('\n')
print('L*U =\n', np.dot(L,U))

```

```

P*A =
[[ 8  8  0]
 [-2 -4  5]
 [ 4  3 -5]]

L*U =
[[ 8.  8.  0.]
 [-2. -4.  5.]
 [ 4.  3. -5.]]

```

Métodos iterativos

Los métodos iterativos están basados en una serie repetitiva de operaciones, comenzando por un valor inicial. A diferencia de los métodos directos, el número de operaciones está condicionado por la convergencia y el valor inicial escogido.

Las ventajas de los métodos iterativos es que tienen un orden de complejidad menor que los métodos directos, y no requieren gran capacidad de memoria (recordemos que factorización LU requiere almacenar las matrices L, U y P)

La gran desventaja radica en la convergencia de los algoritmos. Una condición suficiente, pero no necesaria es que la matriz A debe ser **diagonal dominante**, es decir, los elementos de la diagonal, $a_{i,i}$, deben satisfacer: $|a_{i,i}| \geq \sum_{j \neq i} |a_{i,j}|$

Estos métodos se utilizan, generalmente, en simulaciones con elementos finitos (FEM), o volúmenes finitos (VEM).

Gauss-Seidel

El algoritmo se puede resumir en los siguientes pasos:

1. Asumimos un valor inicial para $x_2^{(0)}, x_3^{(0)}, \dots, x_n^{(0)}$ (con excepción de $x_1^{(0)}$).
 2. Calculamos un nuevo valor para $x_1^{(1)}$ mediante: $x_1^{(1)} = \frac{1}{a_{1,1}} \left[y_1 - \sum_{j \neq 1}^n a_{1,j} x_j^{(0)} \right]$
 3. Utilizando el nuevo valor $x_1^{(1)}$ y el resto de $x^{(0)}$ (con excepción de $x_2^{(0)}$), determinamos $x_2^{(1)}$. $x_2^{(1)} = \frac{1}{a_{2,2}} \left[y_2 - \sum_{j \neq 1,2}^n a_{2,j} x_j^{(0)} - a_{2,1} x_1^{(1)} \right]$
1. Repetimos el paso 3 hasta completar todos los elementos del vector x .
 2. Continuamos con la iteración hasta que el valor de x converge dentro de una tolerancia ε , definida por:

$$\|x^{(i)} - x^{(i-1)}\| < \varepsilon$$

Por ejemplo, resolvamos el siguiente sistema de ecuaciones con el método de Gauss-Seidel:

$$\begin{aligned} 8x_1 + 3x_2 - 3x_3 &= 14 \\ -2x_1 - 8x_2 + 5x_3 &= 5 \\ 3x_1 + 5x_2 + 10x_3 &= -8 \end{aligned}$$

Primero, verificamos que la matriz es diagonal dominante:

```
A = [[ 8.,  3., -3.],
      [-2., -8.,  5.],
      [3.,  5., 10.]]

# coeficientes de la diagonal
diagA = np.diag(np.abs(A))
print('diag(A) = ',diagA)

# suma de los elementos sin la diagonal
off_diagA = np.sum(np.abs(A), axis=1) - diagA
print('off_diag(A) = ',off_diagA)
```

```
diag(A) = [ 8.  8. 10.]
off_diag(A) = [6.  7.  8.]
```

```
if np.all(diagA > off_diagA):
    print('la matriz es diagonal dominante')
else:
    print('la matriz no es diagonal dominante')
```

```
la matriz es diagonal dominante
```

```
def gauss_seidel(A,y,x):

    epsilon = 0.001 # tolerancia
    converged = False # verificar convergencia

    # guardamos el valor inicial
    x_old = np.array(x)

    for k in range(1, 50):
        for i in range(x.size):
            x[i] = y[i] # xi = yi
            for j in range(x.size):
                if i == j: # saltamos i = j
                    continue
                x[i] -= A[i][j]*x[j] # xi = yi - sum(aij*xj)
            x[i] = x[i]/A[i][i] # xi = (yi - sum(aij*xj))/aii

        # comparamos el error con la tolerancia
        dx = norm(x-x_old)

        print("iter =",k,'; x =',x)
        if dx < epsilon:
            converged = True
            print('Converged!')
            break

        # guardamos el valor de x para la nueva iteracion
        x_old = np.array(x)

    if not converged:
        print('No converge, incrementar número de iteraciones')
    return x
```

```
A = [[ 8., 3., -3.],
      [-2., -8., 5.],
      [3., 5., 10.]]

y = np.array([14., 5., -8.])
x = np.array([0.,0.,0.]) # valores iniciales
gauss_seidel(A,y,x)
```

```
iter = 1 ; x = [ 1.75 -1.0625 -0.79375]
iter = 2 ; x = [ 1.85078125 -1.58378906 -0.56333984]
iter = 3 ; x = [ 2.13266846 -1.51025452 -0.68467328]
iter = 4 ; x = [ 2.05959296 -1.56781904 -0.63396837]
iter = 5 ; x = [ 2.100194 -1.54627873 -0.65691883]
iter = 6 ; x = [ 2.08350996 -1.55645176 -0.64682711]
iter = 7 ; x = [ 2.09110925 -1.55204425 -0.65131065]
iter = 8 ; x = [ 2.0877751 -1.55401293 -0.64932607]
iter = 9 ; x = [ 2.08925757 -1.55314318 -0.65020568]
iter = 10 ; x = [ 2.08860156 -1.55352894 -0.649816 ]
Converged!
```

```
array([ 2.08860156, -1.55352894, -0.649816 ])
```

Solución de sistemas de ecuaciones lineales con python

En python la forma más fácil de resolver sistemas de ecuaciones lineales es mediante la función `solve` de `numpy.linalg`. Esta función utiliza factorización LU para resolver el sistema.

Por ejemplo, tomemos el ejemplo de las personas conectadas por elásticos:

Persona	Masa (kg)	Constante del resorte (N/m)	Longitud inicial del elástico
primera	60	50	20
segunda	70	100	20
tercera	80	50	20

Tenemos un sistema de la forma:

$$\begin{bmatrix} 150 & -100 & 0 \\ -100 & 150 & -50 \\ 0 & -50 & 50 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 588.6 \\ 686.7 \\ 784.8 \end{bmatrix}$$

```
import numpy as np

A = np.array([[ 150, -100,  0],
              [-100,  150, -50],
              [  0,  -50,  50]])

y = np.array([588.6, 686.7, 784.8])

x = np.linalg.solve(A, y)
print(x)
```

```
[41.202 55.917 71.613]
```

Notar que en este problema x_1 , x_2 y x_3 representan las posiciones relativas de las personas. Así la posición final está dada por:

```
print('Posición final de las personas: ', x + [20, 40, 60])
```

```
Posición final de las personas: [ 61.202  95.917 131.613]
```

Mediante la librería `scipy` podemos hacer factorización LU.

```
from scipy.linalg import lu
```

```
P, L, U = lu(A)
print('P:\n', P)
print('L:\n', L)
print('U:\n', U)
print('LU:\n', np.dot(L, U))
```

```
P:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
L:
[[ 1.          0.          0.          ]
 [-0.66666667  1.          0.          ]
 [ 0.          -0.6       1.          ]]
U:
[[ 150.          -100.          0.          ]
 [  0.           83.33333333 -50.          ]
 [  0.           0.           20.          ]]
LU:
[[ 150. -100.   0.]
 [-100.  150. -50.]
 [  0.  -50.  50.]]
```

Referencias

- Kong Q., Siau T., Bayen A. M. **Chapter 14: Linear Algebra and Systems of Linear Equations** in [Python Programming and Numerical Methods – A Guide for Engineers and Scientists](#), 1st Ed., Academic Press, 2021
- Chapra S., Canale R. **Parte tres: Ecuaciones algebraicas lineales** en *Métodos Numéricos para Ingenieros*, 6ta Ed., McGraw Hill, 2011