4101553 Métodos Numéricos aplicados a la Ingenieria Civil

Departamento de Ingeniería Civil

Universidad Nacional de Colombia

Sede Manizales

Docente: Juan Nicolás Ramírez Giraldo (jnramirezg@unal.edu.co (mailto:jnramirezg@unal.edu.co))

"*Cum cogitaveris quot te antecedant, respice quot sequantur*"

Séneca

Repositorio de la asignatura (https://github.com/jnramirezg/metodos_numericos_ingenieria_civil/)

Unidad 1: Sistemas de ecuaciones lineales

Métodos de Gauss y Gauss-Jordan

1. Gauss simple
2. Gauss Jordan

Algunas matrices especiales

Previamente definimos algunas matrices especiales que, serán de utilidad para el desarrollo conceptual.

Matriz triangular superior

$$\underline{\underline{U}} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

Matriz triangular inferior

$$\underline{\underline{L}} = \begin{bmatrix} l_{11} & 0 & 0 & \dots & 0 \\ l_{21} & l_{22} & 0 & \dots & 0 \\ l_{31} & l_{32} & l_{33} & \dots & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots \\ l_{n1} & l_{n2} & l_{n3} & \dots & l_{nn} \end{bmatrix}$$

Matriz simétrica

$$\underline{\underline{S}} = \begin{bmatrix} s_{11} & s_{21} & s_{31} & \dots & s_{n1} \\ s_{21} & s_{22} & s_{32} & \dots & s_{n2} \\ s_{31} & s_{32} & s_{33} & \dots & s_{n3} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ s_{n1} & s_{n2} & s_{n3} & \dots & s_{nn} \end{bmatrix}$$

En la que se cumple que $s_{ij} = s_{ji}$ para todo i, j, es decir, $\underline{\underline{S}} = \underline{\underline{S}}^T$.

Matriz escalonada
En un caso particular de 4x5:

$$\underline{\underline{E}} = \begin{bmatrix} 1 & e_{12} & e_{13} & e_{14} & e_{15} \\ 0 & 1 & e_{23} & e_{24} & e_{25} \\ 0 & 0 & 1 & e_{34} & e_{35} \\ 0 & 0 & 0 & 1 & e_{45} \end{bmatrix}$$

Matriz escalonada reducida
En un caso particular de 4x5:

Método de Gauss

Es un método utilizado para resolver un sistema de n ecuaciones:

$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1$

$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \ldots + a_{2n}x_n = b_2$

.

.

.

$a_n x_1 + a_{n2}x_2 + a_{n3}x_3 \ldots + a_{nn}x_n = b_n$

Para hallar la solución se realizan dos fases:

- eliminación de incógnitas hacia adelante,
- hallar las soluciones mediante sustitución hacia atrás.

Acá aparece el concepto de matriz aumentada del sistema, en donde se agrega en la última columna de la matriz de coeficientes $\underline{\underline{A}}$ el vector $\underline{B}$ con las constantes, de esta manera:

$$[\underline{\underline{A}}|\underline{B}]$$

En particular, si $\underline{\underline{A}}$ está definida como:

$$\underline{\underline{A}} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Y $\underline{B}$ como:

$$\underline{B} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

La matriz aumentada del sistema es:

$$[\underline{\underline{A}}|\underline{B}] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

Eliminación de incógnitas hacia adelante

A partir de la matriz aumentada del sistema:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

Se busca llevarla a que sea una matriz escalonada, en donde, es matricialmente válido realizar este tipo de operaciones dentro de la matriz:

1. Multiplicar una fila por un escalar:
   - $f_1 = (-2)f_1$
   - $f_3 = (8)f_3$
2. Sumar o restar filas (suma algebraica):
   - $f_1 = f_1 + f_2$
   - $f3 = f_3 - f_2$
3. Sumar filas multiplicadas por escalares:
   - $f_2 = (0.5)f_1 + f_2$
   - $f_1 = (8)f_1 - (3)f_3$

Nota: Esto es válido porque de ninguna manera afecta la consistencia ni la solución del sistema,

únicamente altera la escala y la cantidad de variables visibles en cada ecuación. Esto solo se puede hacer en **matrices aumentadas** porque al hacer alguna operación se alteran ambos lados de la igualdad: los coeficientes constantes de $\underline{\underline{A}}$ y las constantes de $\underline{B}$.

Luego, no se hacen operaciones entre columnas porque se altera el orden de las soluciones. Si se cambia la columna 2 por la columna 3, entonces la solución x_2 queda en la tercera posición y la x_3 en la segunda.

A partir de estas operaciones entre filas de la matriz aumentada, se empieza un proceso **iterativo** en el que se toma inicialmente el elemento en la posición $i = 1, j = 1$ (en Python sería el 0,0) como *pivote* o *coeficiente*, de tal manera que se obtenga un factor que al aplicarlo a la fila 1 f_1 y lugo sumarle a la fila 2 f_2, elimine el primer término de la fila 2.

$$f_2 = f_2 + (\frac{-a_{21}}{a_{11}})f_1$$

Esto es:

$$f_2 = \begin{bmatrix} a_{21} & a_{22} & a_{23} & | & b_2 \end{bmatrix} + (\frac{-a_{21}}{a_{11}}) * \begin{bmatrix} a_{11} & a_{12} & a_{13} & | & b_1 \end{bmatrix}$$

$$f_2 = \begin{bmatrix} a_{21} & a_{22} & a_{23} & | & b_2 \end{bmatrix} + \begin{bmatrix} (\frac{-a_{21}}{a_{11}})a_{11} & (\frac{-a_{21}}{a_{11}})a_{12} & (\frac{-a_{21}}{a_{11}})a_{13} & | & (\frac{-a_{21}}{a_{11}})b_1 \end{bmatrix}$$

$$f_2 = \begin{bmatrix} a_{21} & a_{22} & a_{23} & | & b_2 \end{bmatrix} + \begin{bmatrix} -a_{21} & (\frac{-a_{21}}{a_{11}})a_{12} & (\frac{-a_{21}}{a_{11}})a_{13} & | & (\frac{-a_{21}}{a_{11}})b_1 \end{bmatrix}$$

$$f_2 = \begin{bmatrix} 0 & a_{22} + (\frac{-a_{21}}{a_{11}})a_{12} & a_{23} + (\frac{-a_{21}}{a_{11}})a_{13} & | & b_2 + (\frac{-a_{21}}{a_{11}})b_1 \end{bmatrix}$$

Para facilidad de visualización:

$$f_2 = \begin{bmatrix} 0 & a'_{22} & a'_{23} & | & b'_2 \end{bmatrix}$$

De esta manera, la matriz aumentada queda:

$$[\underline{\underline{A}}|\underline{B}] = \begin{bmatrix} a_{11} & a_{12} & a_{13} & | & b_1 \\ 0 & a'_{22} & a'_{23} & | & b'_2 \\ a_{31} & a_{32} & a_{33} & | & b_3 \end{bmatrix}$$

De forma análoga, se hace:

$$f_3 = f_3 + (\frac{-a_{21}}{a_{11}})f_1$$

Para que la matriz escalonada quede:

$$[\underline{\underline{A}}|\underline{B}] = \begin{bmatrix} a_{11} & a_{12} & a_{13} & | & b_1 \\ 0 & a'_{22} & a'_{23} & | & b'_2 \\ 0 & a'_{32} & a'_{33} & | & b'_3 \end{bmatrix}$$

Ahora, se establece como pivote a'_{22}, y se realiza la siguiente operación:

$$f_3 = f_3 + (\frac{-a'_{32}}{a'_{22}})f_2$$

Obteniendo:

$$[\underline{\underline{A}}|\underline{B}] = \begin{bmatrix} a_{11} & a_{12} & a_{13} & | & b_1 \\ 0 & a'_{22} & a'_{23} & | & b'_2 \end{bmatrix}$$

Solución mediante sustitución hacia atrás

A partir del último resultado, se extrae de la tercera fila la solución de x_3, así:

$$a''_{33}.\, x_3 = b''_3$$

en donde:

$$x_3 = \frac{b''_3}{a''_{33}}$$

Luego, de la segunda fila:

$$a'_{22}.\, x_2 + a'_{23}.\, x_3 = b'_2$$

Como ya sabemos que $x_3 = \frac{b''_3}{a''_{33}}$, entonces se reemplaza en resultado anterior, dejando únicamente a x_2 como incógnita. Y ya teniendo los resultados de x_2 y x_3 se extrae la ecuación de la primera fila, se reemplaza y se despeja x_1.

Primeros pasos para solución en Python con lista de listas de la eliminación de incógnitas hacia adelante

El resultado final es:

```
def sol_gauss(A, B):
    m = len(A)              # Número de filas de A.
    S = [f[:] for f in A]   # Se crea S inicialmente como una copi
a de A.

    for f in range(m):      # Agrego al final de cada fila, el cor
respondiente elemento de B
        S[f] += [B[f]]

    for k in range(m-1):
        pivote = S[k][k]  # El elemento de la fila 1, columna 1.
```

In [1]:
```
# Matriz de coeficientes constantes.
A = [
     [ 2,  1, -1,  2],
     [ 4,  5, -3,  6],
     [-2,  5, -2,  6],
     [ 4, 11, -4,  8]
                    ]
```

In [2]:
```
# Vector de constantes.
B = [5, 9, 4, 2]
```

El primer reto es construir la matriz aumentada $\underline{\underline{S}}$:

In [3]:
```
m = len(A)              # Número de filas de A, número de soluciones, tama
```

In [4]:
```
S = [f[:] for f in A]  # Se crea S inicialmente como una copia de A.
```

In [5]:
```
S
```

Out[5]:
```
[[2, 1, -1, 2], [4, 5, -3, 6], [-2, 5, -2, 6], [4, 11, -4, 8]]
```

In [6]:
```
# Agrego al final de cada fila, el correspondiente elemento de B
for f in range(m):
    S[f] += [B[f]]
```

In [7]:
```
S
```

Out[7]:
```
[[2, 1, -1, 2, 5], [4, 5, -3, 6, 9], [-2, 5, -2, 6, 4], [4, 11, -4, 8,
2]]
```

Después, es necesario establecer el primer pivote:

In [8]:
```
pivote = S[0][0] # El elemento de la fila 0, columna 0.
```

In [9]:
```
# Se crea una copia auxiliar de S, que recibirá los resultados:
S1 = [f[:] for f in S]
```

Se crean 3 ciclos (m-1) para recorrer la segunda, tercera y cuarta fila realizando esta operación:

$$f_i = f_i + (\frac{-a_{i1}}{a_{11}})f_1$$

El recorrido lo realiza j, el cual debe hacerse m+1 veces, debido a que la matriz tiene una columna más al ser **aumentada**.

In [10]:
```
# f2 = f2 + (-a21/pivote)f1
factor = -S[1][0]/pivote
for j in range(m+1):
    S1[1][j] = S[1][j] + factor*S[0][j]
```

In [11]:
```
S1
```

Out[11]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [-2, 5, -2, 6, 4],
 [4, 11, -4, 8, 2]]
```

In [12]:
```
# f3 = f3 + (-a31/pivote)f1
factor = -S[2][0]/pivote
for j in range(m+1):
    S1[2][j] = S[2][j] + factor*S[0][j]
```

In [13]:
```
S1
```

Out[13]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 6.0, -3.0, 8.0, 9.0],
 [4, 11, -4, 8, 2]]
```

In [14]:
```
# f4 = f4 + (-a41/pivote)f1
factor = -S[3][0]/pivote
for j in range(m+1):
    S1[3][j] = S[3][j] + factor*S[0][j]
```

In [15]:
```
S1
```

Out[15]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 6.0, -3.0, 8.0, 9.0],
 [0.0, 9.0, -2.0, 4.0, -8.0]]
```

Luego, estos tres ciclos se pueden condensar en un ciclo dentro de otro ciclo, donde claramente, los elementos que varían, por ejemplo, en el último ciclo, son aquellos donde está el 3:

```
    factor = -S[3][0]/pivote
    for j in range(m+1):
        S1[3][j] = S[3][j] + factor*S[0][j]

    factor = -S[2][0]/pivote
    for j in range(m+1):
        S1[2][j] = S[2][j] + factor*S[0][j]
```

In [16]:
```
# Si el pivote está en la primera fila, el recorrido debe hacerse desde l
# la última es decir en range(1, m).
for i in range(1, m):
    factor = -S[i][0]/pivote
    for j in range(0, m+1):
        S1[i][j] = S[i][j] + factor*S[0][j]
```

In [17]:
```
S1
```

Out[17]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 6.0, -3.0, 8.0, 9.0],
 [0.0, 9.0, -2.0, 4.0, -8.0]]
```

In [18]:
```
# Se almacenan los resultados nuevamente en S.
S = S1
```

Se repite el proceso para la columna 2, estableciendo como nuevo pivote el segundo elemento de la diagonal. Nótese que es distinto al original que traía la matriz:

In [19]:
```
pivote = S[1][1] # El elemento de la fila 1, columna 1 en Python.
```

In [20]:
```
# Nuevamente se crea una copia auxiliar de S, que recibirá los resultados
S1 = [f[:] for f in S]
```

Se crean 2 ciclos (m-2) para recorrer tercera y cuarta fila realizando esta operación:

$$f_i = f_i + (\frac{-a'_{i2}}{a'_{22}})f'_2$$

El recorrido lo realiza j, el cual debe hacerse m+1 veces, debido a que la matriz tiene una columna más al ser **aumentada**. Sin embargo, el recorrido por el primer elemento de las filas es innecesario porque ya vale 0, por lo tanto, el recorrido se hace en range(1, m+1).

In [21]:
```
# f3 = f3 + (-a'32/pivote)f2
factor = -S[2][1]/pivote
for j in range(1, m+1):
    S1[2][j] = S[2][j] + factor*S[1][j]
```

In [22]:
```
S1
```

Out[22]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 9.0, -2.0, 4.0, -8.0]]
```

In [23]:
```
# f4 = f4 + (-a'42/pivote)f2
factor = -S[3][1]/pivote
for j in range(1, m+1):
    S1[3][j] = S[3][j] + factor*S[1][j]
```

In [24]:
```
S1
```

Out[24]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 0.0, 1.0, -2.0, -5.0]]
```

De forma análoga con los resultados de la columna 1. Los dos ciclos se pueden condensar en un ciclo dentro de otro ciclo, donde claramente, los elementos que varían, por ejemplo, en el último ciclo, son aquellos donde está el 3:

```
factor = -S[3][1]/pivote
for j in range(1, m+1):
    S1[3][j] = S[3][j] + factor*S[1][j]
```

In [25]:
```
# Como el pivote está en la segunda fila, el recorrido debe hacerse desde
# la última es decir en range (2, m).
for i in range(2, m):
    factor = -S[i][1]/pivote
    for j in range(1, m+1):
        S1[i][j] = S[i][j] + factor*S[1][j]
```

In [26]:
```
list(range(2, m))
```

Out[26]:
```
[2, 3]
```

In [27]:
```
S1
```

Out[27]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 0.0, 1.0, -2.0, -5.0]]
```

In [28]:
```
# Se almacenan los resultados nuevamente en S.
S = S1
```

Por último, es claro que para la tercera columna debe aplicar:

$$f_i = f_i + (\frac{-a''_{i3}}{a''_{33}})f'_3$$

In [29]:
```
pivote = S[2][2] # El elemento de la fila 1, columna 1.

S1 = [f[:] for f in S]

# Como el pivote está en la tercera fila, el recorrido debe hacerse solo
# es decir, la última (3, m).
for i in range(3, m):  # Acá aumento 1, respecto al paso anterior.
    factor = -S[i][2]/pivote
    for j in range(2, m+1):  # Acá aumento 1 respecto al paso anterior.
        S1[i][j] = S[i][j] + factor*S[2][j]
```

In [30]:
```
list(range(3, m))
```

Out[30]:
```
[3]
```

In [31]:
```
S1
```

Out[31]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 0.0, 0.0, 2.0, 6.0]]
```

Sintetizando todo lo aterior, se pone en una sola línea los resultados más importantes.

In [32]:
```
m = len(A)              # Número de filas de A.
S = [f[:] for f in A]   # Se crea S inicialmente como una copia de A.

# Agrego al final de cada fila, el correspondiente elemento de B
for f in range(m):
    S[f] += [B[f]]

## Primera columna ##
pivote = S[0][0] # El elemento de la fila 0, columna 0.

S1 = [f[:] for f in S] # Copia auxiliar de S.

# Operaciones: # fi = fi + (-ai1/pivote)f1
for i in range(1, m):
    factor = -S[i][0]/pivote
    for j in range(0, m+1):
        S1[i][j] = S[i][j] + factor*S[0][j]

S = S1

## Segunda columna ##
pivote = S[1][1] # El elemento de la fila 1, columna 1

# Operaciones: # fi = fi + (-ai2/pivote)f2
for i in range(2, m):
    factor = -S[i][1]/pivote
```

```
    for j in range(1, m+1):
        S1[i][j] = S[i][j] + factor*S[1][j]

S = S1

## Tercera columna ##
pivote = S[2][2] # El elemento de la fila 1, columna 1.

S1 = [f[:] for f in S]

# Operaciones: # fi = fi + (-ai3/pivote)f3
for i in range(3, m):
    factor = -S[i][2]/pivote
    for j in range(2, m+1):
        S1[i][j] = S[i][j] + factor*S[2][j]

S = S1
```

In [33]:
```
S1
```

Out[33]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 0.0, 0.0, 2.0, 6.0]]
```

Pero es evidente que aún se puede automatizar más.

Por ejemplo, en el proceso para la fila 1, la fila 2 y la fila 3 tienen bastantes similitudes:

```
for i in range(1, m):
    factor = -S[i][0]/pivote
    for j in range(0, m+1):
        S1[i][j] = S[i][j] + factor*S[0][j]

for i in range(2, m):
    factor = -S[i][1]/pivote
    for j in range(1, m+1):
        S1[i][j] = S[i][j] + factor*S[1][j]

for i in range(3, m):
    factor = -S[i][2]/pivote
    for j in range(2, m+1):
        S1[i][j] = S[i][j] + factor*S[2][j]
```

De esta manera, se establece una variable k que hace el recorrido m-1, es decir, una vez menos que la cantidad de filas del sistema. Definiendo k=0 para el primer ciclo: donde está el 1 se pone k+1 y donde está el 0 se pone k

```
    for i in range(1, m):
        factor = -S[i][0]/pivote
        for j in range(0, m+1):
            S1[i][j] = S[i][j] + factor*S[0][j]

    for i in range(k+1, m):
```

Una forma en Python con lista de listas de la eliminación de incógnitas hacia adelante

In [34]:
```
m = len(A)              # Número de filas de A.
S = [f[:] for f in A]   # Se crea S inicialmente como una copia de A.

for f in range(m):      # Agrego al final de cada fila, el correspondiente
    S[f] += [B[f]]

for k in range(m-1):
    pivote = S[k][k]  # El elemento de la fila 1, columna 1.
    S1 = [f[:] for f in S]
    for i in range(k+1, m):  # Operaciones: # fi = fi + (-aik/pivote)fk
        factor = -S[i][k]/pivote
        for j in range(k, m+1):
            S1[i][j] = S[i][j] + factor*S[k][j]
    S = S1
```

In [35]:
```
S
```

Out[35]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 0.0, 0.0, 2.0, 6.0]]
```

Solución en Python con lista de listas para la sustitución hacia atrás

In [36]:
```
# A partir de los resultados obtenidos de la matriz aumentada en forma es
S
```

Out[36]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 0.0, 0.0, 2.0, 6.0]]
```

Se crea un espacio de memoria para almacenar la solución, inicialmente se llena de ceros de tamaño **m**, es decir, con la cantidad de incógnitas o filas del sistema.

In [37]:
```
sol = [0]*m  # Es una forma sencilla y general de crear una lista llena d
```

In [38]:
```
sol
```

Out[38]:
```
[0, 0, 0, 0]
```

Se halla la última solución, la asociada a la última fila. En Python, la m-1.

$$2x_4 = 6$$

$$x_4 = 3$$

In [39]:
```
S[3][4]/S[3][3]
```

Out[39]:
```
3.0
```

In [40]:
```
# Sin embargo, hay una forma más general que será muy útil para crear una
S[m-1][m]/S[m-1][m-1]
```

Out[40]:
```
3.0
```

In [41]:
```
# Se almacena adecuadamente
sol[m-1] = S[m-1][m]/S[m-1][m-1]
```

In [42]:
```
sol
```

Out[42]:
```
[0, 0, 0, 3.0]
```

Se halla la penúltima solución, la asociada a la penúltima fila o la tercera. En Python, la m-2 o la 2.

```
[0.0, 0.0, -1.0, 4.0, 11.0]
```

$$-x_3 + 4x_4 = 11$$

Ya se conoce el valor de x_4, se reemplaza y se despeja x_3

$$-x_3 + 4(3) = 11$$

$$-x_3 = 11 - 12$$

$$x_3 = 1$$

Viéndolo un poco más sencillo en término de Python:

`S[2][2]` x_3 + `S[2][3]` x_4 = `S[2][4]`

Y

x_4 = `sol[3]`

Es decir,

x_3 = (S[2][4]-S[2][3]*sol[3]) / S[2][2]

In [43]:
```
(S[2][4]-S[2][3]*sol[3]) / S[2][2]
```

Out[43]:
```
1.0
```

In [44]:
```
# Se pone de una manera más general en términos de m (m=4)
(S[m-2][m]-S[m-2][m-1]*sol[m-1]) / S[m-2][m-2]
```

Out[44]:
```
1.0
```

In [45]:
```
# Se almacena adecuadamente
sol[m-2] = (S[m-2][m] - S[m-2][m-1]*sol[m-1]) / S[m-2][m-2]
```

In [46]:
```
sol
```

Out[46]:
```
[0, 0, 1.0, 3.0]
```

Se halla la segunda solución, la asociada a la segunda fila. En Python, la 1 o la m-3.

In [47]:
```
# Recordando la matriz aumentada.
S
```

Out[47]:
```
[[2, 1, -1, 2, 5],
 [0.0, 3.0, -1.0, 2.0, -1.0],
 [0.0, 0.0, -1.0, 4.0, 11.0],
 [0.0, 0.0, 0.0, 2.0, 6.0]]
```

Se extrae la información de la segunda fila (la 1 en Python).

S[1][1] x_2 + S[1][2] x_3 + S[1][3] x_4 = S[1][4]

Pero:

x_3 = sol[2]

x_4 = sol[3]

Entonces,

S[1][1] x_2 + S[1][2]*sol[2] + S[1][3]*sol[3] = S[1][4]

Es decir,

x_2 = (S[1][4] - S[1][2]*sol[2] - S[1][3]*sol[3]) / S[1][1]

In [48]:
```
(S[1][4] - S[1][2]*sol[2] - S[1][3]*sol[3]) / S[1][1]
```

Out[48]:
```
-2.0
```

In [49]:
```
# Se pone de una manera más general en términos de m (m=4)
(S[m-3][m] - S[m-3][m-2]*sol[m-2] - S[m-3][m-1]*sol[m-1]) / S[m-3][m-3]
```

Out[49]: -2.0

In [50]:
```
# Se almacena adecuadamente
sol[m-3] = (S[m-3][m] - S[m-3][m-2]*sol[m-2] - S[m-3][m-1]*sol[m-1]) / S[
```

In [51]:
```
sol
```

Out[51]: [0, -2.0, 1.0, 3.0]

Se recopilan los códigos usados para hallar las tres soluciones:

```
sol[m-1] = S[m-1][m] / S[m-1][m-1]
sol[m-2] = (S[m-2][m] - S[m-2][m-1]*sol[m-1]) / S[m-2][m-2]
sol[m-3] = (S[m-3][m] - S[m-3][m-2]*sol[m-2] - S[m-3][m-1]*sol[m-
1]) / S[m-3][m-3]
```

Se trata de mejorar y unificar la presentación para identificar patrones.

```
sol[m-1] = (S[m-1][m]
) / S[m-1][m-1]
sol[m-2] = (S[m-2][m] - S[m-2][m-1]*sol[m-1])
/ S[m-2][m-2]
sol[m-3] = (S[m-3][m] - S[m-3][m-1]*sol[m-1] - S[m-3][m-2]*sol[m-
2]) / S[m-3][m-3]
```

Por analogía se obtiene la primera solución, la asociada a la primera fila con:

```
sol[m-4] = (S[m-4][m] - S[m-4][m-1]*sol[m-1] - S[m-4][m-2]*sol[m-
2] - S[m-4][m-3]*sol[m-3]) / S[m-4][m-4]
```

In [52]:
```
# Se almacena la primera solución.
sol[m-4] = (S[m-4][m] - S[m-4][m-1]*sol[m-1] - S[m-4][m-2]*sol[m-2] - S[m
```

In [53]:
```
sol
```

Out[53]: [1.0, -2.0, 1.0, 3.0]

¿Cómo se podría resumir el anterior resultado en un ciclo?

In [54]:
```
sol = [0]*m  # Se limpia el espacio de memoria de la solución.
```

Se debe definir un ciclo que recorra `[1, 2, 3, 4]` , es decir, de 1 a m+1, en términos de Python `range(1, m+1)`

In [55]:
```
list(range(1, m+1))
```

Out[55]:
```
[1, 2, 3, 4]
```

Se evidencia que todos las soluciones deben hacer esta operación común, al menos.

```
S[m-1][m] / S[m-1][m-1]
S[m-2][m] / S[m-2][m-2]
S[m-3][m] / S[m-3][m-3]
S[m-4][m] / S[m-4][m-4]
```

Es decir, en términos de una variable `i` que recorra:

```
S[m-i][m] / S[m-i][m-i]
```

Se implenta:

In [56]:
```
# Donde m-i representa las soluciones 3, 2, 1, 0. Y la variable i, los pa
for i in range(1, m+1):
    sol[m-i] +=S[m-i][m] / S[m-i][m-i]
```

Luego, se evidencia que para el paso 2, aparece:

```
-S[m-2][m-1]*sol[m-1]
```

En el 3:

```
-S[m-3][m-1]*sol[m-1] - S[m-3][m-2]*sol[m-2]
```

Y en el 4:

```
-S[m-4][m-1]*sol[m-1] - S[m-4][m-2]*sol[m-2] - S[m-4][m-3]*sol[m-
3]
```

Lo que me indica que, se debería hacer otro ciclo, pero con unos límites de evaluación variables. Es decir, para el primer paso: no evaluar; para el segundo paso: solo una evaluación; para el tercer paso: dos evaluaciones; y para el último paso: tres evaluaciones.

Definiendo una variable j que recorra con una cantidad de pasos variables que vayan aportando al resultado final. Si la primera evaluación de j es 1, entonces, en el segundo paso:

`S[m-2][m-1]*sol[m-1]`

`S[m-2][m-j]*sol[m-j]` para `j in [1]`

En el tercer paso:

`S[m-3][m-j]*sol[m-j]` para `j in [1, 2]`

En el cuarto paso:

`S[m-4][m-j]*sol[m-j]` para `j in [1, 2, 3]`

Es claro que `j` no sirve para los terminos `[m-2]`, `[m-3]` y `[m-4]`. Pero que `i` sí tiene el mismo comportamiento en `sol[m-i]` y `S[m-i]`, como se muestra en la primera adición hecha a las soluciones:

```
for i in range(1, m+1):
    sol[m-i] +=S[m-i][m] / S[m-i][m-i]
```

Y adicionalmente, la mejor para controlar los pasos que debe hacer `j` que, poniendo a `i` como límite.

En el segundo paso:

`sol[m-i] += S[m-i][m-j]*sol[m-j]` para `j in [1]` o en `j in [1, i]`, donde `i=2`

En el tercer paso:

`sol[m-i] += S[m-i][m-j]*sol[m-j]` para `j in [1, 2]` o en `j in [1, i]`, donde `i=3`

En el cuarto paso:

`sol[m-i] += S[m-i][m-j]*sol[m-j]` para `j in [1, 2, 3]` o en `j in [1, i]`,

In [57]:
```
# i toma valores [1, 2, 3, 4]
i = 1
list(range(1, i))
```

Out[57]:
```
[]
```

In [58]:
```
i = 2
list(range(1, i))
```

Out[58]:
```
[1]
```

In [59]:
```
i = 4
list(range(1, i))
```

Out[59]:
```
[1, 2, 3]
```

In [60]:
```
sol = [0]*m  # Se limpia el espacio de memoria de la solución.

for i in range(1, m+1):
    sol[m-i] +=S[m-i][m] / S[m-i][m-i]
    for j in range(1, i):
        sol[m-i] += -S[m-i][m-j]*sol[m-j]/ S[m-i][m-i]
```

In [61]:
```
sol
```

Out[61]:
```
[1.0, -2.0, 1.0, 3.0]
```

Se prueban todos los ciclos juntos.

In [62]:
```
m = len(A)              # Número de filas de A.
S = [f[:] for f in A]   # Se crea S inicialmente como una copia de A.

for f in range(m):      # Agrego al final de cada fila, el correspondiente
    S[f] += [B[f]]

# Eliminación hacia adelante
for k in range(m-1):
    pivote = S[k][k]  # El elemento de la fila 1, columna 1.
    S1 = [f[:] for f in S]
    for i in range(k+1, m):  # Operaciones: # fi = fi + (-aik/pivote)fk
        factor = -S[i][k]/pivote
        for j in range(k, m+1):
            S1[i][j] = S[i][j] + factor*S[k][j]
    S = S1

# Sustitución hacia atrás
sol = [0]*m  # Se limpia el espacio de memoria de la solución.

for i in range(1, m+1):
    sol[m-i] +=S[m-i][m] / S[m-i][m-i]
    for j in range(1, i):
        sol[m-i] += -S[m-i][m-j]*sol[m-j]/ S[m-i][m-i]
```

In [63]:
```
sol
```

Out[63]:
```
[1.0, -2.0, 1.0, 3.0]
```

Función sol_gauss

In [64]:
```
def sol_gauss(A, B):
    '''
        Función que a partir de:
        A: matriz de coeficientes constantes definida como lista de lista
        B: matriz de constantes definida como una lista.

        Devuelve las soluciones del sistema AX=B
    '''
    m = len(A)              # Número de filas de A, cantidad de soluciones
    S = [f[:] for f in A]   # Se crea S inicialmente como una copia de A.

    for f in range(m):      # Agrego al final de cada fila, el correspondi
        S[f] += [B[f]]

    for k in range(m-1):
        pivote = S[k][k]  # El elemento de la fila 1, columna 1.
```

```
        S1 = [f[:] for f in S]
        for i in range(k+1, m):  # Operaciones: # fi = fi + (-aik/pivote)
            factor = -S[i][k]/pivote
            for j in range(k, m+1):
                S1[i][j] = S[i][j] + factor*S[k][j]
        S = S1

    sol = [0]*m  # Se limpia el espacio de memoria de la solución.

    for i in range(1, m+1):
        sol[m-i] +=S[m-i][m] / S[m-i][m-i]
        for j in range(1, i):
            sol[m-i] += -S[m-i][m-j]*sol[m-j]/ S[m-i][m-i]

    return sol
```

Ver: [06-gauss_simple_v1.py (https://github.com/jnramirezg/metodos_numericos_ingenieria_civil/blob/main/codigo/06-gauss_simple_v1.py)](https://github.com/jnramirezg/metodos_numericos_ingenieria_civil/blob/main/codigo/06-gauss_simple_v1.py)

In [65]:
```
sol_gauss(A, B)
```

Out[65]:
```
[1.0, -2.0, 1.0, 3.0]
```

In [66]:
```
# Ejemplo 2.
A1 = [[-1, -1,  6,  9],
      [-5,  5, -3,  6],
      [ 7, -3,  5, -6],
      [ 3, -3, -2,  3]]

B1 = [-29, -54, 38, 41]
```

In [67]:
```
sol_gauss(A1, B1)
```

Out[67]:
```
[4.0, -8.0, -4.0, -1.0]
```

In [68]:
```
A2 = [
      [0, 2, 3],
      [4, 6, 7],
      [2, 1, 6],
                ]

B2 = [8, -3, 5]
```

In [69]:
```
sol_gauss(A2, B2)
```

```
---------------------------------------------------------------------
-----
ZeroDivisionError                         Traceback (most recent call
last)
<ipython-input-69-db4e70557cae> in <module>
----> 1 sol_gauss(A2, B2)

<ipython-input-64-5cda02fd7f0f> in sol_gauss(A, B)
     17         S1 = [f[:] for f in S]
     18         for i in range(k+1, m):  # Operaciones: # fi = fi + (-
```

In [70]:
```
A4 = [[4, 6, 7],
      [0, 2, 3],
      [2, 1, 6],
                ]

B4 = [-3, 8, 5]

sol_gauss(A4, B4)
```

Out[70]:
```
[-5.431818181818182, 0.04545454545454586, 2.6363636363636362]
```

In [71]:
```
A3 = [
      [1, 2, 3],
      [4, 6, 7],
      [1, 2, 3],
                ]

B3 = [8, -3, 5]
```

In [72]:
```
sol_gauss(A3, B3)
```

```
---------------------------------------------------------------------
-----
ZeroDivisionError                         Traceback (most recent call
last)
<ipython-input-72-8c066bd2a903> in <module>
----> 1 sol_gauss(A3, B3)

<ipython-input-64-5cda02fd7f0f> in sol_gauss(A, B)
     25
     26     for i in range(1, m+1):
---> 27         sol[m-i] +=S[m-i][m] / S[m-i][m-i]
     28         for j in range(1, i):
     29             sol[m-i] += -S[m-i][m-j]*sol[m-j]/ S[m-i][m-i]

ZeroDivisionError: float division by zero
```

Método de Gauss Jordan

Es una variación de la eliminación de Gauss simple en el que en vez de eliminar la incógnica únicamente en las filas inferiores, se hace en todas las demás, tanto superiores como inferiores.

Adicionalmente, todas las filas se normalizan al dividirlas entre su propio pivote.

Se busca llegar a una matriz de este tipo, a partir de operaciones entre filas de la matriz aumentada:

$$\underline{\underline{S}} = \begin{bmatrix} 1 & 0 & 0 & 0 & r_{15} \\ 0 & 1 & 0 & 0 & r_{25} \\ 0 & 0 & 1 & 0 & r_{35} \\ 0 & 0 & 0 & 1 & r_{45} \end{bmatrix}$$

Si se extrae la matriz $\underline{\underline{A}}$ de coeficientes constantes, resulta ser la matriz identidad de orden n. Y si se extrae el vector B de constantes, se obtiene el vector de soluciones del sistema.

Construcción del método a partir del módulo numpy

Resulta bastante sencillo, a partir del desarrollo lógico hecho anteriormente para la eliminación de gauss simple, obtener un desarrollo del método de Gauss-Jordan.

Para efectos prácticos y de aprendizaje del módulo `numpy`, se obtendrá la solución únicamente usando operaciones básicas, ya sean matriciales o no matriciales.

In [73]:
```
import numpy as np
```

Por ejemplo, si se tiene el siguiente sistema de ecuaciones:

$$3x_1 - 0.1x_2 - 0.2x_3 = 7.85$$

$$0.1x_1 + 7x_2 - 0.3x_3 = -19.3$$

$$0.3x_1 - 0.2x_2 + 10x_3 = 71.4$$

Se organiza la información primero en listas, así:

In [74]:
```
A = [
     [  3, -0.1, -0.2],
     [0.1,    7, -0.3],
     [0.3, -0.2,   10],
                      ]

B = [7.85, -19.3, 71.4]
```

In [75]:
```
# Se convierten al formato np.array

A = np.array(A)
B = np.array([B]).T
```

¿Por qué se pone `.T` en la defincón de `B` ? ¿Por qué se pone `B` en la definición inicial con doble lista?

In [76]:
```
# Se verifica A.
A
```

Out[76]:
```
array([[ 3. , -0.1, -0.2],
       [ 0.1,  7. , -0.3],
       [ 0.3, -0.2, 10. ]])
```

In [77]:
```
# Se verifica B
B
```

Out[77]:
```
array([[  7.85],
       [-19.3 ],
       [ 71.4 ]])
```

In [78]:
```
# Se crea la matriz aumentada
S = np.append(A, B, axis=1)  # Se pone 1 porque se agrega una columna, se
```

In [79]:
```
S
```

Out[79]:
```
array([[  3.  ,  -0.1 ,  -0.2 ,   7.85],
       [  0.1 ,   7.  ,  -0.3 , -19.3 ],
       [  0.3 ,  -0.2 ,  10.  ,  71.4 ]])
```

In [80]:
```
# Se define el numero de filas, o la cantidad de soluciones como:
m = len(S)  # También puede ser S.shape[0]
```

In [81]:
```
len(S)
```

Out[81]:
```
3
```

Recordando la expresión simbólica de la matriz aumentada:

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

Se normaliza la primera fila con el primer pivote de acuerdo a la siguiente operación entre filas:

$$f_1 = (\frac{1}{a_{11}})f_1$$

In [82]:
```
pivote = S[0, 0]  # Se define el primer pivote.
```

In [83]:
```
# Se normaliza la primera fila con el primer pivote.
# En términos de operaciones con filas f1 = (1/a11)*f1
S[0, :] = S[0, :]/pivote
```

In [84]:
```
S
```

Out[84]:
```
array([[ 1.00000000e+00, -3.33333333e-02, -6.66666667e-02,
         2.61666667e+00],
       [ 1.00000000e-01,  7.00000000e+00, -3.00000000e-01,
        -1.93000000e+01],
       [ 3.00000000e-01, -2.00000000e-01,  1.00000000e+01,
         7.14000000e+01]])
```

Para eliminar la primera incógnita de la segunda y la tercera ecuación, al ya tenerse normalizada la primera fila, solo basta con hacer esta operación:

Se realizan dos pasos (m-1) para eliminar la primera incógnita de la segunda y la tercera fila realizando esta operación:

$f_i = f_i + (-a_{i1})f_1$ con $i = 2, 3$ o en Python `[1, 2]`

In [85]:
```
S[1, :] = S[1, :] - S[1, 0]*S[0, :]  # f2 = f2 - (a21)*f1
```

In [86]:
```
S
```

Out[86]:
```
array([[ 1.00000000e+00, -3.33333333e-02, -6.66666667e-02,
         2.61666667e+00],
       [ 0.00000000e+00,  7.00333333e+00, -2.93333333e-01,
        -1.95616667e+01],
       [ 3.00000000e-01, -2.00000000e-01,  1.00000000e+01,
         7.14000000e+01]])
```

In [87]:
```
# Fácilmente, se hace lo mismo con la tercera fila.
S[2, :] = S[2, :] - S[2, 0]*S[0, :]  # f3 = f3 - (a31)*f1
```

In [88]:
```
S
```

Out[88]:
```
array([[ 1.00000000e+00, -3.33333333e-02, -6.66666667e-02,
         2.61666667e+00],
       [ 0.00000000e+00,  7.00333333e+00, -2.93333333e-01,
        -1.95616667e+01],
       [ 0.00000000e+00, -1.90000000e-01,  1.00200000e+01,
         7.06150000e+01]])
```

Estas dos operaciones se pueden organizar de manera general en un ciclo:

```
S[1, :] = S[1, :] - S[1, 0]*S[0, :]  # f2 = f2 - (a21)*f1
S[2, :] = S[2, :] - S[2, 0]*S[0, :]  # f3 = f3 - (a31)*f1
```

Se define una variable `i` que haga el recorrido `[1, 2]`, es decir, `range(1, m)` de una manera más general.

```
S[i, :] = S[i, :] - S[i, 0]*S[0, :]  # fi = fi - (ai1)*f1
```

In [89]:
```
for i in [1, 2]:
    S[i, :] = S[i, :] - S[i, 0]*S[0, :]  # fi = fi - (ai1)*f1
```

In [90]:
```
S
```

Out[90]:
```
array([[ 1.00000000e+00, -3.33333333e-02, -6.66666667e-02,
         2.61666667e+00],
       [ 0.00000000e+00,  7.00333333e+00, -2.93333333e-01,
        -1.95616667e+01],
       [ 0.00000000e+00, -1.90000000e-01,  1.00200000e+01,
         7.06150000e+01]])
```

Se continúa con la segunda fila.

In [91]:
```
pivote = S[1, 1]  # Se define el segundo pivote.
```

In [92]:
```
# Se normaliza la segunda fila con el segundo pivote.
# En términos de operaciones con filas f2 = (1/a22)*f2
S[1, :] = S[1, :]/pivote
```

In [93]:
```
S
```

Out[93]:
```
array([[ 1.00000000e+00, -3.33333333e-02, -6.66666667e-02,
         2.61666667e+00],
       [ 0.00000000e+00,  1.00000000e+00, -4.18848168e-02,
        -2.79319372e+00],
       [ 0.00000000e+00, -1.90000000e-01,  1.00200000e+01,
         7.06150000e+01]])
```

Aquí aparece la variación con el método de eliminación de Gauss. Pues se elimina la segunda incógnita no solo de la tercera ecuación, sino de la primera. Luego, al ya tenerse normalizada la segunda fila, solo basta con hacer esta operación:

Se realizan dos pasos (m-1) para eliminar la segunda incógnita de la primera y la tercera fila realizando esta operación:

$f_i = f_i + (-a_{i2})f_1$ con $i = 1, 3$ o en Python `[0, 2]`

In [94]:
```
S[0, :] = S[0, :] - S[0, 1]*S[1, :]  # f1 = f1 - (a12)*f2
S[2, :] = S[2, :] - S[2, 1]*S[1, :]  # f3 = f3 - (a32)*f2
```

In [95]:
```
S
```

Out[95]:
```
array([[ 1.00000000e+00,  0.00000000e+00, -6.80628272e-02,
         2.52356021e+00],
       [ 0.00000000e+00,  1.00000000e+00, -4.18848168e-02,
        -2.79319372e+00],
       [ 0.00000000e+00,  0.00000000e+00,  1.00120419e+01,
         7.00842932e+01]])
```

In [96]:
```
# Fácilmente se puede sintetizar en
for i in [0, 2]:
    S[i, :] = S[i, :] - S[i, 1]*S[1, :]  # fi = fi - (ai2)*f2
```

De forma análoga, para la última columna.

In [97]:
```
pivote = S[2, 2]  # Se define el tercer pivote.

# Se normaliza la tercera fila con el tercer pivote.
# En términos de operaciones con filas f3 = (1/a33)*f3
S[2, :] = S[2, :]/pivote
```

In [98]:
```
S
```

Out[98]:
```
array([[ 1.        ,  0.        , -0.06806283,  2.52356021],
       [ 0.        ,  1.        , -0.04188482, -2.79319372],
       [ 0.        ,  0.        ,  1.        ,  7.        ]])
```

In [99]:
```
# Se realiza la eliminación la tercera incógnita de la primera y la segun
for i in [0, 1]:
    S[i, :] = S[i, :] - S[i, 2]*S[2, :]  # fi = fi - (ai3)*f3
```

In [100]:
```
S
```

Out[100]:
```
array([[ 1. ,  0. ,  0. ,  3. ],
       [ 0. ,  1. ,  0. , -2.5],
       [ 0. ,  0. ,  1. ,  7. ]])
```

Teniendo la solución, se comparan los procesos para eliminar incognitas de las tres columnas:

```
pivote = S[0, 0]  # Se define el primer pivote.
S[0, :] = S[0, :]/pivote  # Normalización con f1 = (1/a11)*f1
for i in [1, 2]:
    S[i, :] = S[i, :] - S[i, 0]*S[0, :]  # fi = fi - (ai1)*f1

pivote = S[1, 1]  # Se define el segundo pivote.
S[1, :] = S[1, :]/pivote  # Normalización con f2 = (1/a22)*f2
for i in [0, 2]:
```

In [101]:
```
for j in range(m):
    pivote = S[j, j]  # Se define el pivote j.
    S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
    for i in range(m):
        if i != j:  # Se exluye la operación cuando i=j.
            S[i, :] = S[i, :] - S[i, j]*S[j, :]  # fi = fi - (aij)*fj
```

In [102]:
```
S
```

Out[102]:
```
array([[ 1. ,  0. ,  0. ,  3. ],
       [ 0. ,  1. ,  0. , -2.5],
       [ 0. ,  0. ,  1. ,  7. ]])
```

In [103]:
```
# Se extrae la solución.
sol = S[:, -1]  # La última columna de la matriz aumentada.
```

In [104]:
```
sol
```

Out[104]:
```
array([ 3. , -2.5,  7. ])
```

Se concentran todos los resultados lógicos y de programación en una función.

In [105]:
```
def np_gauss_jordan(A, B):
    '''
        Función que utiliza el módulo numpy para hallar la solución del s
        AX=B, en donde:
        A: coeficientes constantes, se ingresa como una lista de listas.
        X: incógnitas, sol.
        B: constantes, se ingresa como una lista.
        La solución se obtiene con la técnica Gauss-Jordan.
    '''
    import numpy as np
    A = np.array(A)
    B = np.array([B]).T

    S = np.append(A, B, axis=1)  # Se crea la matriz aumentada.
    m = S.shape[0]               # Número de filas o de soluciones.

    for j in range(m):
        pivote = S[j, j]          # Se define el pivote j, ajj.
        S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*
        for i in range(m):
```

```
            if i != j:            # Se exluye la operación cuando i=j.
                S[i, :] = S[i, :] - S[i, j]*S[j, :]  # Eliminación con fi

    sol = S[:, -1]  # Se extrae la solución de la última columna de S.

    return list(sol)
```

In [106]:
```
A = [
     [  3, -0.1, -0.2],
     [0.1,    7, -0.3],
     [0.3, -0.2,   10],
                      ]

B = [7.85, -19.3, 71.4]
```

In [107]:
```
np_gauss_jordan(A, B)
```

Out[107]:
```
[3.0, -2.5, 7.000000000000002]
```

Problemas que pueden ocurrir:

- Sistema singular
- Sistema con algún cero en la diagonal

Soluciones:

- Pivoteo parcial
- Reorganización de filas

Nuevos comandos

- np.append()