MEC501 - Manejo y conversión de energía solar térmica

Francisco V. Ramirez-Cuevas

Aug 08, 2022

CONTENTS

This is a small sample book to give you a feel for how book content is structured. It shows off a few of the major file types, as well as some sample content. It does not go in-depth into any particular topic - check out the Jupyter Book documentation for more information.

Check out the content pages bundled with this sample book to see more.

- *Unidad 1.1 Aspectos generales de programación y algoritmos*
- *1.2 Algebra lineal y sistemas de ecuaciones lineales*

MEC301 - Metodos Numéricos

CHAPTER

ONE

UNIDAD 1.1 ASPECTOS GENERALES DE PROGRAMACIÓN Y ALGORITMOS

Profesor: Francisco Ramírez Cuevas Fecha: 1 de Agosto 2022

1.1 Complejidad de algoritmos

1.1.1 ¿Qué es un algoritmo?

Un algoritmo es una serie ordenada de operaciones sistemáticas que permite hacer un cálculo y hallar la solución de un tipo de problemas.

Por ejemplo:

```
def f(n):
    out = 0
    for i in range(n):
        for j in range(n):
            out += i*j
    return out
```

La complejidad de un algoritmo es la **relación entre el tamaño del input** N y la **cantidad de operaciones para completarlo**. Una forma de determinar la complejidad del algoritmo es **contabilizar las operaciones básicas**:

- sumas
- restas
- multiplicaciones
- divisiones
- asignación de variables
- llamados a otras funciones

Por ejemplo, en el siguiente algoritmo:

```
def f(n):
    out = 0
    for i in range(n):
        for j in range(n):
```

(continues on next page)

(continued from previous page)

```
            out += i*j
    return out
```

El número de operaciones son:

- sumas: N^2
- restas: 0
- multiplicaciones: N^2
- divisiones: 0
- asignación de variables: $2N^2 + N + 1$
- llamados a otras funciones: 0

Así, el **total de operaciónes** para completar el algoritmo es $4N^2 + N + 1$.

1.1.2 Notación *Big-O*

A medida que el tamaño de N aumenta, las operaciones de mayor orden se hacen dominantes. Así, podemos decir que la complejidad del algoritmo anterior es del orden $O(N^2)$. Esta notación, denominada ***Big-O***, es comúnmente utiilzada para **determinar la complejidad del algoritmo cuando N es de gran tamaño**.

> **Nota** Un algoritmo tiene complejidad **polynomial** cuando es del tipo $O(N^c)$, donde c es una constante.

Analicemos la complejidad del siguiente algortimo:

```
def my_divide_by_two(n):

    out = 0
    while n > 1:
        n /= 2
        out += 1

    return out
```

A medida que N crece podemos ver que la parte dominante de este algoritmo esta dentro de la operación `while`.

Si analizamos el número de iteraciones I para un determinado N, notaremos que estos están en la relacción $N/2^I = 1$, es decir $I \approx \log N$. Así, la complejidad de este algoritmo es $O(\log N)$.

> **Nota** Un algoritmo tiene complejidad **logaritmica** cuando es del tipo $O(\log N)$.

1.1.3 Serie de Fibonacci y complejidad exponencial

Una operación matemática puede ser ejecutada mediante algoritmos con diferente complejidad. Por ejemplo, consideremos la serie de Fibonacci.

Esta operación puede ejecutarse de dos maneras: (1) de forma iterativa, (2) de forma recursiva

(1) Forma iterativa. complejidad $O(N)$

```
def my_fib_iter(n):

    out = [1, 1]

    for i in range(2, n+1):
        out.append(out[i - 1] + out[i - 2])

    return out[-1]
```

```
my_fib_iter(5)
```

```
8
```

(2) Forma recursiva. complejidad $O\left(2^N\right)$

```
def my_fib_rec(n):

    if n < 2:
        out = 1
    else:
        out = my_fib_rec(n-1) + my_fib_rec(n-2)

    return out
```

```
my_fib_rec(5)
```

```
8
```

Nota Un algoritmo tiene complejidad **exponencial** cuando es del tipo $O(c^N)$, donde c es una constante.

1.1.4 Notación *Big-O* y tiempo de computación

La complejidad en la notación *Big-O* nos entrega una referencia del tiempo computacional dedicado para un determinado algoritmo.

Así, por ejemplo, si consideramos un procesador Intel i7-12700K - 5GHz *(≈ 5 billones de operaciones por segundo)*:

- `my_fib_iter(100)` tomaría ≈ 0.2 nanosegundos
- `my_fib_recur(100)` tomaría ≈ 8 trillones de años

Podemos evaluar el tiempo de ejecución con la sentencia `%time`

```
%time a = my_fib_iter(30)
```

```
CPU times: user 10 µs, sys: 2 µs, total: 12 µs
Wall time: 15 µs
```

```
%time a = my_fib_rec(30) #Nota. No probar N>30
```

```
CPU times: user 216 ms, sys: 0 ns, total: 216 ms
Wall time: 216 ms
```

nota En general, se deben evitar los algoritmos de complejidad exponencial

1.2 Representación binaria y errores de reondeo

En un computador, la información es almacenada en formato binario. Un **bit** puede tener dos valores: 0 o 1. El computador es capaz de interpretar número utilizando códigos binarios.

Por ejemplo, el código de 8 bits 001000101 es equivalente a:

$$0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 37 \tag{1.1}$$

Cada variable tiene una cantidad de bits asociada.

Tipo	Nombre	Número de bits	Rango de valores
bool	Boolean	1	`True` o `Flase`
int32	Single precision integer	32	-2147483648 a 2147483647
float64	Double presition float	64	$(-1)^s 2^{e-1024}\,(1+f)$ s: 1 bit; e: 11 bits; f: 52 bits

En python el tipo de variable se asigna dinámicamente. El número de bits depende de la versión de Python y el formato de la máquina en número de bits. Por ejemplo, para una máquina de 64 bits:

```
type(34) #int64
```

```
int
```

```
type(0.4e8) #float64
```

```
float
```

Usando "Numpy", podemos controlar el número de bits asignados a una determinada variable

```
import numpy as np
np.zeros(1, dtype='int16')
```

```
array([0], dtype=int16)
```

1.2.1 Redondeo en variables tipo `float`

En variables tipo `float`, para un determinado número de bits, existe un máximo y mínimo valor que puede ser representado. En Python, valores mayores o menores a estos son representados como `inf` o `0`, respectivamente.

Podemos deterinar estos límites mediante la librería `sys`

```
import sys
sys.float_info
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.
↪2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,␣
↪epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

```
2e+308 #mayor al maximo 1.7976931348623157e+308
```

```
inf
```

```
1e-324 # menor al mínimo no normalizado 5e-324
```

```
0.0
```

```
# Mínimo no normalizado
sys.float_info.min * sys.float_info.epsilon
```

```
5e-324
```

1.2.2 Errores de redondeo en variables tipo `float`

Las **variables del tipo `int`** no son divisibles y, por lo tanto, **no sufren errores de redondeo**:

```
5 - 2 == 3
```

```
True
```

Sin embargo, una **variable del tipo `float`** es divisible. Esto significa que existe una cantidad de dígitos significativos reservados para un número, lo que **puede inducir errores de redondeo**:

```
0.1 + 0.2 + 0.3 == 0.6
```

```
False
```

Para este tipo de operaciones es recomendable utilizar la función `round`

```
round(0.1 + 0.2 + 0.3) == round(0.6)
```

```
True
```

1.2.3 Acumulacion de errores de reondeo

Cuando un código ejecuta una secuencia de operaciones, los errores de redonde suelen amplficarse.

```
# Si ejecutamos esta operación una vez
1 + 1/3 - 1/3
```

```
1.0
```

```
def add_and_subtract(iterations):
    result = 1

    for i in range(iterations):
        result += 1/3

    for i in range(iterations):
        result -= 1/3
    return result
```

```
add_and_subtract(100) # Si ejecutamos esta operación 100 veces
```

```
1.0000000000000002
```

```
add_and_subtract(1000) # Si ejecutamos esta operación 1000 veces
```

```
1.0000000000000064
```

```
add_and_subtract(10000) # Si ejecutamos esta operación 10000 veces
```

```
1.0000000000001166
```

1.3 Identificación de errores y debugging

Cuando los códigos de programación son grandes, a veces es necesario utlizar herramientas de ***debugging***. Estas herramientas nos permiten revisar las distintas etapas dentro de un algoritmo.

Podemos llamar al debugger agregando mediante la librería *python debugger* `pdb`.

Por ejemplo, consideremos la siguiente función

```
def square_number(x):
    sq = x**2
    sq += x

    return sq
```

```
square_number('10')
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_217/2602675188.py in <module>
----> 1 square_number('10')

/tmp/ipykernel_217/2802720476.py in square_number(x)
      1 def square_number(x):
----> 2     sq = x**2
      3     sq += x
      4
      5     return sq

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

Agregando la sentencia `%pdb on` antes de llamar la función podemos analizar el codigo y detectar posibles fuentes de error.

```
# llamamos al debugger de python
%pdb on
square_number('10')
```

```
Automatic pdb calling has been turned ON
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [27], in <cell line: 3>()
      1 # llamamos al debugger de python
      2 get_ipython().run_line_magic('pdb', 'on')
----> 3 square_number('10')

Input In [25], in square_number(x)
      1 def square_number(x):
----> 2     sq = x**2
      3     sq += x
      5     return sq

TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

```
> c:\users\francisco.ramirez.c\appdata\local\temp\ipykernel_13000\2802720476.
 ↪py(2)square_number()

ipdb> help

Documented commands (type help <topic>):
========================================
EOF     commands   enable     ll         pp       s                until
a       condition  exit       longlist   psource  skip_hidden      up
alias   cont       h          n          q        skip_predicates  w
args    context    help       next       quit     source           whatis
b       continue   ignore     p          r        step             where
break   d          interact   pdef       restart  tbreak
bt      debug      j          pdoc       return   u
```

(continues on next page)

(continued from previous page)

```
c       disable     jump        pfile       retval    unalias
cl      display     l           pinfo       run       undisplay
clear   down        list        pinfo2      rv        unt

Miscellaneous help topics:
==========================
exec  pdb

ipdb> h a
a(rgs)
        Print the argument list of the current function.
ipdb> a
x = '10'
ipdb> h p
p expression
        Print the value of the expression.
ipdb> p x**2
*** TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
ipdb> p locals()
{'x': '10'}
ipdb> quit
```

```
# detenemos el debugger
%pdb off
```

```
Automatic pdb calling has been turned OFF
```

También podemos agregar *breakpoints* en distintas líneas de código para detener el *debugger*.

```
import pdb
def square_number(x):

    pdb.set_trace() # agregamos un 1er breakpoint
    sq = x**2

    pdb.set_trace() # agregamos un 2do breakpoint

    sq += x

    return sq
```

```
square_number(3)
```

```
> c:\users\francisco.ramirez.c\appdata\local\temp\ipykernel_13000\1502937933.
 ↪py(5)square_number()

ipdb> a
x = 3
ipdb> p locals()
{'x': 3}
ipdb> sq
*** NameError: name 'sq' is not defined
ipdb> continue
```

(continues on next page)

(continued from previous page)

```
> c:\users\francisco.ramirez.c\appdata\local\temp\ipykernel_13000\1502937933.
 ↪py(9)square_number()

ipdb> p locals()
{'x': 3, 'sq': 9}
ipdb> sq
9
ipdb> p sq - x
6
ipdb> quit
```

Algunos comandos útiles de `pdb`:

- `help`: lista de todos los comandos del debugger
- `h` *#comando*: detalle del funcionamiento de un comando en específico
- `a` o `args`: muestra el valor del argumento de la función
- `p`: imprime el valor de una expresión específica. Usar `locals()` para mostrar valor de variables locales
- `pdb.trace()`: agrega un *breakpoint* (pausa en el código)
- `continue`: continua con el código despues de un *breakpoint*
- `quit`: finaliza el debugger.

1.4 Referencias

Kong Q., Siauw T., Bayen A. M. "Python Programming and Numerical Methods – A Guide for Engineers and Scientists", 1st Ed., Academic Press, 2021

- Capitulo 8 (Complejidad de algoritmos)
- Capítulo 9 (Representación binaria y errores de redondeo)
- Calítulo 10 (Identificación de errores y debugging)

MEC301 - Metodos Numéricos

CHAPTER

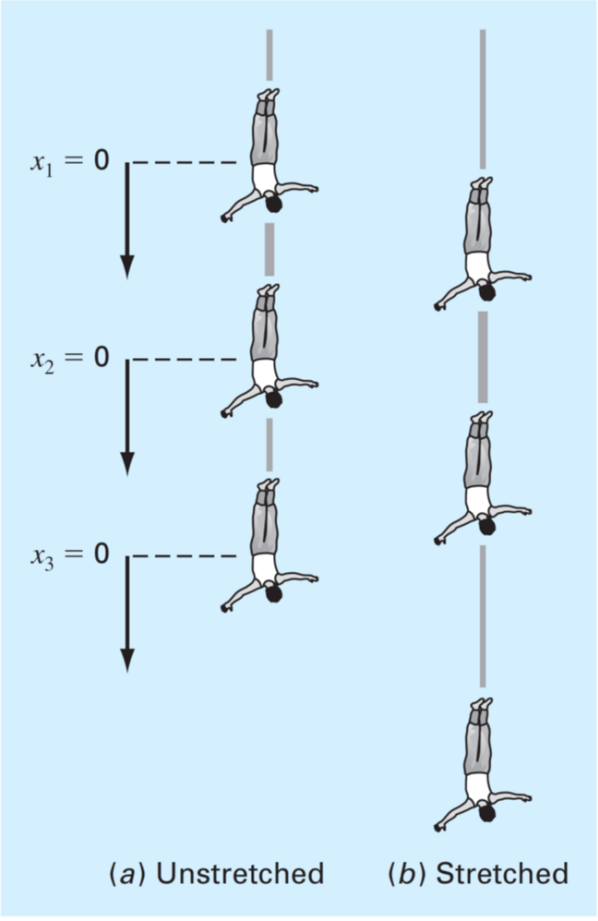
TWO

1.2 ALGEBRA LINEAL Y SISTEMAS DE ECUACIONES LINEALES

Profesor: Francisco Ramírez Cuevas Fecha: 8 de Agosto 2022

2.1 Introducción a los sistemas de ecuaciones lineales

Consideremos el caso de tres personas conectada por cuerdas elásticas.

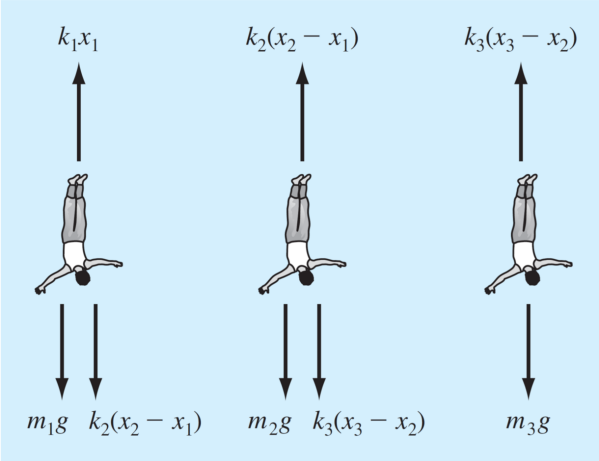


En la primera figura (a), los tres cuerpos están en la posición inicial de forma que los elásticos están totalmente extendidos, **pero no estirados**. Definimos el cambio en la posición inicial de cada persona, como: x_1, x_2, x_3.

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



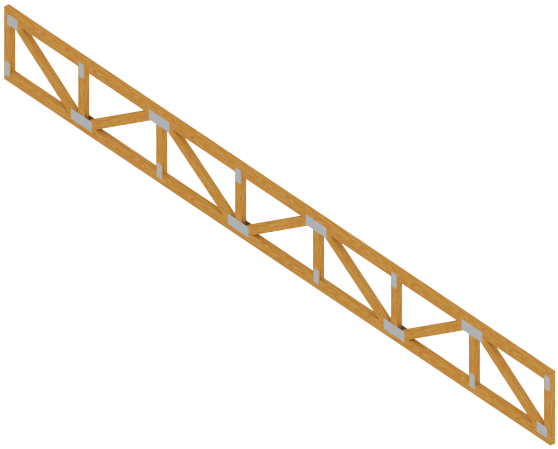
$$m_1\frac{d^2x_1}{dt^2} = m_1g + k_2(x_2 - x_1) - k_1x_1$$
$$m_2\frac{d^2x_2}{dt^2} = m_2g + k_3(x_3 - x_2) - k_2(x_1 - x_2)$$
$$m_3\frac{d^2x_3}{dt^2} = m_3g + k_3(x_2 - x_3)$$

En condiciones de equilibrio:

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

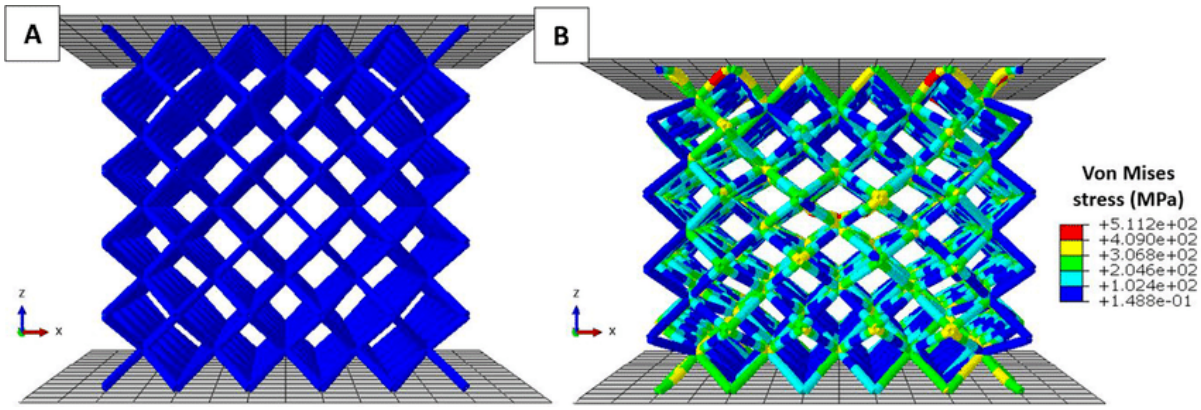
En el ejemplo anterior, derivamos un sistema de ecuaciones lineales con 3 incognitas el cual podemos resolver con técnicas analíticas.

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 incognitas 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).



2.1.1 Definición general

Decimos que una ecuación es lineal cuando:

1. Todas sus incognitas están **únicamente** separadas por sumas o restas
2. El exponente de cada incognita 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 la podemos transformar 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 esta compuesto por más de una ecuación lineal, tal como en el ejemplo de las personas conectadas por cuerdas elásticas

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

2.1.2 Representación matricial

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

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

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

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

La forma matricial de esta ecuación tiene la siguiente forma:

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

O, similarmente,

$$Ax = y,$$

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

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:

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

2.1.3 Repaso de matrices

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

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

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:

$$to$$

$$|M| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} =$$

$$a \begin{vmatrix} \square & \square & \square \\ \square & e & f \\ \square & h & i \end{vmatrix} - b \begin{vmatrix} \square & \square & \square \\ d & \square & f \\ g & \square & i \end{vmatrix} + c \begin{vmatrix} \square & \square & \square \\ d & e & \square \\ g & h & \square \end{vmatrix}$$

$$=$$

$$a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$$

$$=$$

$$aei - afh + bfg - bdi + cdh - ceg$$

b
e cd
h fg
i

= 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

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.

2.1.4 Representación en python

Para representar sistemas de ecuciones lineales en python utilizamos variables del tipo *numyp 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 libreria `numpy`.

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

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

2.1.5 Caracterización de sistemas de ecuaciones lineales

Un sistema de ecuaciones lineales tiene solución única, si y solo si el número de incognitas 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: $(\text{ec. } 3) = 2 \times (\text{ec. } 1) + (\text{ec. } 2)$

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

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:

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

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

- **El sistema tiene solución única** si rank $([A|y])$ = rank (A), y rank $(A) = n$
- **El sistema tiene infinitas soluciones** si rank $([A|y])$ = rank (A), y rank $(A) < n$
- **El no tiene soluciones** si rank $([A|y])$ = 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 incognitas, 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 incognitas, n = 3
```

$$\text{rank}\,([A|y]) = \text{rank}\,(A)$$
$$\text{rank}\,(A) n$$

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_242/453559162.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 "$\text{Cond}(A)$*" 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.

2.2 Métodos de solución directos

2.2.1 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: Deteriminamos 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 atras.

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

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

2.2.2 Factorización LU

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

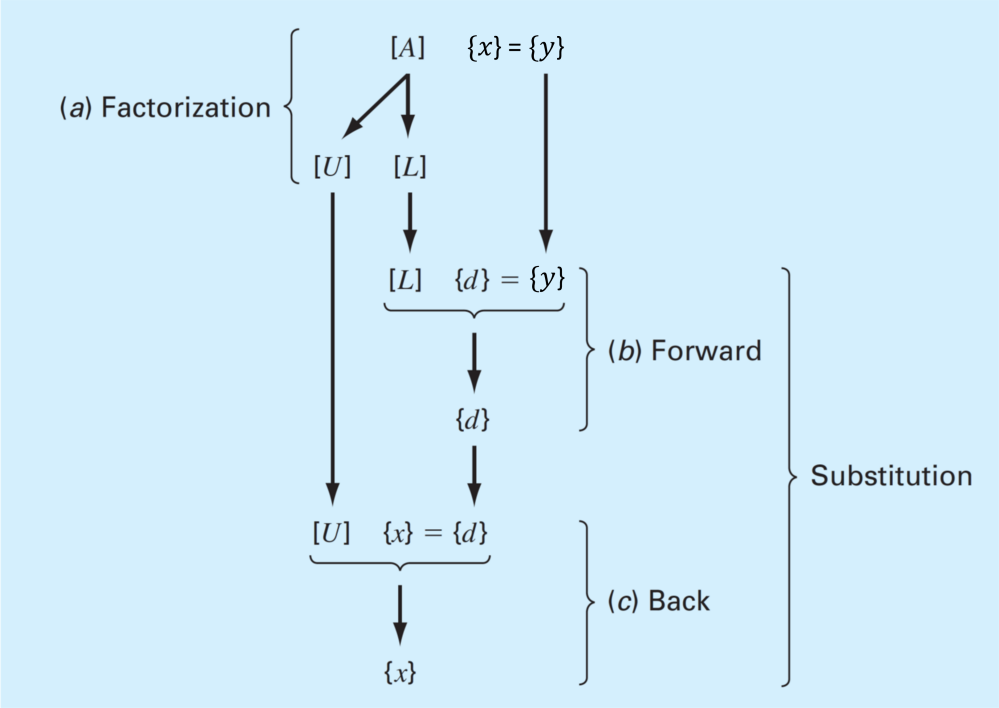
$$A = LU$$

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	$L_{ii} = 1, i = 1, 2, ..., n$
Crout	$U_{ii} = 1, i = 1, 2, ..., n$
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 factorizacion 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 explicita 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 permitaciones.

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.]]
```

2.3 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).

2.3.1 Gauss-Seidel

El algoritmo se puede resumir en los siguientes pasos:

1. Asumimos un valor inicial para $x_2^{(0)}, x_3^{(0)}, \cdots, x_n^{(0)}$ (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 métodod 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
```

(continues on next page)

(continued from previous page)

```
    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  ])
```

2.4 Solución de sistemas de ecuaciones lineales con python

En python la forma más facil de resolver sistemas de ecuaciones lineal 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 incial 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.]]
```

(continues on next page)

(continued from previous page)

```
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.]]
```

2.5 Referencias

- Kong Q., Siauw 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