

```
JuanFrancisco Pinto
Taller LU

In [3]: load_ext autoreload

In [4]: # ----- logging -----
import logging
from importlib import reload
from datetime import datetime

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s] %(message)s',
    stream=sys.stdout,
    datefmt='%m-%d %H:%M:%S',
)
logging.info(datetime.now())

In [5]: autoreload 2
from sys import reload
from datetime import datetime

In [6]: A = [[1, 3, 4], [2, 1, 3], [4, 2, 1]]

In [7]: import numpy as np

# =====
def gauss_jordan(A: np.ndarray) -> np.ndarray:
    """Realiza la eliminación de Gauss-Jordan

    ## Parameters
    """
    """A": matriz del sistema de ecuaciones lineales. Debe ser de tamaño n-by-n, donde n es el número de incógnitas.

    ## Return
    """
    """A": matriz reducida por filas.

    """
    # --- encontrar pivote
    p = None
    for pi in range(1, m):
        if A[pi, i] == 0:
            # no es número
            continue

        if p is None:
            # first nonzero element
            p = pi
            continue

        if abs(A[pi, i]) < abs(A[p, i]):
            p = pi

    if p is None:
        # no pivot found.
        raise ValueError("No existe solución única.")

    if p != i:
        # swap rows
        logging.debug(f"Interchanging rows {i} y {p}")
        A[i, :] = A[p, :].copy()
        A[p, :] = A[i, :].copy()
        A[i, i] = A[i, i].copy()
        A[p, i] = A[i, i].copy()

    # --- eliminación: loop por fila
    for j in range(1, n):
        # Eliminar los elementos por debajo del pivote
        for i in range(j+1, n):
            continue # skip pivot row

            m = A[i, i] / A[p, i]
            A[i, i:] = A[i, i:] - m * A[p, i:]

            # dividir para la diagonal
            A[i, i] = A[i, i] / A[i, i]

            logging.info(f"ui(A)")

            if A[i, i] - 1, n - 1] == 0:
                raise ValueError("No existe solución única.")

            print(f"ui(A)")

            # --- Sustitución hacia atrás
            # solución = np.zeros(n)
            # solución[n - 1] = A[n - 1, n] / A[n - 1, n - 1]

            # for i in range(n - 2, -1, -1):
            #     for j in range(i + 1, n):
            #         solución[i] = (A[i, j] - suma) / A[i, i]

            return A

In [8]: from pprint import pprint
pprint(A)
gauss_jordan(A)

[[1, 3, 4], [2, 1, 3], [4, 2, 1]]
[[0, -5, -5], [0, -10, -15], [0, 0, 1]]
[[1, 0, 1], [-5, 1, 1], [0, 0, 1]]
[[0, -5, -5], [0, -10, -15], [0, 0, 1]]
Out[8]: array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]])

In [9]: # Power matrix identidad a la derecha
n = len(A)
A_aug = np.hstack((A, np.eye(n)))
A_aug

Out[9]: array([[1., 3., 4., 1., 0., 0.], [2., 1., 3., 0., 1., 0.], [4., 2., 1., 0., 0., 1.]])

In [10]:

In [10]: np.vstack((A, np.eye(n)))

Out[10]: array([[1., 3., 4., 1., 0., 0.], [2., 1., 3., 0., 1., 0.], [4., 2., 1., 0., 0., 1.]])

In [11]: _M_inv = gauss_jordan(A_aug)
_M_inv

[[0, 1, 3, 4, 1, 0, 0]
[0, -5, -5, -5, -1, 0, 1]
[0, -10, -15, -4, 0, 1, 1]]
[[0, 1, 0, 0.4, -0.2, 0, 1]
[0, 0, -5, 0, -2, 1, 1]
[0, 0, 0, 0.4, -0.6, 0.2, 1]]
Out[11]: array([[1., 0., 0., -0.2, 0.2, 0.2], [-5., 1., 0., 0.4, -0.6, 0.2], [-0., -0., 1., -0., 0.4, -0.2]])

In [12]: _M_inv[:, n:]

Out[12]: array([-0.2, 0.2, 0.2], [0.4, -0.6, 0.2], [-0., 0.4, -0.2])

Comprobando respuesta

In [13]: np.linalg.linalg(np.array(A))

Out[13]: array([-0.2, 0.2, 0.2], [0.4, -0.6, 0.2], [-0., 0.4, -0.2])

Calculo de la matriz inversa del resto de literales

In [33]: A = [[1,3,4],[2,1,3],[4,2,1]]

# Power matrix identidad a la derecha
n = len(A)
A_aug = np.hstack((A, np.eye(n)))
A_aug

# implementamos gauss jordan
_M_inv = gauss_jordan(A_aug)
_M_inv

# mostramos la inversa
_M_inv[:, n:]

[[0, 1, 3, 4, 1, 0, 0]
[0, -5, -5, -5, -1, 0, 1]
[0, -10, -15, -4, 0, 1, 1]]
[[0, 1, 0, 0.4, -0.2, 0, 1]
[0, 0, -5, 0, -2, 1, 1]
[0, 0, 0, 0.4, -0.6, 0.2, 1]]
Out[33]: array([-0.2, 0.2, 0.2], [0.4, -0.6, 0.2], [-0., 0.4, -0.2])

In [30]: A = [[1,2,3],[0,1,4],[5,6,0]]

# Power matrix identidad a la derecha
n = len(A)
A_aug = np.hstack((A, np.eye(n)))
A_aug

# implementamos gauss jordan
_M_inv = gauss_jordan(A_aug)
_M_inv

# mostramos la inversa
_M_inv[:, n:]

[[0, 1, 2, 3, 1, 0, 0]
[0, 1, 4, 0, 1, 1, 0]
[0, -4, -15, -5, 0, 1, 1]]
[[0, 1, 0, -5, 1, -2, 0]
[0, 1, 4, 0, 1, 1, 0]
[0, 0, 1, -5, 4, 1, 1]]
[[0, 1, 0, -5, 1, -2, 0]
[0, 1, 0, 20, -15, -4]
[0, 0, 1, -5, 4, 1, 1]]
Out[30]: array([-24., 18., 5.], [20., -15., -4.], [-5., 4., 1.])

In [31]: A = [[4,2,1],[2,1,3],[1,3,4]]

# Power matrix identidad a la derecha
n = len(A)
A_aug = np.hstack((A, np.eye(n)))
A_aug

# implementamos gauss jordan
_M_inv = gauss_jordan(A_aug)
_M_inv

# mostramos la inversa
_M_inv[:, n:]

[[0, 1, 3, 4, 0, 0, 1]
[0, -5, -5, 0, 1, -2, 1]
[0, -10, -15, 0, -4, 1]]
[[0, 1, 0, 0.6, -0.2]
[0, 1, 0, -2, -0.4]
[0, 0, -5, 1, -2, 0]]
[[0, 1, 0, 0.2, 0.2, -0.2]
[0, 1, 0, 0.2, -0.6, 0.4]
[0, 0, 1, -0.2, 0.4, -0.1]]
Out[31]: array([[0.2, 0.2, -0.2], [0.2, -0.6, 0.4], [-0.2, 0.4, -0.1]])

In [32]: A = [[2,4,6,1],[4,7,5,-6],[2,5,18,10],[6,12,38,16]]

# Power matrix identidad a la derecha
n = len(A)
A_aug = np.hstack((A, np.eye(n)))
A_aug

# implementamos gauss jordan
_M_inv = gauss_jordan(A_aug)
_M_inv

# mostramos la inversa
_M_inv[:, n:]

[[0, 1, 2, 3, 4, 0, 0, 0]
[0, -1, -7, -8, -2, 1, 0, 0]
[0, 1, 12, 9, -1, 0, 1, 0]
[0, 0, 20, 13, -3, 0, 0, 1]]
[[0, 1, 0, -15, -3.5, 2, 0, 0]
[0, 0, 1, 7, 8, 2, -1, -0, -0]
[0, 0, 0, 2, -3, 1, 1, 0]
[0, 0, 0, 20, 13, -3, 0, 0, 1]]
[[0, 1, 0, 0, -13.3, -10.1, 4.2, 2.2, 0, 1]
[0, 1, 0, 6.6, 6.2, -2.4, -1.4, -0, 1]
[0, 0, 1, 0.2, 0.4, 0.2, 0.2, 0, 1]
[0, 0, 0, 0, 9, 9, -4, -4, 1, 1]]
[[0, 1, 0, 0, 0, 0, 0, 0, 3.2, -1.7111111]
[0, 1, 0, 0, 0, 0, 0, 0, -0.4, 0.5333333]
[0, 0, 1, 0, 0, 0, 0, 0, -0.8, 0.2888889]
[0, 0, 0, 1, 0, 0, 0, 0, 1, 0.4444444]]
Out[32]: array([[3.2, -1.7111111, -3.7111111, 1.4777778], [-0.4, 0.5333333, 1.5333333, -0.7333333], [-0.8, 0.2888889, 0.2888889, -0.2222222], [1., 0.4444444, -0.4444444, 0.1111111]])

Calcule la descomposición LU para estas matrices y encuentre la solución para estos vectores de valores independientes b

In [40]: import numpy as np

def lu_factor(A):
    """
    Factoriza con pivoteo parcial:
    P * A = L * U
    Retorna P, L, U

    A = np.array(A, dtype=float, copy=True)
    n = A.shape[0]
    if A.shape[0] != A.shape[1]:
        raise ValueError("A debe ser cuadrada (n*n).")

    P = np.eye(n)
    L = np.eye(n)
    U = A.copy()

    for k in range(n):
        # pivoteo parcial
        p = k + np.argmax(np.abs(U[k, :]))
        if abs(U[p, k]) < 1e-15:
            raise ValueError("No existe solución única (pivote = 0).")

        if p != k:
            U[k, p:] = U[p, k:]
            P[k, p] = P[p, k]
            if k > 0:
                L[p, k] = L[p, k] - L[k, k] * U[k, k]

        for i in range(k + 1, n):
            L[i, k] = U[i, k] / U[k, k]
            U[i, k:] = U[i, k:] - L[i, k] * U[k, k:]

    return P, L, U

def forward_substitution(L, b):
    """Resuelve L * y = b (L triangular inferior)."""
    n = L.shape[0]
    y = np.zeros(n, dtype=float)
    for i in range(n):
        y[i] = (b[i] - np.dot(L[i, 1:], y[1:])) / L[i, i]
    return y

def back_substitution(U, y):
    """Resuelve U * x = y (U triangular superior)."""
    n = U.shape[0]
    x = np.zeros(n, dtype=float)
    for i in range(n - 1, -1, -1):
        s = np.dot(U[i, i+1:], x[i+1:])
        if abs(U[i, i]) < 1e-15:
            raise ValueError("No tenemos solución única (pivote = 0 en U).")
        x[i] = (y[i] - s) / U[i, i]
    return x

def lu_solve(A, b, verbose=True):
    """
    Resuelve Ax=b usando LU:
    PA=LU, luego Ly=Pb y Ux=y
    Retorna x como (n,1)

    A = np.array(A, dtype=float, copy=True)
    b = np.array(b, dtype=float, copy=True)
    if b.ndim == 2 and b.shape[1] == 1:
        b = b[:, 0]
    if b.ndim != 1:
        raise ValueError("b debe ser (n,1) o (n,1).")

    P, L, U = lu_factor(A)
    pb = P @ b
    y = forward_substitution(L, pb)
    x = back_substitution(U, y).reshape(-1, 1)

    if verbose:
        with np.printoptions(precision=6, suppress=True):
            print("P=", P)
            print("L=", L)
            print("U=", U)
            print("P=Ux", x)
            print("Verificamos Ax=Ux", A @ x)

    return x

In [41]: A = [[1, 3, 4], [2, 1, 3], [4, 2, 1]]
b = [[1], [2], [4]]
x = lu_solve(A, b, verbose=True)

P=
[[0, 0, 1]
[1, 0, 0]
[0, 1, 0]]
L=
[[1, 0, 0]
[0, 1, 0]
[0, 0, 1]]
U=
[[4, 2, 1]
[0, 2, 0.5]]
x=
[[1]
[0]
[2]]
Verificamos Ax=
[[1]
[2]
[4]]

In [42]: A = [[1, 2, 3], [0, 1, 4], [5, 6, 0]]
b = [[3], [-5], [2]]
x = lu_solve(A, b, verbose=True)

P=
[[0, 0, 1]
[0, 1, 0]
[1, 0, 0]]
L=
[[1, 0, 0]
[0, 1, 0]
[0, 2, 0.5]]
U=
[[5, 6, 0]
[0, 1, 4]
[0, 0, -0.2]]
x=
[[-152.]
[127.]
[-31.]]
Verificamos Ax=
[[3.]
[-5.]
[2.]]

In [43]: A = [[4, 2, 1], [2, 1, 3], [1, 3, 4]]
b = [[7], [9], [-1]]
x = lu_solve(A, b, verbose=True)

P=
[[1, 0, 0]
[0, 1, 0]
[0, 0, 1]]
L=
[[1, 0, 0]
[0, 1, 0]
[0, 0, 1]]
U=
[[4, 2, 1]
[0, 2, 0.5]]
x=
[[1]
[0]
[2]]
Verificamos Ax=
[[7.]
[9.]
[-1.]]

In [44]: A = [[2, 4, 6, 1], [4, 7, 5, -6], [2, 5, 18, 10], [6, 12, 38, 16]]
b = [[1], [2], [4], [5]]
x = lu_solve(A, b, verbose=True)

P=
[[0, 0, 0, 1]
[0, 1, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]]
L=
[[1, 0, 0, 0]
[0, 1, 0, 0]
[0, 0, 1, 0]
[0, 0, 0, 1]]
U=
[[6, 12, 38, 16]
[0, -1, -20.333333, -16.666667]
[0, 0, -15, -12]
[0, 0, 0, 1]]
x=
[[1.4222222]
[-0.6666667]
[-3.5777778]
[0.2888889]]
Verificamos Ax=
```

