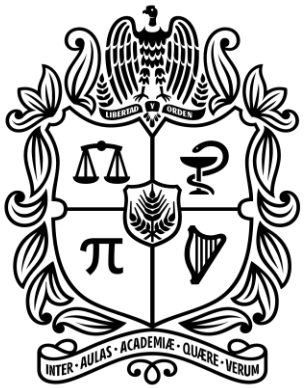
Métodos Numéricos Aplicados a la Ingeniería Civil

1.3-Gauss y Gauss-Jordan



UNIVERSIDAD NACIONAL DE COLOMBIA

Juan Nicolás Ramírez Giraldo
jnramirezg@unal.edu.co

Departamento de Ingeniería Civil
Facultad de Ingeniería y Arquitectura
Universidad Nacional de Colombia
Sede Manizales

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

Mejoramiento de las soluciones de Gauss y Gauss-Jordan

Hasta ahora, la obtención de la solución mediante los métodos de eliminación de Gauss y Gauss-Jordan sistematiza la obtención de las soluciones a partir de la matriz aumentada original, tomando los elementos de la diagonal principal sin consideración alguna de los problemas que pueden causarse.

Veamos el programa desarrollado usando en módulo Numpy para el método de eliminación de Gauss-Jordan.

Ver: 07-gauss_jordan_v1.py

Mejoramiento de las soluciones de Gauss y Gauss-Jordan

```python
def np_gauss_jordan(A, B):
    '''
        Función que utiliza el módulo numpy para hallar la solución del sistema
        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)*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, :]  # Eliminación con fi = fi - (aij)*fj

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

    return list(sol)
```

Mejoramiento de las soluciones de Gauss y Gauss-Jordan

En particular, revisemos los ciclos con las operaciones principales:

```
for j in range(m):
    pivote = S[j, j]
    S[j, :] = S[j, :]/pivote
    for i in range(m):
        if i != j:
            S[i, :] = S[i, :] - S[i, j]*S[j, :]
```

Se evidencia que el elemento pivote es el único denominador en las operaciones dentro de los ciclos, por lo que la fuente de errores está asociada a los casos donde este elemento se vuelve cero (o tiende a cero).

Caso 1: matriz singular

Como vimos en el método de solución mediante la regla de Cramer, hay dos formas en las que el sistema de **ecuaciones es singular**. Una ocurre porque hay infinitas soluciones y la otra porque no hay ninguna solución.

En un **sistema con infinitas soluciones**:

Si la matriz aumentada del sistema original es:

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 1 \\ 2 & 4 & 2 & 2 \\ 1 & -1 & 1 & 1 \end{array}\right]$$

Para eliminar el primer elemento de la tercera fila, solo es necesario hacer:

$$f_3 = f_3 + (-1).\, f_1$$

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 1 \\ 2 & 4 & 2 & 2 \\ 0 & 0 & 0 & 0 \end{array}\right]$$

Caso 1: matriz singular

En un **sistema con infinitas soluciones**:

Se cumplió el objetivo de eliminar el primer elemento de la tercera fila, pero al revisar la tercera ecuación:

$$(0).\, x_1 + (0).\, x_2 + (0).\, x_3 = 0$$

$$0 = 0$$

Lo que implica que no hay ecuaciones suficientes para resolver el sistema, es decir, **infinitas soluciones**.

Caso 1: matriz singular

En un **sistema con infinitas soluciones**:

[3]:
```
A = [[ 1, -1,  1],
     [ 2,  4,  2],
     [ 1, -1,  1]]

B = [1, 2, 1]

np_gauss_jordan(A, B)
```

```
<ipython-input-2-ca6358b30d55>:21: RuntimeWarning: invalid value encountered in
true_divide
  S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
```

[3]:
```
[nan, nan, nan]
```

Caso 1: matriz singular

En un **sistema sin solución**:

Si la matriz aumentada del sistema original es:

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 1 \\ -2 & 2 & -2 & 8 \\ 2 & 4 & 2 & 2 \end{array}\right]$$

Para eliminar el primer elemento de la segunda fila, solo es necesario hacer:

$$f_2 = f_2 + (2).\, f_1$$

$$\left[\begin{array}{ccc|c} 1 & -1 & 1 & 1 \\ 0 & 0 & 0 & 10 \\ 2 & 4 & 2 & 2 \end{array}\right]$$

Caso 1: matriz singular

En un **sistema sin solución**:

Se cumplió el objetivo de eliminar el primer elemento de la segunda fila, pero al revisar la segunda ecuación:

$$(0).\, x_1 + (0).\, x_2 + (0).\, x_3 = 10$$

$$0 = 10$$

Una de las ecuaciones se convirtió en una inconsistencia matemática, es decir, **no hay soluciones**.

Caso 1: matriz singular

En un **sistema sin solución**:

[4]:
```
A = [[ 1, -1,  1],
     [-2,  2, -2],
     [ 2,  4,  2]]

B = [1, 2, 1]

np_gauss_jordan(A, B)
```

```
<ipython-input-2-ca6358b30d55>:21: RuntimeWarning: divide by zero encountered in
true_divide
  S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
<ipython-input-2-ca6358b30d55>:21: RuntimeWarning: invalid value encountered in
true_divide
  S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
```

[4]:
```
[nan, nan, nan]
```

Caso 1: matriz singular

¿Cómo se previene que la función `np_gauss_jordan()` *arroje soluciones erróneas en el caso de una matriz singular?*

Identificando antes del cálculo que involucra el pivote, si la fila correspondiente de la matriz aumentada está llena de ceros, al menos en la parte de coeficientes constantes.

¿Qué significa que una columna se haga cero?

Se cuenta con más ecuaciones que incógnitas, pues una de las incógnitas no toma valores dentro de ninguna ecuación. O incluso, que hay una inconsistencia en el sistema.

Es claro entonces que, el sistema se hace **singular** si alguna fila o columna de $\underline{\underline{A}}$ sea cero.

Ya vimos cómo se llega a tener una fila de ceros y que efectos tiene sobre la solución, ahora veamos qué ocurre cuando se llega a una columna de ceros.

Caso 1: matriz singular

Por ejemplo, el siguiente sistema 3x3:

[5]:
```
A = [[  1, -6,  0],
     [ -4,  2,  0],
     [  7,  2,  0]]

B = [10, 8, 4]

np_gauss_jordan(A, B)
```

```
<ipython-input-2-ca6358b30d55>:21: RuntimeWarning: divide by zero encountered in
true_divide
  S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
<ipython-input-2-ca6358b30d55>:21: RuntimeWarning: invalid value encountered in
true_divide
  S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
<ipython-input-2-ca6358b30d55>:24: RuntimeWarning: invalid value encountered in
multiply
  S[i, :] = S[i, :] - S[i, j]*S[j, :]  # Eliminación con fi = fi - (aij)*fj
```

[5]:
```
[nan, nan, inf]
```

Caso 1: matriz singular

Por ejemplo, el siguiente sistema 3x3:

[5]:
```
A = [[  1, -6,  0],
     [ -4,  2,  0],
     [  7,  2,  0]]

B = [10, 8, 4]
```

Veamos el sistema de ecuaciones asociado:

$$\begin{cases} x_1 - 6x_2 = 10 \\ -4x_1 + 2x_2 = 8 \\ 7x_1 + 2x_2 = 4 \end{cases}$$

Si únicamente hay dos incógnitas y tres ecuaciones, es posible hallar x_1 y x_2 a partir de:

$$\begin{cases} x_1 - 6x_2 = 10 \\ -4x_1 + 2x_2 = 8 \end{cases} \text{ o } \begin{cases} x_1 - 6x_2 = 10 \\ 7x_1 + 2x_2 = 4 \end{cases} \text{ o } \begin{cases} -4x_1 + 2x_2 = 8 \\ 7x_1 + 2x_2 = 4 \end{cases}$$

Es decir, no hay una solución única.

Solución caso 1: matriz singular

El reto es identificar si la fila (en la parque izquierda) o columna del pivote está llena de ceros en la matriz aumentada.

Dada una matriz aumentada:

[9]:
```
S = np.array([[ 1, -1,  1,  1],
              [ 0,  0,  0, 10],
              [ 1,  4,  2,  2]])
```

Solución caso 1: matriz singular

[9]:
```
S = np.array([[ 1, -1,  1,  1],
              [ 0,  0,  0, 10],
              [ 1,  4,  2,  2]])
```

Se construye una estrategia para identificar alguna de las dos condiciones suponiendo que el ciclo principal en términos de `j` va en `j=1`, así:

[10]:
```
m = len(S)  # Tamaño de matriz.

# Exracción del ciclo principal.
j = 1
pivote = S[j, j]

# Verificación fila.
fila_j = S[j, :-1]
ver_ceros_f = np.isclose(fila_j, 0)  # Verificación de ceros.
sum_ceros_f = ver_ceros_f.sum()      # Cantidad de ceros.

# Verificación columna.
columna_j = S[:, j]
ver_ceros_c = np.isclose(columna_j, 0)  # Verificación de ceros.
sum_ceros_c = ver_ceros_c.sum()         # Cantidad de ceros.

if sum_ceros_f == m or sum_ceros_c == m :
    sol = 'Sistema singular.'
```

Solución caso 1: matriz singular

[9]:
```
S = np.array([[ 1, -1,  1,  1],
              [ 0,  0,  0, 10],
              [ 1,  4,  2,  2]])
```

Se construye una estrategia para identificar alguna de las dos condiciones suponiendo que el ciclo principal en términos de `j` va en `j=1`, así:

[11]:
```
S[j, :-1]
```

[11]:
```
array([0, 0, 0])
```

[12]:
```
S[:, j]
```

[12]:
```
array([-1,  0,  4])
```

[13]:
```
np.isclose(fila_j, 0)
```

[13]:
```
array([ True,  True,  True])
```

[14]:
```
ver_ceros_f.sum()
```

[14]:
```
3
```

Ver más sobre `np.close()` aquí

Caso 2: Ceros en la diagonal sin singularidad

Hay un caso en el que durante el proceso de eliminación mediante alguna de las dos técnicas vistas (Gauus o Gauss-Jordan) puede aparecer un cero en la diagonal principal, pero esto no implica que el sistema sea singular. Por eso, no es posible realizar esta operación:

```
for j in range(m):
    pivote = S[j, j]
    S[j, :] = S[j, :]/pivote
```

Por ejemplo, dado este sistema de ecuaciones de 3x3:

$$\begin{cases} x_2 + x_3 = 0 \\ x_1 + 2x_2 + x_3 = 0 \\ 2x_1 - x_2 + x_3 = 4 \end{cases}$$

Si se resuelve con una técnica a mano, su solución es: $x_1 = 1$, $x_2 = -1$ y $x_3 = 1$.

Luego, si se soluciona mediante la función `np_gauss_jordan()`, este fallará porque la matriz aumentada en el paso `j=0` tendrá un denominador 0.

Caso 2: Ceros en la diagonal sin singularidad

Si se resuelve con una técnica a mano, su solución es: $x_1 = 1$, $x_2 = -1$ y $x_3 = 1$.

[15]:
```
A = [[0,  1, 1],
     [1,  2, 1],
     [2, -1, 1]]

B = [0, 0, 4]

np_gauss_jordan(A, B)
```

```
<ipython-input-2-ca6358b30d55>:21: RuntimeWarning: divide by zero encountered in tru
e_divide
  S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
<ipython-input-2-ca6358b30d55>:21: RuntimeWarning: invalid value encountered in true
_divide
  S[j, :] = S[j, :]/pivote  # Normalización con filas fj = (1/ajj)*fj
```

[15]:
```
[nan, nan, nan]
```

Caso 2: Ceros en la diagonal sin singularidad

Si se resuelve con una técnica a mano, su solución es: $x_1 = 1$, $x_2 = -1$ y $x_3 = 1$.

Y si se cambian la primera y la segunda fila de posición tanto en $\underline{\underline{A}}$ como en $\underline{B}$:

[16]:
```
A = [[1,  2, 1],
     [0,  1, 1],
     [2, -1, 1]]

B = [0, 0, 4]

np_gauss_jordan(A, B)
```

[16]:
```
[1, -1, 1]
```

Nos da la solución $x_1 = 1$, $x_2 = -1$ y $x_3 = 1$, es decir, la verdadera.

Caso 3: Valores muy cercanos a cero en la diagonal

Resultaba muy problemático para casos en los que al realizar operaciones, el redondeo jugaba un papel muy importante, sin embargo, para el caso de Python se presentará en casos muy críticos, aún así se debe controlar.

Veamos el caso de los siguientes cuatro sistemas de ecuaciones de 2x2, que por construcción se sabe que su respuesta exacta es: $x_1 = \frac{1}{3}$ y $x_2 = \frac{2}{3}$.

A medida de que se agregan ceros al primer coeficiente de la primera ecuación y ceros a la constante, observemos qué ocurre:

Caso 3: Valores muy cercanos a cero en la diagonal

Veamos el caso de los siguientes cuatro sistemas de ecuaciones de 2x2, que por construcción se sabe que su respuesta exacta es: $x_1 = \frac{1}{3}$ y $x_2 = \frac{2}{3}$.

[17]:
```
A1 = [[0.00000003, 3.0000],[1.0000, 1.0000]]
B1 = [ 2.00000001, 1.0000]
np_gauss_jordan(A1, B1)
```

[17]:
```
[0.3333333358168602, 0.6666666666666666]
```

[18]:
```
A2 = [[0.00000000000003, 3.0000],[1.0000, 1.0000]]
B2 = [ 2.00000000000001, 1.0000]
np_gauss_jordan(A2, B2)
```

[18]:
```
[0.3359375, 0.6666666666666667]
```

[19]:
```
A3 = [[0.000000000000000000000003, 3.0000],[1.0000, 1.0000]]
B3 = [ 2.000000000000000000000001, 1.0000]
np_gauss_jordan(A3, B3)
```

[19]:
```
[0.0, 0.6666666666666667]
```

[20]:
```
A4 = [[0.000000000000000000000003, 3.0000],[1.0000, 1.0000]]
B4 = [ 2.000000000000000000000001, 1.0000]
np_gauss_jordan(A4, B4)
```

[20]:
```
[-8589934592.0, 0.6666666666666667]
```

Caso 3: Valores muy cercanos a cero en la diagonal

Las soluciones teóricas son $x_1 = \frac{1}{3}$ y $x_2 = \frac{2}{3}$, pero nos dio esto:

Sistema 1: `[0.3333333358168602, 0.6666666666666666]`

Sistema 2: `[0.3359375000000000, 0.6666666666666667]`

Sistema 3: `[0.0000000000000000, 0.6666666666666667]`

Sistema 4: `[-8589934592.000000, 0.6666666666666667]`

Lo cual se debe a que entre más parecido sea el primer pivote a cero, Python perderá capacidad de almacenamiento de cifras decimales, y así, como pasamos de una respuesta significativamente buena en el sistema 1, a una respuesta nula en el sistema 3 y una respuesta que busca asemejarse a `-inf` en el sistema 4.

Caso 3: Valores muy cercanos a cero en la diagonal

Busquemos alguna estrategia para evitar que el primer pivote sea cero, por ejemplo, en sistema 4. Una opción es cambiar la posición de las filas tanto en $\underline{\underline{A}}$ como en $\underline{B}$:

[21]:
```
A5 = [[1.0000, 1.0000],[0.00000000000000000000000003, 3.0000]]
B5 = [ 1.0000,2.000000000000000000000001]
np_gauss_jordan(A5, B5)
```

[21]:
```
[0.33333333333333337, 0.6666666666666666]
```

Lo cual nos lleva a una respuesta muy buena, muy cercana a la exacta. Observemos que el primer pivote es `1.0000`, es decir, la división no genera ningún problema.

Las soluciones teóricas son $x_1 = \frac{1}{3}$ y $x_2 = \frac{2}{3}$

Solución caso 2 y caso 3: pivoteo parcial

La técnica consiste en poner en la posición del **pivote** mediante operaciones entre filas el elemento mayor absoluto con respecto al resto de los demás elementos de la correspondiente fila.

Por ejemplo, para una matriz ampliada de un sistema 4x4, después de ser reducida su primera columna, quedó:

$$\left[\begin{array}{cccc|c} 1 & -1 & 1 & 2 & 1 \\ 0 & 1 & -2 & -1 & 2 \\ 0 & -1 & 7 & 3 & 1 \\ 0 & 9 & -5 & 2 & -1 \end{array}\right]$$

Solución caso 2 y caso 3: pivoteo parcial

$$\left[\begin{array}{cccc|c} 1 & -1 & 1 & 2 & 1 \\ 0 & 1 & -2 & -1 & 2 \\ 0 & -1 & 7 & 3 & 1 \\ 0 & 9 & -5 & 2 & -1 \end{array}\right]$$

Los máximos de cada fila son:

Fila 2: $max_{f_2} = 2$

Fila 3: $max_{f_3} = 7$

Fila 4: $max_{f_4} = 9$

Luego, se revisa la división entre cada elemento de la columna 2 respecto al máximo de su fila correspondiente:

$$\left|\frac{a_{22}}{max_{f_2}}\right| = \left|\frac{1}{2}\right| = 0.50$$

$$\left|\frac{a_{23}}{max_{f_3}}\right| = \left|\frac{-1}{7}\right| = 0.14$$

$$\left|\frac{a_{24}}{max_{f_4}}\right| = \left|\frac{9}{9}\right| = 1.00$$

Solución caso 2 y caso 3: pivoteo parcial

$$\left[\begin{array}{cccc|c} 1 & -1 & 1 & 2 & 1 \\ 0 & 1 & -2 & -1 & 2 \\ 0 & -1 & 7 & 3 & 1 \\ 0 & 9 & -5 & 2 & -1 \end{array}\right]$$

Por lo tanto, es conveniente poner de pivote en la segunda fila, la cuarta fila, es decir, intercambiar de posiciones la fila 2 con la fila 4.

$$\left[\begin{array}{cccc|c} 1 & -1 & 1 & 2 & 1 \\ 0 & 1 & -2 & -1 & 2 \\ 0 & -1 & 7 & 3 & 1 \\ 0 & 9 & -5 & 2 & -1 \end{array}\right]$$

Solución caso 2 y caso 3: pivoteo parcial

$$\left[\begin{array}{cccc|c} 1 & -1 & 1 & 2 & 1 \\ 0 & 1 & -2 & -1 & 2 \\ 0 & -1 & 7 & 3 & 1 \\ 0 & 9 & -5 & 2 & -1 \end{array}\right]$$

Esto se puede poner dentro de una rutina en la que `j` indique el indice del elemento de diagonal de referencia.

En el caso anterior, se tiene que `j=1` y que la matriz es:

[22]:
```
S = np.array([[1, -1,  1,  2,  1],
              [0,  1, -2, -1,  2],
              [0, -1,  7,  3,  1],
              [0,  9, -5,  2, -1]])

j = 1
```

Solución caso 2 y caso 3: pivoteo parcial

[22]:
```
S = np.array([[1, -1,  1,  2,  1],
              [0,  1, -2, -1,  2],
              [0, -1,  7,  3,  1],
              [0,  9, -5,  2, -1]])

j = 1
```

[23]:
```
### Pivoteo parcial ###
R = S[j:, j:]  # Se extrae la matriz con primer elemento el pivote.
max_fil = abs(R[:,:-1]).max(axis=1)  # Máximo de cada fila.
col_j = abs(R[:, 0])                 # Elementos bajo el pivote.
# Máxima relación. Se pone 1e-14 para evitar división entre cero.
div = col_j/(max_fil + 1e-14)
idx = np.where(div == max(div))[0][0] + j  # Ubicación del máximo.
S1 = np.array([f[:] for f in S])           # Copia de S.
S[j,  :] = S1[idx, :]  # Se ubica la nueva fila pivote.
S[idx,:] = S1[  j, :]  # Se intercambia con la posición de la ant.
```

Solución caso 2 y caso 3: pivoteo parcial

[22]:
```
S = np.array([[1, -1,  1,  2,  1],
              [0,  1, -2, -1,  2],
              [0, -1,  7,  3,  1],
              [0,  9, -5,  2, -1]])

j = 1
```

[24]:
```
S
```

[24]:
```
array([[ 1, -1,  1,  2,  1],
       [ 0,  9, -5,  2, -1],
       [ 0, -1,  7,  3,  1],
       [ 0,  1, -2, -1,  2]])
```

Función final: pivoteo parcial y detección de singularidad

[25]:
```python
def np_gauss_jordan(A, B):
    '''
        Función que utiliza el módulo numpy para hallar la solución del sistema
        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  # Esto es por si se olvida llamar antes.
    A = np.array(A, dtype=float)
    B = np.array([B], dtype=float).T

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

Función final: pivoteo parcial y detección de singularidad

```
    for j in range(m):
        ### Verificación sistema singular ###
        # Verificación fila.
        fila_j = S[j, :-1]
        ver_ceros_f = np.isclose(fila_j, 0)  # Verificación de ceros.
        sum_ceros_f = ver_ceros_f.sum()      # Cantidad de ceros.
        # Verificación columna.
        columna_j = S[:, j]
        ver_ceros_c = np.isclose(columna_j, 0)  # Verificación de ceros.
        sum_ceros_c = ver_ceros_c.sum()         # Cantidad de ceros.

        if sum_ceros_f == m or sum_ceros_c == m :
            sol = 'Sistema singular'
            break
```

Función final: pivoteo parcial y detección de singularidad

```
        else:
            ### Pivoteo parcial ###
            R = S[j:, j:]  # Se extrae la matriz con primer elemento el pivote.
            max_fil = abs(R[:,:-1]).max(axis=1)  # Máximo de cada fila.
            col_j = abs(R[:, 0])                 # Elementos bajo el pivote.
            # Máxima relación. Se pone 1e-14 para evitar división entre cero.
            div = col_j/(max_fil + 1e-14)
            idx = np.where(div == max(div))[0][0] + j  # Ubicación del máximo.
            S1 = np.array([f[:] for f in S])           # Copia de S.
            S[j,  :] = S1[idx, :]  # Se ubica la nueva fila pivote.
            S[idx,:] = S1[  j, :]  # Se intercambia con la posición de la ant.

            ### Pivote y normalización ###
            pivote = S[j, j]          # Se define el pivote j, ajj.
            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, :]  # Eliminación con fi = fi - (aij)*fj

    if sol != 'Sistema singular':
        sol = list(S[:, -1])  # Se extrae la solución de la última columna de S.

    return sol
```

Ver: [08-gauss_jordan_pivote_parcial.py]

Función final: pivoteo parcial y detección de singularidad

Ejemplos de funcionamiento

[26]:
```
A1 = [[ 2,  -1],
      [-1, 0.5]]

B1 = [4, 1]

np_gauss_jordan(A1, B1)
```

[26]:
```
'Sistema singular'
```

[27]:
```
# Ejemplo de pivoteo parcial 2:
A2 = [[ 0, -2,  0,  3],
      [ 0,  1,  0, -2],
      [ 1,  0, -1,  0],
      [ 0,  4,  2,  0]]

B2 = [-3, 1, 1, 2]

np_gauss_jordan(A2, B2)
```

[27]:
```
[-4.0, 3.0, -5.0, 1.0]
```

Ver: 08-gauss_jordan_pivote_parcial.py

Resumen de nuevos comandos usados

- `np.isclose()`
- `A.sum()`
- `np.where()`
- `abs()`
- `A.max(axis=1)`

Referencias

Chapra, S. C., & Canale, R. P. (2015). *Métodos Numéricos para ingenieros* (7.ª ed.). México D.F.: Mc Graw Hill.