

Apéndice D

Guías Explicativas

A continuación, se presentan las guías explicativas elaboradas como recursos de divulgación académica en el marco del presente proyecto. Cada guía detalla la construcción de los códigos en Python para la resolución de todos los casos de estudio por tipología, mediante el Método Matricial de Rigidez y sus modelos equivalentes con OpenSees.

Armaduras

MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON

ARMADURAS: EJERCICIO 1

```
import numpy as np
import math
import matplotlib.pyplot as plt

print("METODO MATRICIAL DE RIGIDEZ PARA ARMADURAS")

# ===== DATOS DE ENTRADA =====
print("\n==== DATOS DE ENTRADA =====")
# Propiedades de Los elementos
E = 2e8 # Módulo de elasticidad [kN/m2]
A = 0.0015 # Área transversal [m2]
L = np.array([4.0, 3.0, 4.0, 3.0, 5.0, 5.0]) # Longitud de Los elementos [m]
a = np.array([0, 90, 180, 270, 36.8699, 323.1301]) # Ángulo de Los elementos [°]
m = 6 # Número de elementos
n = 4 # Número de nodos
GL = 8 # Grados de Libertad (2gL/nodo)

print(f"Módulo de elasticidad: {E:.2e} kN/m2")
print(f"Área trasnversal: {A} m2")
print(f"Longitud de los elementos: {L} m")
print(f"Ángulo de inclinación de los elementos: {a} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")

# ===== COORDENADAS NODOS =====
coordenadas_nodos = np.array([
    [0, 0], # Nodo 1
    [4, 0], # Nodo 2
    [4, 3], # Nodo 3
    [0, 3] # Nodo 4
])

# ===== ELEMENTOS =====
Elementos = np.array([
    [1, 2], # Elemento 1
    [2, 3], # Elemento 2
    [3, 4], # Elemento 3
    [4, 1], # Elemento 4
    [1, 3], # Elemento 5
    [4, 2] # Elemento 6
])

# ===== GRÁFICA DEL SISTEMA ORIGINAL =====
#Crear gráfica
plt.figure(figsize=(8, 5))
plt.title('SISTEMA ORIGINAL', fontsize=14, fontweight='bold')
```

```
# Dibujar elementos originales
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]
    plt.plot([xi, xf], [yi, yf], '--', color='black', linewidth=2)

# Dibujar nodos
for i, (x, y) in enumerate(coordenadas_nodos):
    plt.plot(x, y, 'o', color='orange', markersize=7)

# Dibujar Los apoyos
plt.plot(0, 0-0.15, '^', color="maroon", markersize=20) # Apoyo fijo
plt.plot(4, 0-0.15, 'o', color="maroon", markersize=20) # Apoyo móvil

# Dibujar La fuerza externa
plt.arrow(-1, 3, 0.8, 0, head_width=0.1, head_length=0.2, fc='b', ec='b',
linewidth=4) plt.text(-1, 3.15, '10 kN', fontsize=11, color='blue',
fontweight='bold')

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid('True', alpha=0.3)
plt.axis('equal')
plt.show()

# ===== GRADOS DE LIBERTAD =====
# E1/E2/E3/E4/E5/E6
Nx = np.array([1, 3, 5, 7, 1, 7]) # Dx GL inicial
Ny = np.array([2, 4, 6, 8, 2, 8]) # Dy GL inicial
Fx = np.array([3, 5, 7, 1, 5, 3]) # Dx GL final
Fy = np.array([4, 6, 8, 2, 6, 4]) # Dy GL final

# ===== ENSAMBLE MATRIZ GLOBAL =====
print("\n===== ENSAMBLE MATRIZ GLOBAL =====")

# Crear matriz global
kG = np.zeros((GL, GL))

for i in range(m):
    # Convertir el ángulo a radianes
    theta = math.radians(a[i])

    # Calcular coseno y seno
    c = math.cos(theta)
    s = math.sin(theta)

    Factor = (A * E) / L[i]
```

```
# Componentes de la matriz de rigidez del elemento_i en coordenadas globas
kg_e = np.zeros((4, 4))
kg_e[0, 0] = Factor * c**2
kg_e[0, 1] = Factor * c * s
kg_e[0, 2] = -Factor * c**2
kg_e[0, 3] = -Factor * c * s

kg_e[1, 0] = Factor * c * s
kg_e[1, 1] = Factor * s**2
kg_e[1, 2] = -Factor * c * s
kg_e[1, 3] = -Factor * s**2

kg_e[2, 0] = -Factor * c**2
kg_e[2, 1] = -Factor * c * s
kg_e[2, 2] = Factor * c**2
kg_e[2, 3] = Factor * c * s

kg_e[3, 0] = -Factor * c * s
kg_e[3, 1] = -Factor * s**2
kg_e[3, 2] = Factor * c * s
kg_e[3, 3] = Factor * s**2

# Ensamblaje de matriz global del sistema
GL_elem = [Nx[i]-1, Ny[i]-1, Fx[i]-1, Fy[i]-1]

for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj]

#print(f"\nELEMENTO: {i+1}")
#print(f"Longitud: {L[i]} m, Ángulo: {a[i]}°")
#print(f"Matriz de rigidez global del elemento:\n {np.round(kg_e, 0)}")
#print(f"Matriz de rigidez global del sistema después del elemento {i+1}:\n
{np.round(kG, 0)}")
print(f"Matriz de rigidez global del sistema:\n {np.round(kG, 0)}")

# ===== RESTRICCIONES =====
print("\n==== RESTRICCIONES =====")
# 0=Libre, 1=Restringido
#
# GL: 1 2 3 4 5 6 7 8
restricciones = np.array([1, 1, 0, 1, 0, 0, 0, 0])

for i in range(n):
    rx = "Restringido" if restricciones[i*2]==1 else "Libre"
    ry = "Restringido" if restricciones[i*2+1]==1 else "Libre"
    print(f"Node {i+1}: Dx={restricciones[i*2]} {rx}, Dy={restricciones[i*2+1]}
{ry}")
```

```
# ===== VECTOR FUERZAS EXTERNAS =====
print("\n===== VECTOR FUERZAS EXTERNAS =====")
#      GL: 1  2  3  4  5  6  7  8
F = np.array([0, 0, 0, 0, 0, 0, 10, 0])
print(f"Vector de fuerzas externas: {F} kN")

# ===== REDUCCIÓN DEL SISTEMA =====
print("\n===== REDUCCIÓN DEL SISTEMA =====")
# Grados de libertad activos (Desplazamientos son desconocidos y Las fuerzas conocidas)
GL_activos = np.where(restricciones==0)[0]
n_GL_activos = len(GL_activos)
print(f"\nGrados de libertad activos: {n_GL_activos}")
print(f"Indices: {GL_activos+1}")

# Matriz global reducida (GL activos)
K_reducida = kG[np.ix_(GL_activos, GL_activos)]
F_reducido = F[GL_activos]
print(f"\nMatriz global reducida:\n {np.round(K_reducida, 0)}")
print(f"Vector de fuerzas reducido: {np.round(F_reducido, 0)}")

# ===== SOLUCIÓN DEL SISTEMA =====
print("\n\n===== SOLUCIÓN DEL SISTEMA =====")
# Resolver el sistema de ecuaciones {F} = [kG]{U}
K_reducida_inv = np.linalg.inv(K_reducida)
U_desconocidos = np.matmul(K_reducida_inv, F_reducido)

# Reconstruir vector de desplazamientos
U_totales = np.zeros(GL)
U_totales[GL_activos] = U_desconocidos

print("\n===== DESPLAZAMIENTOS =====")
for i in range(n):
    Ux = U_totales[i*2]
    Uy = U_totales[i*2+1]
    print(f"Node {i+1}: Ux={U_totales[i*2]:.2e}, Uy={U_totales[i*2+1]:.2e}")

print("\n===== REACCIONES =====")
# Calcular las reacciones
Reacciones = np.matmul(kG, U_totales)

for i in range(n):
    rx = Reacciones[i*2]
    ry = Reacciones[i*2+1]

    if abs(rx)>1e-6 or abs(ry)>1e-6:
        print(f"Node {i+1}: Rx = {rx:.3f} kN, Ry = {ry:.3f} kN")
```

```

print("\n===== FUERZAS INTERNAS/AXIALES =====")
# [kL][T]{UG}

F_elem = np.zeros(m)

for i in range(m):
    # Obtener Los desplazamientos del elemento
    Ue = np.array([
        U_totales[Nx[i]-1],
        U_totales[Ny[i]-1],
        U_totales[Fx[i]-1],
        U_totales[Fy[i]-1]
    ])

    # Calcular matriz de rigidez local
    theta = math.radians(a[i])
    c = math.cos(theta)
    s = math.sin(theta)
    Factor = (A * E) / L[i]

    kL = Factor * np.array([[1, -1], [-1, 1]])
    T = np.array([[c, s, 0, 0], [0, 0, c, s]])
    kL_T = np.matmul(kL, T)
    kL_T_Ue = np.matmul(kL_T, Ue)
    Fuerza_axial = kL_T_Ue[1]
    tipo = "Tracción" if Fuerza_axial > 0 else "Compresión" if Fuerza_axial < 0 else ""
    print(f"Elemento {i+1}: {Fuerza_axial:.3f} kN - {tipo}")

# ===== GRÁFICA DEL SISTEMA DEFORMADO =====
FS = 1000 # Factor de escala
plt.figure(figsize=(8, 5))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar elementos originales
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]
    plt.plot([xi, xf], [yi, yf], '--', color='gray', linewidth=2, alpha=0.5)

# Dibujar estructura deformada
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    # Coordenadas originales de los nodos
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]

    # Desplazamientos de los nodos
    desp_i = np.array([U_totales[(nodo_i-1)*2], U_totales[(nodo_i-1)*2+1]])
    desp_f = np.array([U_totales[(nodo_f-1)*2], U_totales[(nodo_f-1)*2+1]])

```

```
# Coordenadas de formadas amplificadas
xi_def = xi + desp_i[0]*FS
yi_def = yi + desp_i[1]*FS
xf_def = xf + desp_f[0]*FS
yf_def = yf + desp_f[1]*FS

plt.plot([xi_def, xf_def], [yi_def, yf_def], '-', color='green', linewidth=2)

# Dibujar los apoyos
plt.plot(0, 0-0.15, '^', color="maroon", markersize=20) # Apoyo fijo
plt.plot(4, 0-0.15, 'o', color="maroon", markersize=20) # Apoyo móvil

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid('True', alpha=0.3)
plt.axis('equal')
plt.show()
```


METODO MATRICIAL DE RIGIDEZ PARA ARMADURAS

===== DATOS DE ENTRADA =====

Módulo de elasticidad: 2.00e+08 kN/m2

Área transversal: 0.0015 m²

Longitud de los elementos: [4. 3. 4. 3. 5. 5.] m

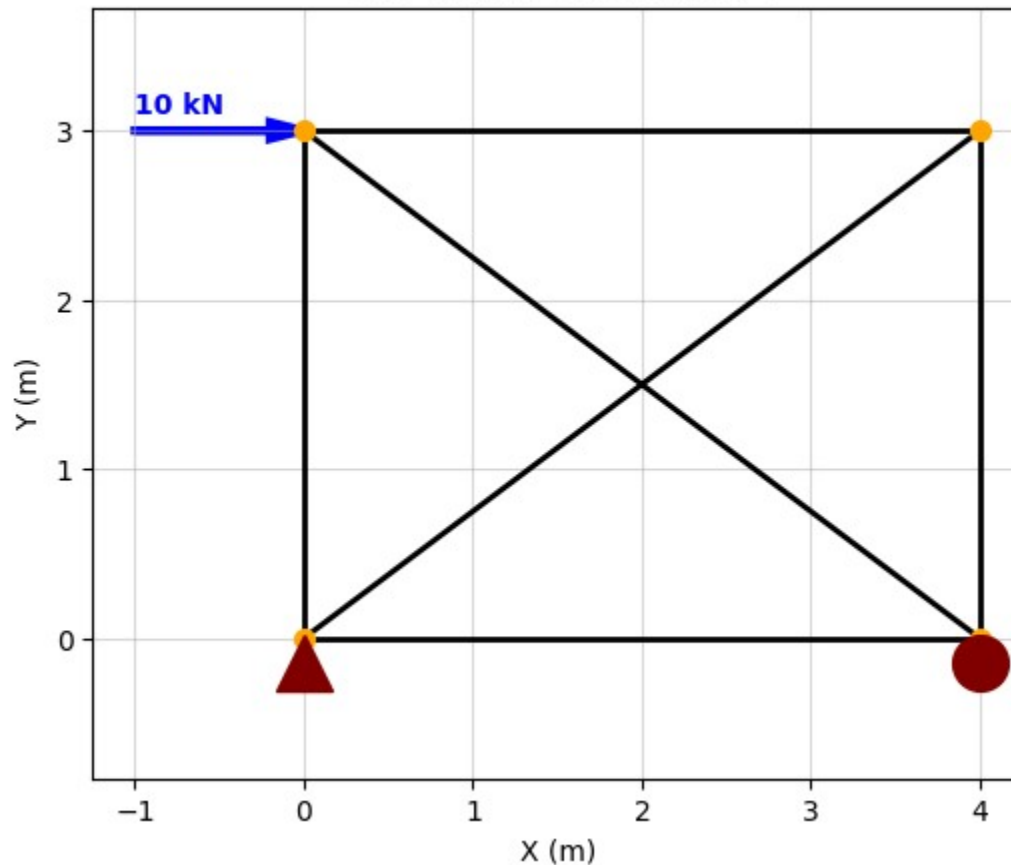
Ángulo de inclinación de los elementos: [0. 90. 180. 270. 36.8699 323.1301] °

Número de elementos: 6

Número de nodos: 4

Número de grados de libertad: 8

SISTEMA ORIGINAL



===== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA =====

Matriz global del sistema:

[113400.	28800.	-75000.	0.	-38400.	-28800.	-0.	-0.]
[28800.	121600.	0.	0.	-28800.	-21600.	-0.	-100000.]
[-75000.	0.	113400.	-28800.	-0.	-0.	-38400.	28800.]
[0.	0.	-28800.	121600.	-0.	-100000.	28800.	-21600.]
[-38400.	-28800.	-0.	-0.	113400.	28800.	-75000.	0.]
[-28800.	-21600.	-0.	-100000.	28800.	121600.	0.	-0.]
[-0.	-0.	-38400.	28800.	-75000.	0.	113400.	-28800.]
[-0.	-100000.	28800.	-21600.	0.	-0.	-28800.	121600.]]

===== RESTRICCIONES =====

Nodo 1: Dx=1 (Restringido), Dy=1 (Restringido)
Nodo 2: Dx=0 (Libre), Dy=1 (Restringido)
Nodo 3: Dx=0 (Libre), Dy=0 (Libre)
Nodo 4: Dx=0 (Libre), Dy=0 (Libre)

===== VECTOR DE FUERZAS EXTERNAS =====

Fuerzas externas: [0 0 0 0 0 0 10 0] kN

===== REDUCCIÓN DEL SISTEMA =====

Grados de libertad activos: 5
Índices: [3 5 6 7 8]

Matriz global reducida:

```
[[113400.    -0.    -0. -38400.  28800.]  
 [    -0. 113400.  28800. -75000.    0.]  
 [    -0.  28800. 121600.    0.    -0.]  
 [-38400. -75000.    0. 113400. -28800.]  
 [ 28800.    0.    -0. -28800. 121600.]]
```

Vector de fuerzas reducido: [0 0 0 10 0] kN

===== SOLUCIÓN DEL SISTEMA =====

===== DESPLAZAMIENTOS =====

Nodo 1: Ux = 0.000e+00 m, Uy = 0.000e+00 m
Nodo 2: Ux = 6.667e-05 m, Uy = 0.000e+00 m
Nodo 3: Ux = 1.583e-04 m, Uy = -3.750e-05 m
Nodo 4: Ux = 2.250e-04 m, Uy = 3.750e-05 m

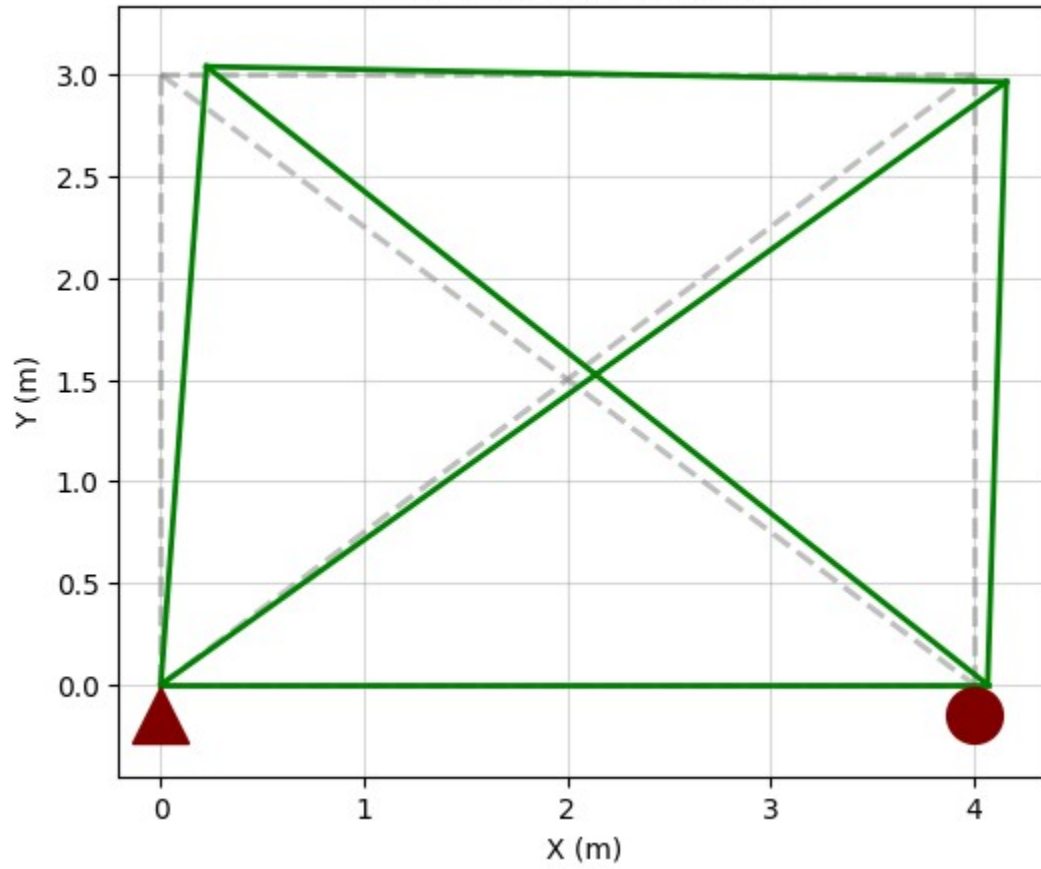
===== REACCIONES =====

Nodo 1: Rx = -10.00 kN, Ry = -7.50 kN
Nodo 2: Rx = 0.00 kN, Ry = 7.50 kN
Nodo 4: Rx = 10.00 kN, Ry = 0.00 kN

===== FUERZAS INTERNAS =====

Elemento 1: 5.000 kN - Tracción
Elemento 2: -3.750 kN - Compresión
Elemento 3: -5.000 kN - Compresión
Elemento 4: 3.750 kN - Tracción
Elemento 5: 6.250 kN - Tracción
Elemento 6: -6.250 kN - Compresión

SISTEMA DEFORMADO



MODELO OPENSEES ARMADURAS: EJERCICIO 1

```
# ===== IMPORTACIÓN DE BIBLIOTECAS =====
import openseespy.opensees as ops # Biblioteca principal de OpenSees para análisis estructural
import numpy as np # Biblioteca para cálculos numéricos
import matplotlib.pyplot as plt # Biblioteca para visualización gráfica

# ===== CONFIGURACIÓN INICIAL =====
print("MODELO EN OPENSEES PARA EL EJERCICIO 1 DE ARMADURAS")

ops.wipe() # Limpiar cualquier modelo existente en memoria
ops.model('basic', '-ndm', 2, '-ndf', 2) # Crear un nuevo modelo básico: '-ndm', 2: Modelo
en 2 dimensiones (X, Y). '-ndf', 2: 2 grados de libertad por nodo (desplazamientos en X e
Y)

# ===== DATOS DE ENTRADA =====
A = 0.0015 # Área transversal de Los elementos en m²
ops.uniaxialMaterial('Elastic', 1, 2e8) # Definir material elástico: 'Elastic': Tipo de
material (lineal elástico); 1: Identificador del material; Módulo de elasticidad en kN/m²

# ===== CREACIÓN DE NODOS =====
# Definir coordenadas de Los 4 nodos de La armadura
ops.node(1, 0.0, 0.0) # Nodo 1
ops.node(2, 4.0, 0.0) # Nodo 2
ops.node(3, 4.0, 3.0) # Nodo 3
ops.node(4, 0.0, 3.0) # Nodo 4

# ===== CREACIÓN DE ELEMENTOS (BARRAS) =====
Elementos = [] # Lista para almacenar información de elementos

# Elemento 1: Barra horizontal inferior
ops.element("Truss", 1, 1, 2, A, 1)
Elementos.append({"ID": 1, "Nodo_i": 1, "Nodo_j": 2})

# Elemento 2: Barra vertical derecha
ops.element("Truss", 2, 2, 3, A, 1)
Elementos.append({"ID": 2, "Nodo_i": 2, "Nodo_j": 3})

# Elemento 3: Barra horizontal superior
ops.element("Truss", 3, 3, 4, A, 1)
Elementos.append({"ID": 3, "Nodo_i": 3, "Nodo_j": 4})

# Elemento 4: Barra vertical izquierda
ops.element("Truss", 4, 4, 1, A, 1)
Elementos.append({"ID": 4, "Nodo_i": 4, "Nodo_j": 1})

# Elemento 5: Diagonal principal (izquierda inferior a derecha superior)
ops.element("Truss", 5, 1, 3, A, 1)
Elementos.append({"ID": 5, "Nodo_i": 1, "Nodo_j": 3})
```

```
# Elemento 6: Diagonal secundaria (izquierda superior a derecha inferior)
ops.element("Truss", 6, 4, 2, A, 1)
elementos.append({"ID": 6, "Nodo_i": 4, "Nodo_j": 2})

# ===== CONDICIONES DE CONTORNO (APOYOS) =====
# Fijar grados de libertad: ops.fix(nodo_ID, restricción_X, restricción_Y)
# 0: Libre (puede moverse), 1: Restringido (no puede moverse)
ops.fix(1, 1, 1) # Nodo 1: Apoyo fijo (restringido en X e Y)
ops.fix(2, 0, 1) # Nodo 2: Apoyo móvil (Libre en X, restringido en Y)

# ===== CARGAS APLICADAS =====
# Definir serie temporal para la aplicación de cargas
ops.timeSeries("Linear", 1) # Carga aplicada linealmente en el tiempo

# Crear patrón de carga
ops.pattern("Plain", 1, 1) # Patrón de carga estática, asociado a la serie temporal 1

ops.load(4, 10, 0) # Aplicar carga en nodo 4: 10 kN en dirección X positiva, 0 kN en
dirección Y

# ===== GRÁFICA DEL SISTEMA ORIGINAL =====
plt.figure(figsize=(6, 5))
plt.title('Sistema original')

# Dibujar todos los elementos como líneas negras
for element_data in Elementos:
    nodo_i = element_data["Nodo_i"]
    nodo_j = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos del elemento
    xi, yi = ops.nodeCoord(nodo_i)
    xj, yj = ops.nodeCoord(nodo_j)

    # Dibujar la barra
    plt.plot([xi, xj], [yi, yj], 'k-', lw=2)

# Dibujar los nodos como puntos púrpuras
for i in range(1, 5):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'o', color='purple', markersize=7)

# Dibujar flecha indicando la carga aplicada
plt.arrow(ops.nodeCoord(4)[0]-1, ops.nodeCoord(4)[1], 0.8, 0, head_width=0.1,
          head_length=0.2, fc='b', ec='b', lw=3)
plt.text(ops.nodeCoord(4)[0]-1, ops.nodeCoord(4)[1]+0.1, '10 kN', color='b')

# Dibujar símbolos para los tipos de apoyo
plt.plot(ops.nodeCoord(1)[0], ops.nodeCoord(1)[1]-0.1, '^', color='maroon', markersize=20)
# Apoyo fijo
plt.plot(ops.nodeCoord(2)[0], ops.nodeCoord(2)[1]-0.1, 'o', color='maroon', markersize=20)
# Apoyo móvil
```

```
# Configurar ejes y visualización
plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid(True, alpha=0.5)
plt.axis('equal') # Mantener relación de aspecto 1:1
plt.show()

# ===== CONFIGURACIÓN DEL ANÁLISIS ESTÁTICO =====
ops.system('BandSPD') # Solucionador: Banda, Simétrico, Positivo Definido
ops.numberer('RCM') # Renumeración: Reverse Cuthill-McKee (optimiza ancho de banda)
ops.constraints('Plain') # Manejo de restricciones: método estándar
ops.integrator('LoadControl', 1.0) # Control de carga: aplicar 100% de la carga en un paso
ops.algorithm('Linear') # Algoritmo: Lineal (adecuado para análisis elástico lineal)
ops.analysis('Static') # Tipo de análisis: estático

# Configurar grabación de resultados en archivos
ops.recorder('Node', '-file', 'desplazamientos.out', '-time', '-nodeRange', 1, 4, '-dof',
1, 2, 'disp')
ops.recorder('Element', '-file', 'fuerzas_elementos.out', '-time', '-elementRange', 1, 6,
'-dof', 'globalForce')

# Ejecutar el análisis (un solo paso ya que es lineal)
ops.analyze(1)

# ===== OBTENCIÓN DE RESULTADOS =====
print("\n=== RESUMEN DE RESULTADOS ===")

print("\n=== DESPLAZAMIENTOS DE LOS NODOS ===")
for i in range(1, 5):
    disp = ops.nodeDisp(i) # Obtener desplazamientos del nodo i
    print(f"Nodo {i}: Ux = {disp[0]:.3e} m, Uy = {disp[1]:.3e} m")

print("\n=== FUERZAS AXIALES ===")
axial_forces = {} # Diccionario para almacenar fuerzas axiales por elemento
for element_data in Elementos:
    ele_id = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Obtener fuerzas internas del elemento (fuerzas en los nodos)
    forces = ops.eleForce(ele_id)
    force_x = forces[0] # Fuerza en dirección X
    force_y = forces[1] # Fuerza en dirección Y

    # Calcular vector de dirección del elemento
    dir_x = xf - xi
    dir_y = yf - yi
    L = np.sqrt(dir_x**2 + dir_y**2) # Longitud del elemento
```

```
# Vector unitario de dirección
unit_dir_x = dir_x/L
unit_dir_y = dir_y/L
# Cálculo de fuerza axial: proyección de la fuerza sobre la dirección del elemento
axial_force = force_x*unit_dir_x + force_y*unit_dir_y
axial_forces[ele_id] = axial_force

# Determinar si es Tracción(-) o Compresión(+)
tipo = "Tracción" if axial_force<0 else "Compresión" if axial_force>0 else ""
print(f"Elemento {ele_id}: {axial_force:.3f} kN - {tipo}")

print("\n=== REACCIONES ===")
ops.reactions() # Calcular reacciones

# Obtener componentes de reacción para cada apoyo
R1_x = ops.nodeReaction(1, 1) # Reacción en X del nodo 1
R1_y = ops.nodeReaction(1, 2) # Reacción en Y del nodo 1
R2_x = ops.nodeReaction(2, 1) # Reacción en X del nodo 2
R2_y = ops.nodeReaction(2, 2) # Reacción en Y del nodo 2

print(f"Reacción en apoyo fijo (Nodo 1): Rx = {R1_x:.3f} kN, Ry = {R1_y:.3f} kN")
print(f"Reacción en apoyo móvil (Nodo 2): Rx = {R2_x:.3f} kN, Ry = {R2_y:.3f} kN")

print("\n=== VERIFICACIÓN DE EQUILIBRIO ===")

Suma_Fx = R1_x + R2_x + 10 # Sumatoria de fuerzas en X (reacciones + carga externa)
Suma_Fy = R1_y + R2_y # Sumatoria de fuerzas en Y
Suma_M1 = -(10*3) + (R2_y*4) # Sumatoria de momentos respecto al nodo 1
print(f"ΣFx = {Suma_Fx:.6f} kN")
print(f"ΣFy = {Suma_Fy:.6f} kN")
print(f"ΣM1 = {Suma_M1:.6f} kN")

# Verificar si las sumatorias son cercanas a cero (dentro de tolerancia)
if abs(Suma_Fx)<1e-6 and abs(Suma_Fy)<1e-6 and abs(Suma_M1)<1e-6:
    print("✓ Equilibrio verificado")
else:
    print("✗ Error en equilibrio")

# ===== GRÁFICAS ADICIONALES DE RESULTADOS =====
plt.figure(figsize=(6, 10))

# === GRÁFICA 1: SISTEMA DEFORMADO (AMPLIFICADO) ===
plt.subplot(2, 1, 1)
plt.title('Sistema deformado')

# Primero dibujar el sistema original (líneas punteadas)
for element_data in Elementos:
    ele_id = element_data["ID"]
```

```

Ni = element_data["Nodo_i"]
Nf = element_data["Nodo_j"]

xi, yi = ops.nodeCoord(Ni)
xf, yf = ops.nodeCoord(Nf)

# Sistema original en líneas punteadas grises
plt.plot([xi, xf], [yi, yf], 'k--', lw=1, alpha=0.5, label='Original' if
element_data["ID"] == 1 else "")

scale_factor = 1000 # Factor de amplificación para visualizar deformaciones (Los
desplazamientos reales son muy pequeños)

# Dibujar sistema deformado (amplificado)
for element_data in Elementos:
    ele_id = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Obtener desplazamientos reales
    disp_i = ops.nodeDisp(Ni)
    disp_f = ops.nodeDisp(Nf)

    # Calcular coordenadas deformadas (amplificadas)
    xi_def = xi + disp_i[0] * scale_factor
    yi_def = yi + disp_i[1] * scale_factor
    xf_def = xf + disp_f[0] * scale_factor
    yf_def = yf + disp_f[1] * scale_factor

    # Sistema deformado en líneas verdes continuas
    plt.plot([xi_def, xf_def], [yi_def, yf_def], 'g-', lw=2, label='Deformada' if
element_data["ID"] == 1 else "")

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.legend()

# === GRÁFICA 2: DIAGRAMA DE FUERZAS AXIALES ===
plt.subplot(2, 1, 2)
plt.title('Fuerzas Axiales (Rojo=Tracción, Azul=Compresión)')

colors = [] # Asignar colores según tipo de fuerza axial
for ele_id in range(1, 7):
    if axial_forces[ele_id] >= 0:
        colors.append('blue') # Compresión
    else:
        colors.append('red') # Tracción

```



```
# Dibujar elementos con colores según fuerza axial
for i, element_data in enumerate(Elementos):
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

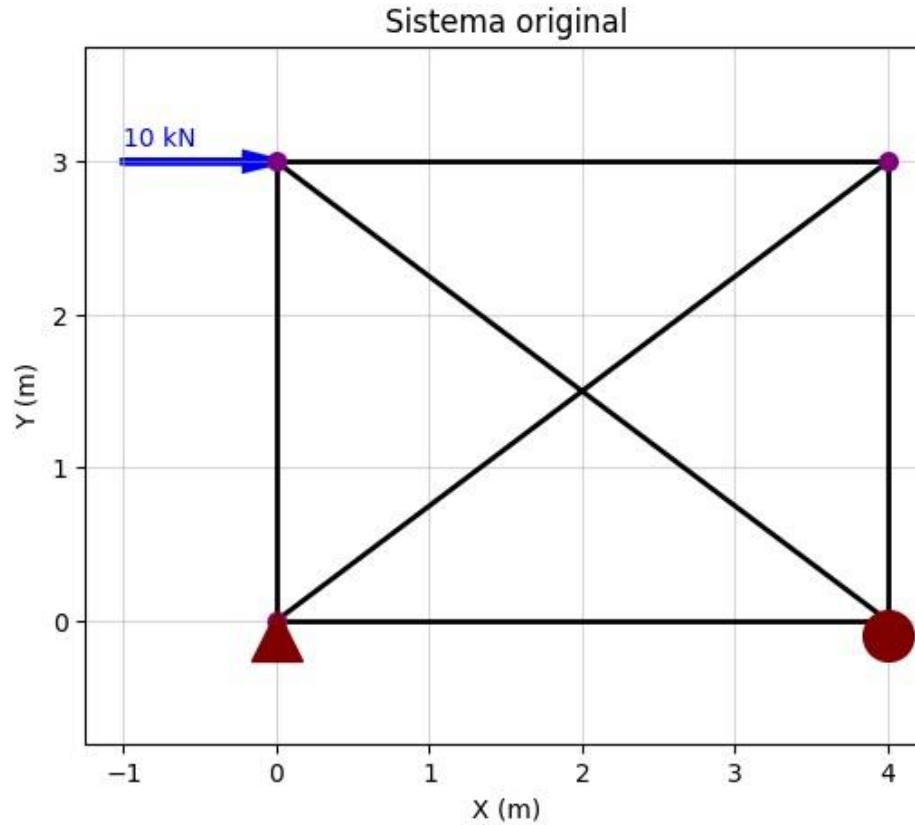
    # Dibujar barra con color correspondiente y etiqueta de fuerza
    plt.plot([xi, xf], [yi, yf], color=colors[i], linewidth=2, label=f'Ele {i+1}:
{axial_forces[i+1]:+.1f} kN')

# Dibujar nodos
for i in range(1, 5):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'ko', markersize=6)

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.legend()

plt.tight_layout() # Ajustar espaciado entre subplots
plt.show()
```

MODELO EN OPENSEES PARA EL EJERCICIO 1 DE ARMADURAS



=== RESUMEN DE RESULTADOS ===

=== DESPLAZAMIENTOS DE LOS NODOS ===

Nodo 1: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m
 Nodo 2: $U_x = 6.667e-05$ m, $U_y = 0.000e+00$ m
 Nodo 3: $U_x = 1.583e-04$ m, $U_y = -3.750e-05$ m
 Nodo 4: $U_x = 2.250e-04$ m, $U_y = 3.750e-05$ m

=== FUERZAS AXIALES ===

Elemento 1: -5.000 kN - Tracción
 Elemento 2: 3.750 kN - Compresión
 Elemento 3: 5.000 kN - Compresión
 Elemento 4: -3.750 kN - Tracción
 Elemento 5: -6.250 kN - Tracción
 Elemento 6: 6.250 kN - Compresión

=== REACCIONES ===

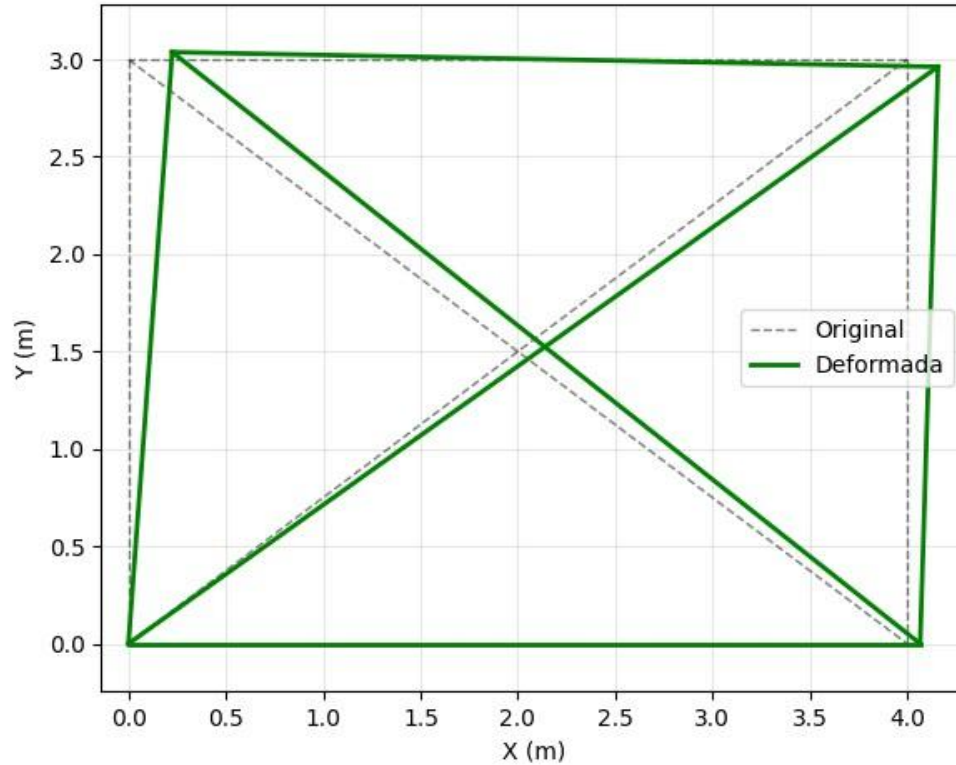
Reacción en apoyo fijo (Nodo 1): $R_x = -10.000$ kN, $R_y = -7.500$ kN
 Reacción en apoyo móvil (Nodo 2): $R_x = 0.000$ kN, $R_y = 7.500$ kN

=== VERIFICACIÓN DE EQUILIBRIO ===

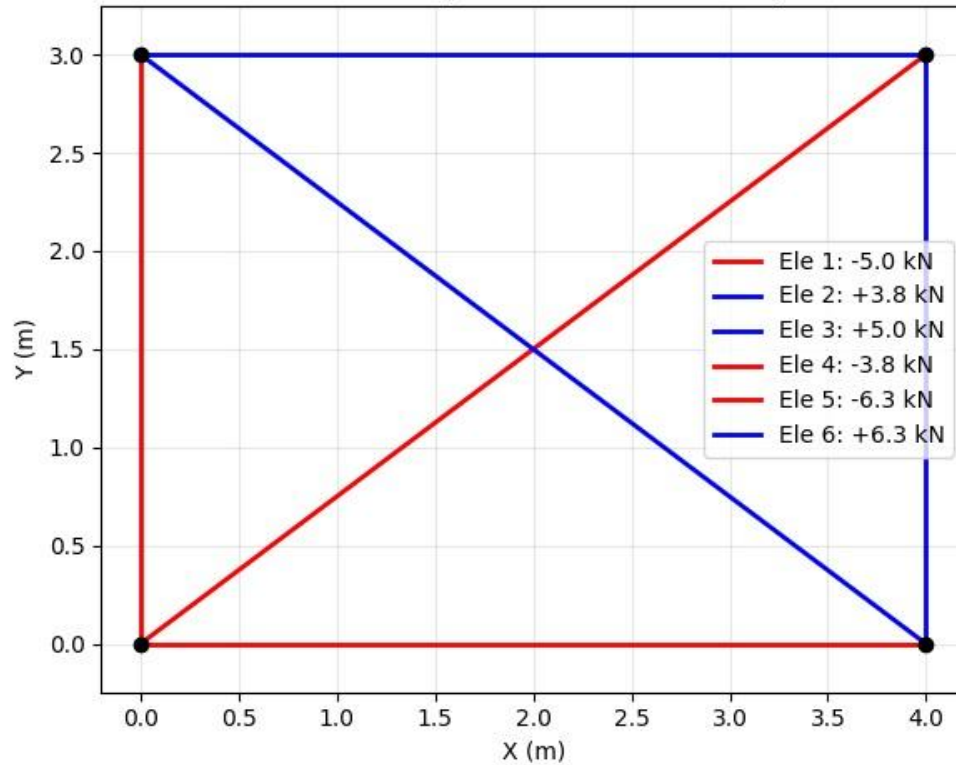
$\Sigma F_x = -0.000000$ kN
 $\Sigma F_y = 0.000000$ kN
 $\Sigma M_1 = 0.000000$ kN

✓ Equilibrio verificado

Sistema deformado



Fuerzas Axiales (Rojo=Tracción, Azul=Compresión)



MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON

ARMADURAS: EJERCICIO 2

```
import numpy as np
import math
import matplotlib.pyplot as plt

print("MÉTODO MATRICIAL DE RIGIDEZ PARA ARMADURAS EJERCICIO 2 - ASENTAMIENTOS EN LOS APOYOS")

# ===== DATOS DE ENTRADA =====
print("\n===== DATOS DE ENTRADA =====")
# Propiedades de los elementos
E = np.array([2e8, 2e8, 2e8, 2e8, 2e8, 2e8, 2e8, 2e8, 2e8, 2e8, 2e8, 2e8]) # Módulo de elasticidad [kN/m²]
A = np.array([0.0015, 0.0015, 0.0015, 0.0015, 0.0015, 0.0015, 0.0015, 0.0015, 0.0015, 0.0015, 0.0015, 0.0015]) # Área sección transversal [m²]
L = np.array([4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 5.65685, 5.65685, 5.65685, 5.65685]) # Longitud de los elementos [m]
a = np.array([0, 0, 0, 0, 0, 90, 90, 90, 45, -45, 45, -45]) # Ángulo de elementos respecto al eje X [°]
m = 12 # Número de elementos en la armadura
n = 7 # Número de nodos en la armadura
GL = 14 # Grados de libertad totales (7 nodos × 2 gL/nodo)

print(f"Módulo de elasticidad: {E} kN/m²")
print(f"Área de la sección transversal: {A} m²")
print(f"Longitud: {L} m")
print(f"Ángulo de inclinación: {a} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")

# ===== COORDENADAS DE NODOS =====
# Coordenadas [X, Y] de cada nodo en metros
coordenadas_nodos = np.array([
    [0.0, 4.0], # Nodo 1
    [4.0, 4.0], # Nodo 2
    [8.0, 4.0], # Nodo 3
    [0.0, 0.0], # Nodo 4
    [4.0, 0.0], # Nodo 5
    [8.0, 0.0], # Nodo 6
    [12.0, 0.0] # Nodo 7
])

# ===== CONECTIVIDAD DE ELEMENTOS =====
# Define la conectividad entre nodos para cada elemento [Nodo inicial, Nodo final]
Elementos = np.array([
    [1, 2], # E1: N1 - N2
    [2, 3], # E2: N2 - N3
    [4, 5], # E3: N4 - N5
    [5, 6], # E4: N5 - N6
    [6, 7], # E5: N6 - N7
])
```

```
[4, 1], # E6: N4 - N1
[5, 2], # E7: N5 - N2
[6, 3], # E8: N6 - N3
[4, 2], # E9: N4 - N2
[2, 6], # E10: N2 - N6
[5, 3], # E11: N5 - N3
[3, 7]] # E12: N3 - N7

# ===== GRÁFICA SISTEMA ORIGINAL =====
plt.figure(figsize=(8, 4))
plt.title('SISTEMA ORIGINAL', fontsize=14, fontweight='bold')

# Dibujar elementos originales
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    # Obtener coordenadas de los nodos del elemento
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]
    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=2)

# Dibujar nodos
for i, (x, y) in enumerate(coordenadas_nodos):
    plt.plot(x, y, 'o', color='gold', markersize=6)

# Dibujar apoyos (condiciones de frontera)
plt.text(0-0.3, 4, '▸', fontsize=30, color='maroon', ha='center', va='center') # Apoyo
fijo Nodo 1
plt.plot(0, 0-0.3, '^', color='maroon', markersize=20) # Apoyo fijo Nodo 4
plt.plot(8, 0-0.3, 'o', color='maroon', markersize=20) # Apoyo móvil Nodo 6

# Dibujar fuerza aplicada
plt.arrow(12, 0, 0, -0.8, head_width=0.2, head_length=0.2, fc='steelblue',
ec='steelblue', linewidth=2)
plt.text(9.5, -1, '-19.61 kN', fontsize=11, color='steelblue', fontweight='bold')

plt.xlabel('X [m]')
plt.ylabel('Y [m]')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.show()

# ===== GRADOS DE LIBERTAD POR ELEMENTO =====
# Mapeo de grados de libertad para cada elemento (desplazamientos en X e Y)
#
# E1|E2|E3|E4|E5|E6|E7|E8|E9|E10|E11|E12
Nx = np.array([1, 3, 7, 9, 11, 7, 9, 11, 7, 3, 9, 5]) # Dx GL inicial
Ny = np.array([2, 4, 8, 10, 12, 8, 10, 12, 8, 4, 10, 6]) # Dy GL inicial
Fx = np.array([3, 5, 9, 11, 13, 1, 3, 5, 3, 11, 5, 13]) # Dx GL final
Fy = np.array([4, 6, 10, 12, 14, 2, 4, 6, 4, 12, 6, 14]) # Dy GL final

# ===== ASENTAMIENTOS EN APOYOS =====
print("\n===== ASENTAMIENTOS =====")
# Vector de asentamientos prescritos para cada grado de libertad [m]
```

```
#
13 14
asentamientos = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -0.02,
0.0, 0.0])

# Mostrar asentamientos por nodo
for i in range(n):
    Asx = asentamientos[i*2]
    Asy = asentamientos[i*2+1]
    if abs(Asx) > 0 or abs(Asy) > 0:
        print(f"Nodo {i+1}: As_x = {Asx:.3f} m , As_y = {Asy:.3f} m")

# ===== ENSAMBLE MATRIZ GLOBAL DE RIGIDEZ =====
print("\n\n==== ENSAMBLAJE DE LA MATRIZ DEL SISTEMA =====")
kG = np.zeros((GL, GL)) # Crear matriz global de rigidez (GL x GL)
k_elementos = [] # Lista para almacenar matrices de rigidez de cada elemento (para uso
posterior)

# Ensamblar matriz global elemento por elemento
for i in range(m):
    theta = math.radians(a[i]) # Convertir ángulo a radianes

    # Calcular coseno y seno del ángulo del elemento
    c = math.cos(theta)
    s = math.sin(theta)

    # Crear matriz de rigidez 4x4 del elemento en coordenadas globales
    kg_e = np.zeros((4, 4))

    # Calcular constante de rigidez axial (AE/L)
    Factor = (A[i] * E[i]) / L[i]

    # Construir matriz de rigidez del elemento en coordenadas globales
    kg_e[0, 0] = Factor * c**2
    kg_e[0, 1] = Factor * c * s
    kg_e[0, 2] = -Factor * c**2
    kg_e[0, 3] = -Factor * c * s

    kg_e[1, 0] = Factor * c * s
    kg_e[1, 1] = Factor * s**2
    kg_e[1, 2] = -Factor * c * s
    kg_e[1, 3] = -Factor * s**2

    kg_e[2, 0] = -Factor * c**2
    kg_e[2, 1] = -Factor * c * s
    kg_e[2, 2] = Factor * c**2
    kg_e[2, 3] = Factor * c * s

    kg_e[3, 0] = -Factor * c * s
    kg_e[3, 1] = -Factor * s**2
    kg_e[3, 2] = Factor * c * s
    kg_e[3, 3] = Factor * s**2
```

```
# Guardar matriz del elemento para uso posterior
k_elementos.append(kg_e.copy())

# Obtener grados de libertad del elemento (convertir a índices base 0)
GL_elem = [Nx[i]-1, Ny[i]-1, Fx[i]-1, Fy[i]-1]

# Ensamblar contribución del elemento en la matriz global
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj]

#print(f"\nELEMENTO: {i+1}")
#print(f"Longitud: {L[i]} m, Ángulo: {a[i]}°")
#print(f"Cos(θ): {c:.2f}, Sen(θ): {s:.2f}")
#print(f"Matriz del elemento:\n {np.round(kL, 0)}")
#print(f"Matriz global después del elemento {i+1}:\n {np.round(kG, 0)}")
print(f"Matriz de rigidez global del sistema ({GL}×{GL}) [kN/m]:\n{np.round(kG, 0)}")

# ===== VECTOR DE FUERZAS POR ASENTAMIENTOS =====
print("\n===== VECTOR DE FUERZAS POR ASENTAMIENTOS =====")
# Inicializar vector de fuerzas equivalentes debido a asentamientos
F_As = np.zeros(GL)
# Calcular fuerzas por asentamientos para cada elemento
for i in range(m):
    # Obtener la matriz de rigidez del elemento en coordenadas globales
    kGe = k_elementos[i]

    # Extraer desplazamientos por asentamientos para los GL del elemento
    U_asentamiento = np.array([
        asentamientos[Nx[i]-1], # Desplazamiento X nodo inicial
        asentamientos[Ny[i]-1], # Desplazamiento Y nodo inicial
        asentamientos[Fx[i]-1], # Desplazamiento X nodo final
        asentamientos[Fy[i]-1]] # Desplazamiento Y nodo final

    # Calcular fuerzas del elemento debido a asentamientos: {f_As} = [ke]_G × {U_As}
    Fe_As = np.matmul(kGe, U_asentamiento)

    # Ensamblar el vector global de fuerzas por asentamientos
    F_As[Nx[i]-1] += Fe_As[0] # Fuerza en GL X nodo inicial
    F_As[Ny[i]-1] += Fe_As[1] # Fuerza en GL Y nodo inicial
    F_As[Fx[i]-1] += Fe_As[2] # Fuerza en GL X nodo final
    F_As[Fy[i]-1] += Fe_As[3] # Fuerza en GL Y nodo final
print(f"Vector de fuerzas equivalentes por asentamientos [kN]:\n{np.round(F_As, 0)}")

# ===== DEFINICIÓN DE RESTRICCIONES (CONDICIONES DE CONTORNO) =====
print("\n===== RESTRICCIONES =====")
# Vector que indica restricciones: 0 = Grado de Libertad Libre, 1 = Grado de Libertad restringido
#
# GL: 1 2 3 4 5 6 7 8 9 10 11 12 13 14
restricciones = np.array([1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0])
```



```
# Mostrar estado de restricciones por nodo
for i in range(n):
    rtx = "[Restringido]" if restricciones[i*2] == 1 else "[Libre]"
    rty = "[Restringido]" if restricciones[i*2+1] == 1 else "[Libre]"
    print(f"Node {i+1}: Dx={restricciones[i*2]} {rtx}, Dy={restricciones[i*2+1]} {rty}")

# Identificar grados de libertad activos (desplazamientos desconocidos, fuerzas conocidas)
GL_activos = np.where(restricciones == 0)[0] # Índices de GL activos
n_GL_activos = len(GL_activos) # Número de grados de libertad activos
print(f"\nNúmero de grados de libertad activos: {n_GL_activos}")
print(f"Índices de los grados de libertad activos: {GL_activos+1}")

# Identificar grados de libertad restringidos (desplazamientos conocidos, fuerzas desconocidas)
GL_restringidos = np.where(restricciones==1)[0] # Índices de los grados de libertad restringidos
n_GL_restringidos = len(GL_restringidos) # Número de grados de libertad restringidos
print(f"\nNúmero de grados de libertad restringidos: {n_GL_restringidos}")
print(f"Índices de los grados de libertad restringidos: {GL_restringidos+1}")

# ===== VECTOR DE FUERZAS EXTERNAS APLICADAS =====
print("\n===== VECTOR DE FUERZAS EXTERNAS =====")
# Vector de fuerzas nodales externas [kN]
# Nota: Fuerza negativa en Y indica fuerza hacia abajo
# GL: 1 2 3 4 5 6 7 8 9 10 11 12 13 14
F = np.array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -19.6133])
# Fuerza de -19.6133 kN en dirección Y, aplicada en nodo 7 (GL 14)
print(f"Vector de fuerzas externas: {F} kN")

# ===== REDUCCIÓN DEL SISTEMA DE ECUACIONES =====
print("\n===== REDUCCIÓN DEL SISTEMA =====")
# Reducir el sistema eliminando grados de libertad restringidos
# Extraer submatriz y subvectores para grados de libertad activos
K_reducida = kG[np.ix_(GL_activos, GL_activos)] # Matriz de rigidez reducida
F_reducido = F[GL_activos] # Vector de fuerzas externas reducido
F_As_reducido = F_As[GL_activos] # Vector de fuerzas por asentamientos reducido

print(f"Matriz de rigidez reducida ({n_GL_activos} x {n_GL_activos}) [kN/m]\n{np.round(K_reducida, 0)}")
print(f"\nVector de fuerzas reducido [kN]: {np.round(F_reducido, 3)}")
print(f"\nVector de fuerzas por asentamiento reducido [kN]: {np.round(F_As_reducido, 3)}")

# ===== SOLUCIÓN DEL SISTEMA DE ECUACIONES =====
print("\n\n===== SOLUCIÓN DEL SISTEMA =====")
# Resolver sistema de ecuaciones: {F_ext} = [K]{U} + {F_As}
# → [K]{U} = {F_ext} - {F_As}
# → {U} = [K]-1{F_ext} - {F_As}
# Calcular inversa de la matriz de rigidez reducida
K_reducida_inv = np.linalg.inv(K_reducida)
```



```
# Calcular desplazamientos desconocidos
U_desconocidos = np.matmul(K_reducida_inv, F_reducido - F_As_reducido)
# Reconstruir vector completo de desplazamientos
U_totales = np.zeros(GL)
U_totales[GL_activos] = U_desconocidos # Desplazamientos calculados
U_totales[GL_restringidos] = asentamientos[GL_restringidos] # Desplazamientos prescritos
(asentamientos)

# ===== CÁLCULO DE DESPLAZAMIENTOS EN LOS NODOS =====
print("\nDESPLAZAMIENTOS EN LOS NODOS:")
for i in range(n):
    Ux = U_totales[i*2] # Desplazamiento en X del nodo i
    Uy = U_totales[i*2+1] # Desplazamiento en Y del nodo i
    print(f"Node {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m")

# ===== CÁLCULO DE REACCIONES EN LOS APOYOS =====
print("\nREACCIONES EN LOS APOYOS:")
# Calcular reacciones: {R} = [K_global]{U_totales} - {F_As}
Reacciones = np.matmul(kG, U_totales)

for i in range(n):
    Rx = Reacciones[i*2]
    Ry = Reacciones[i*2+1]

    # Mostrar solo reacciones significativas (mayores a 1e-6 kN)
    if abs(Rx) > 1e-6 or abs(Ry) > 1e-6:
        print(f"Node {i+1}: Rx = {Rx:.2f} kN, Ry = {Ry:.2f} kN")

# ===== CÁLCULO DE FUERZAS INTERNAS EN LOS ELEMENTOS =====
print("\nFUERZAS AXIALES DE LOS ELEMENTOS:")
F_elem = np.zeros(m) # Vector para almacenar fuerzas axiales

for i in range(m):
    # Obtener desplazamientos nodales del elemento (incluyendo asentamientos)
    Ue = np.array([
        U_totales[Nx[i]-1], # Ux nodo inicial
        U_totales[Ny[i]-1], # Uy nodo inicial
        U_totales[Fx[i]-1], # Ux nodo final
        U_totales[Fy[i]-1]] # Uy nodo final

    # Calcular fuerza axial en coordenadas Locales
    theta = math.radians(a[i]) # Ángulo del elemento en radianes
    c = math.cos(theta)
    s = math.sin(theta)
    Factor = (A[i] * E[i]) / L[i] # Rigidez axial del elemento

    # Matriz de rigidez en coordenadas Locales (2x2 para barra axial)
    ke_loc = Factor * np.array([[1, -1], [-1, 1]])

    # Matriz de transformación de coordenadas (global → Local)
    Te = np.array([[c, s, 0, 0], [0, 0, c, s]])
```

```
# Calcular fuerza axial: {f_local} = [k_local][T]{U_global}
ke_loc_Te = np.matmul(ke_loc, Te)
ke_loc_Te_Ue = np.matmul(ke_loc_Te, Ue)
# La fuerza axial es el segundo término del vector (fuerza en el nodo final)
Fuerza_axial = ke_loc_Te_Ue[1]
F_elem[i] = Fuerza_axial

# Determinar tipo de esfuerzo
tipo = "Tracción" if Fuerza_axial>0 else "Compresión" if Fuerza_axial<0 else "Fuerza
axial nula"
print(f"Elemento {i+1}: {Fuerza_axial:7.3f} kN - {tipo}")

# ===== GRÁFICA DEL SISTEMA DEFORMADO =====
FS = 10 # Factor de escala para visualización de desplazamientos (amplifica
deformaciones)
plt.figure(figsize=(8, 4))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar elementos en posición original (líneas discontinuas)
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]
    plt.plot([xi, xf], [yi, yf], '--', color='darkgrey', linewidth=2, alpha=0.7)

# Dibujar apoyos en posición original
plt.text(0-0.3, 4, '▸', fontsize=30, color='lightcoral', ha='center', va='center')
plt.plot(0, 0-0.3, '^', color='lightcoral', markersize=20) plt.plot(8, 0-0.3, 'o',
color='lightcoral', markersize=20)

# Preparar arrays para coordenadas deformadas
coord_nodos_def = np.zeros((len(coordenadas_nodos), 2)) # Matriz para nodos deformados

# Dibujar estructura deformada
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    # Coordenadas originales
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]

    # Desplazamientos de Los nodos
    desp_i = np.array([U_totales[(nodo_i-1)*2], U_totales[(nodo_i-1)*2+1]])
    desp_f = np.array([U_totales[(nodo_f-1)*2], U_totales[(nodo_f-1)*2+1]])

    # Coordenadas deformadas (amplificadas por FS)
    xi_def = xi + desp_i[0] * FS
    yi_def = yi + desp_i[1] * FS
    xf_def = xf + desp_f[0] * FS
    yf_def = yf + desp_f[1] * FS

    # Guardar coordenadas deformadas por NODO
    coord_nodos_def[nodo_i-1] = [xi_def, yi_def]
    coord_nodos_def[nodo_f-1] = [xf_def, yf_def]
```

```
# Dibujar elemento deformado
plt.plot([xi_def, xf_def], [yi_def, yf_def], '-', color='forestgreen', linewidth=2)

# Dibujar apoyos después de la deformación
plt.text(coord_nodos_def[0][0]-0.3, coord_nodos_def[0][1], '►', fontsize=30,
color='maroon', ha='center', va='center')
plt.plot(coord_nodos_def[3][0], coord_nodos_def[3][1]-0.3, '^', color='maroon',
markersize=20)
plt.plot(coord_nodos_def[5][0], coord_nodos_def[5][1]-0.3, 'o', color='maroon',
markersize=20)

# Configurar gráfica
plt.xlabel('X [m]')
plt.ylabel('Y [m]')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

MÉTODO MATRICIAL DE RIGIDEZ PARA ARMADURAS EJERCICIO 2 - ASENTAMIENTOS EN LOS APOYOS

===== DATOS DE ENTRADA =====

Módulo de elasticidad: [2.e+08 2.e+08 2.e+08 2.e+08 2.e+08 2.e+08 2.e+08 2.e+08 2.e+08
2.e+08 2.e+08 2.e+08] kN/m²

Área de la sección transversal: [0.0015 0.0015 0.0015 0.0015 0.0015 0.0015 0.0015 0.0015 0.0015
0.0015 0.0015 0.0015 0.0015] m²

Longitud: [4. 4. 4. 4. 4. 4. 4. 4. 5.65685 5.65685 5.65685 5.65685] m

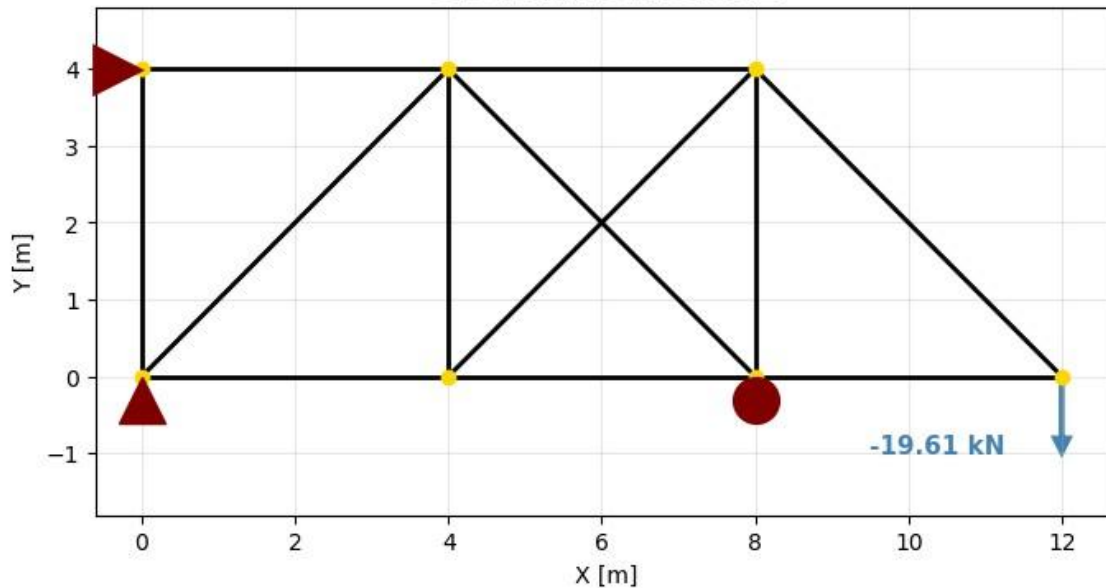
Ángulo de inclinación: [0 0 0 0 0 90 90 90 45 -45 45 -45] °

Número de elementos: 12

Número de nodos: 7

Número de grados de libertad: 14

SISTEMA ORIGINAL



===== ASENTAMIENTOS =====

Nodo 6: As_x = 0.000 m , As_y = -0.020 m

===== ENSAMBLAJE DE LA MATRIZ DEL SISTEMA =====

Matriz de rigidez global del sistema (14×14) [kN/m]:

```
[ [ 75000.  0. -75000.  0.  0.  0. -0. -0.  0.  0.  0.  0.  0.  0. ]
[  0. 75000.  0.  0.  0.  0. -0. -75000.  0.  0.  0.  0.  0.  0. ]
[ -75000.  0. 203033.  0. -75000.  0. -26517. -26517. -0. -0. -26517. 26517.  0.  0. ]
[  0.  0.  0. 128033.  0.  0. -26517. -26517. -0. -75000. 26517. -26517.  0.  0. ]
[  0.  0. -75000.  0. 128033.  0.  0.  0. -26517. -26517. -0. -0. -26517. 26517. ]
[  0.  0.  0.  0.  0. 128033.  0.  0. -26517. -26517. -0. -75000. 26517. -26517. ]
[ -0. -0. -26517. -26517.  0.  0. 101517. 26517. -75000.  0.  0.  0.  0.  0. ]
[ -0. -75000. -26517. -26517.  0.  0. 26517. 101517.  0.  0.  0.  0.  0.  0. ]
[  0.  0. -0. -0. -26517. -26517. -75000.  0. 176517. 26517. -75000.  0.  0.  0. ]
[  0.  0. -0. -75000. -26517. -26517.  0.  0. 26517. 101517.  0.  0.  0.  0. ]
[  0.  0. -26517. 26517. -0. -0.  0.  0. -75000.  0. 176517. -26517. -75000.  0. ]
[  0.  0. 26517. -26517. -0. -75000.  0.  0.  0.  0. -26517. 101517.  0.  0. ]
[  0.  0.  0.  0. -26517. 26517.  0.  0.  0.  0. -75000.  0. 101517. -26517. ]
[  0.  0.  0.  0. 26517. -26517.  0.  0.  0.  0.  0.  0. -26517. 26517. ] ]
```

===== VECTOR DE FUERZAS POR ASENTAMIENTOS =====

Vector de fuerzas equivalentes por asentamientos [kN]:

[0. 0. -530. 530. 0. 1500. 0. 0. 0. 0. 530. -2030. 0. 0.]

===== RESTRICCIONES =====

Nodo 1: Dx=1 [Restringido], Dy=1 [Restringido]

Nodo 2: Dx=0 [Libre], Dy=0 [Libre]

Nodo 3: Dx=0 [Libre], Dy=0 [Libre]

Nodo 4: Dx=1 [Restringido], Dy=1 [Restringido]

Nodo 5: Dx=0 [Libre], Dy=0 [Libre]

Nodo 6: Dx=0 [Libre], Dy=1 [Restringido]

Nodo 7: Dx=0 [Libre], Dy=0 [Libre]

Número de grados de libertad activos: 9

Índices de los grados de libertad activos: [3 4 5 6 9 10 11 13 14]

Número de grados de libertad restringidos: 5

Índices de los grados de libertad restringidos: [1 2 7 8 12]

===== VECTOR DE FUERZAS EXTERNAS =====

Vector de fuerzas externas: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. -19.6133] kN

===== REDUCCIÓN DEL SISTEMA =====

Matriz de rigidez reducida (9 x 9) [kN/m]

```
[[203033. 0. -75000. 0. -0. -0. -26517. 0. 0.]
 [ 0. 128033. 0. 0. -0. -75000. 26517. 0. 0.]
 [-75000. 0. 128033. 0. -26517. -26517. -0. -26517. 26517.]
 [ 0. 0. 0. 128033. -26517. -26517. -0. 26517. -26517.]
 [-0. -0. -26517. -26517. 176517. 26517. -75000. 0. 0.]
 [-0. -75000. -26517. -26517. 26517. 101517. 0. 0. 0.]
 [-26517. 26517. -0. -0. -75000. 0. 176517. -75000. 0.]
 [ 0. 0. -26517. 26517. 0. 0. -75000. 101517. -26517.]
 [ 0. 0. 26517. -26517. 0. 0. 0. -26517. 26517.]]
```

Vector de fuerzas reducido [kN]: [0. 0. 0. 0. 0. 0. 0. 0. -19.613]

Vector de fuerzas por asentamiento reducido [kN]: [-530.33 530.33 0. 1500. 0.
0. 530.33 0. 0.]

===== SOLUCIÓN DEL SISTEMA =====

DESPLAZAMIENTOS EN LOS NODOS:

Nodo 1: Ux = 0.000e+00 m, Uy = 0.000e+00 m

Nodo 2: Ux = 4.021e-03 m, Uy = -9.337e-03 m

Nodo 3: Ux = 5.055e-03 m, Uy = -1.949e-02 m

Nodo 4: Ux = 0.000e+00 m, Uy = 0.000e+00 m

Nodo 5: Ux = -2.141e-03 m, Uy = -1.011e-02 m

Nodo 6: Ux = -3.510e-03 m, Uy = -2.000e-02 m

Nodo 7: Ux = -3.772e-03 m, Uy = -2.905e-02 m

REACCIONES EN LOS APOYOS:

Nodo 1: $R_x = -301.56 \text{ kN}$, $R_y = 0.00 \text{ kN}$

Nodo 4: $R_x = 301.56 \text{ kN}$, $R_y = 140.97 \text{ kN}$

Nodo 6: $R_x = 0.00 \text{ kN}$, $R_y = -121.36 \text{ kN}$

Nodo 7: $R_x = 0.00 \text{ kN}$, $R_y = -19.61 \text{ kN}$

FUERZAS AXIALES DE LOS ELEMENTOS:

Elemento 1: 301.556 kN - Tracción

Elemento 2: 77.532 kN - Tracción

Elemento 3: -160.585 kN - Compresión

Elemento 4: -102.666 kN - Compresión

Elemento 5: -19.613 kN - Compresión

Elemento 6: 0.000 kN - Fuerza axial Nula

Elemento 7: 57.919 kN - Tracción

Elemento 8: 38.305 kN - Tracción

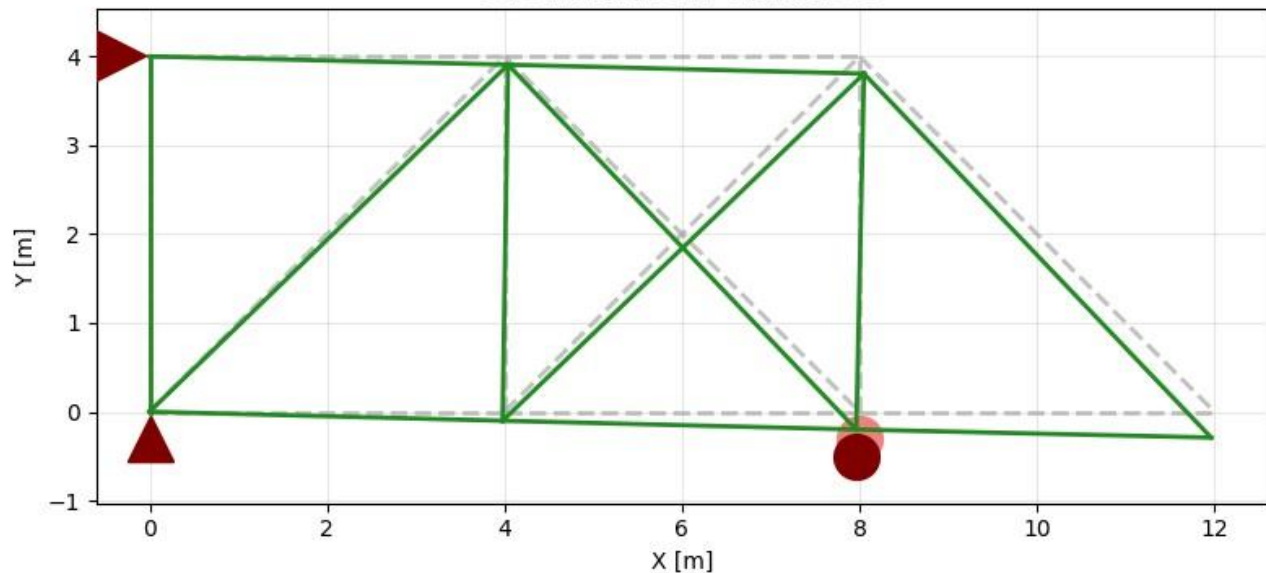
Elemento 9: -199.364 kN - Compresión

Elemento 10: 117.454 kN - Tracción

Elemento 11: -81.909 kN - Compresión

Elemento 12: 27.737 kN - Tracción

SISTEMA DEFORMADO



MODELO OPENSEES

ARMADURAS: EJERCICIO 2

```
import openseespy.opensees as ops
import numpy as np
import matplotlib.pyplot as plt

print("MODELO EN OPENSEES: EJERCICIO 2 - ARMADURAS CON ASENTAMIENTOS EN APOYOS")

# ===== INICIALIZACIÓN DEL MODELO =====
ops.wipe() # Limpiar cualquier modelo previo de la memoria de OpenSees
ops.model('basic', '-ndm', 2, '-ndf', 2) # Crear nuevo modelo básico: '-ndm', 2:
Modelo en 2 dimensiones (plano XY); '-ndf', 2: 2 grados de libertad por nodo

# ===== DATOS DE ENTRADA =====
A = 0.0015 # Área de la sección transversal de todas las barras [m²]
ops.uniaxialMaterial('Elastic', 1, 2e8) # Definir material elástico lineal: 'Elastic':
Tipo de material; 1: Tag (identificador) del material; 2e8: Módulo de elasticidad kN/m²

# ===== DEFINICIÓN DE LA GEOMETRÍA - NODOS =====
# Crear los 7 nodos de la armadura con sus coordenadas (X, Y) en metros
ops.node(1, 0.0, 4.0) # Nodo 1
ops.node(2, 4.0, 4.0) # Nodo 2
ops.node(3, 8.0, 4.0) # Nodo 3
ops.node(4, 0.0, 0.0) # Nodo 4
ops.node(5, 4.0, 0.0) # Nodo 5
ops.node(6, 8.0, 0.0) # Nodo 6
ops.node(7, 12.0, 0.0) # Nodo 7

# ===== DEFINICIÓN DE LA CONECTIVIDAD - ELEMENTOS =====
elements = [] # Lista para almacenar información de los elementos

ops.element("Truss", 1, 1, 2, A, 1) # Elemento 1: Nodo 1 → Nodo 2
elements.append({"ID": 1, "Nodo_i": 1, "Nodo_f": 2})

ops.element("Truss", 2, 2, 3, A, 1) # Elemento 2: Nodo 2 → Nodo 3
elements.append({"ID": 2, "Nodo_i": 2, "Nodo_f": 3})

ops.element("Truss", 3, 4, 5, A, 1) # Elemento 3: Nodo 4 → Nodo 5
elements.append({"ID": 3, "Nodo_i": 4, "Nodo_f": 5})

ops.element("Truss", 4, 5, 6, A, 1) # Elemento 4: Nodo 5 → Nodo 6
elements.append({"ID": 4, "Nodo_i": 5, "Nodo_f": 6})

ops.element("Truss", 5, 6, 7, A, 1) # Elemento 5: Nodo 6 → Nodo 7
elements.append({"ID": 5, "Nodo_i": 6, "Nodo_f": 7})

ops.element("Truss", 6, 4, 1, A, 1) # Elemento 6: Nodo 4 → Nodo 1
elements.append({"ID": 6, "Nodo_i": 4, "Nodo_f": 1})

ops.element("Truss", 7, 5, 2, A, 1) # Elemento 7: Nodo 5 → Nodo 2
elements.append({"ID": 7, "Nodo_i": 5, "Nodo_f": 2})
```

```
ops.element("Truss", 8, 6, 3, A, 1) # Elemento 8: Nodo 6 → Nodo 3
elements.append({"ID": 8, "Nodo_i": 6, "Nodo_f": 3})

ops.element("Truss", 9, 4, 2, A, 1) # Elemento 9: Nodo 4 → Nodo 2
elements.append({"ID": 9, "Nodo_i": 4, "Nodo_f": 2})

ops.element("Truss", 10, 2, 6, A, 1) # Elemento 10: Nodo 2 → Nodo 6
elements.append({"ID": 10, "Nodo_i": 2, "Nodo_f": 6})

ops.element("Truss", 11, 5, 3, A, 1) # Elemento 11: Nodo 5 → Nodo 3
elements.append({"ID": 11, "Nodo_i": 5, "Nodo_f": 3})

ops.element("Truss", 12, 3, 7, A, 1) # Elemento 12: Nodo 3 → Nodo 7
elements.append({"ID": 12, "Nodo_i": 3, "Nodo_f": 7})

# ===== VISUALIZACIÓN: SISTEMA ORIGINAL =====
plt.figure(figsize=(6, 4))
plt.title('SISTEMA ORIGINAL', fontsize=14, fontweight='bold')

# Graficar elementos (barras)
for element_data in elements:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_f"]

    # Obtener coordenadas de los nodos del elemento
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Dibujar elemento como línea negra
    plt.plot([xi, xf], [yi, yf], 'k-', lw=2)

# Graficar nodos
for i in range(1, 8):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'o', color='gold', markersize=5)

# Dibujar apoyos con símbolos
plt.text(0-0.3, 4, '▸', fontsize=20, color='maroon', ha='center', va='center') # Apoyo
fijo Nodo 1
plt.plot(0, 0-0.3, '^', color='maroon', markersize=15) # Apoyo fijo Nodo 4
plt.plot(8, 0-0.3, 'o', color='maroon', markersize=15) # Apoyo móvil Nodo 6

# Dibujar fuerza aplicada
plt.arrow(12, 0, 0, -0.8, head_width=0.2, head_length=0.2, fc='steelblue',
ec='steelblue', linewidth=2)
plt.text(9.4, -1, '-19.61 kN', fontsize=11, color='steelblue', fontweight='bold')

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid(True, alpha=0.3)
plt.axis('equal')
```



```
plt.show()

# ===== DEFINICIÓN DE ASENTAMIENTOS =====
asentamiento_6_y = -0.02 # Asentamiento en el nodo 6: 2 cm hacia abajo [m]

# ===== CONFIGURACIÓN DEL ANÁLISIS ESTÁTICO =====
# Configurar los componentes del análisis estático lineal

# 1. Solucionador del sistema de ecuaciones
ops.system('BandSPD') # Banda, Simétrico, Positivo Definido (eficiente para matrices
banda)

# 2. Numeración de grados de libertad
ops.numberer('RCM') # Reverse Cuthill-McKee: reduce el ancho de banda de la matriz

# 3. Manejo de restricciones (especial para asentamientos no homogéneos)
ops.constraints('Transformation') # Método de transformación para restricciones no
homogéneas

# 4. Control de la aplicación de cargas
ops.integrator('LoadControl', 1.0) # Aplica el 100% de las cargas en un solo paso

# 5. Algoritmo de solución
ops.algorithm('Linear') # Algoritmo lineal (adecuado para análisis elástico lineal)

# 6. Tipo de análisis
ops.analysis('Static') # Análisis estático

# ===== APLICACIÓN DE ASENTAMIENTOS (RESTRICCIONES NO HOMOGÉNEAS) =====
# Los asentamientos se modelan como desplazamientos prescritos (no cero)
# Patrón para asentamientos (usando single-point constraints)
ops.timeSeries("Constant", 2) # Serie temporal constante para asentamientos
ops.pattern("Plain", 2, 2) # Patrón de carga asociado a la serie temporal 2

# Aplicar asentamiento en el nodo 6 (apoyo móvil)
# ops.sp(nodo, grado_de_libertad, valor): impone un desplazamiento prescrito
ops.sp(6, 2, asentamiento_6_y) # Nodo 6, GL 2 (Y), desplazamiento = -0.02 m

# Definir apoyos fijos (desplazamiento prescrito = 0)
ops.sp(1, 1, 0.0) # Nodo 1, GL 1 (X): restringido
ops.sp(1, 2, 0.0) # Nodo 1, GL 2 (Y): restringido
ops.sp(4, 1, 0.0) # Nodo 4, GL 1 (X): restringido
ops.sp(4, 2, 0.0) # Nodo 4, GL 2 (Y): restringido

# ===== APLICACIÓN DE CARGAS EXTERNAS =====
# Patrón para cargas externas
ops.timeSeries("Constant", 1) # Serie temporal constante para cargas
ops.pattern("Plain", 1, 1) # Patrón de carga asociado a la serie temporal 1

# Aplicar carga vertical en el nodo 7
# ops.load(nodo, Fx, Fy): aplica fuerzas nodales
ops.load(7, 0.0, -19.6133) # Fx = 0 kN, Fy = -19.6133 kN (hacia abajo)
```

```
# ===== EJECUCIÓN DEL ANÁLISIS =====
ops.analyze(1) # Realizar el análisis en un solo paso (estático lineal)

# ===== OBTENCIÓN DE RESULTADOS =====
print("\n=== RESUMEN DE RESULTADOS ===")

# 1. DESPLAZAMIENTOS NODALES
print("\n=== DESPLAZAMIENTOS EN LOS NODOS ===")
for i in range(1, 8):
    disp = ops.nodeDisp(i) # Obtener vector de desplazamientos [Dx, Dy]
    print(f"Node {i}: Dx = {disp[0]:.3e} m, Dy = {disp[1]:.3e} m")

# 2. FUERZAS AXIALES EN ELEMENTOS
print("\n=== FUERZAS AXIALES ===")
axial_forces = {} # Diccionario para almacenar fuerzas axiales por elemento

for element_data in elements:
    ele_id = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_f"]

    # Obtener coordenadas de los nodos
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Obtener fuerzas internas del elemento
    # eleForce retorna: [Fx_i, Fy_i, Fx_j, Fy_j] para elementos truss 2D
    forces = ops.eleForce(ele_id)
    force_x = forces[0] # Fuerza en X en nodo inicial
    force_y = forces[1] # Fuerza en Y en nodo inicial

    # Calcular vector de dirección del elemento
    dir_x = xf - xi
    dir_y = yf - yi

    # Calcular longitud del elemento
    length = np.sqrt(dir_x**2 + dir_y**2)

    # Calcular vector unitario de dirección
    unit_dir_x = dir_x / length
    unit_dir_y = dir_y / length

    # Calcular fuerza axial: proyección de la fuerza sobre la dirección del elemento
    # F_axial = F . u (producto punto)
    axial_force = force_x * unit_dir_x + force_y * unit_dir_y
    axial_forces[ele_id] = axial_force

    # Determinar tipo de esfuerzo
    # Convención OpenSees común: positivo = tracción, negativo = compresión
    tipo = "Tracción" if axial_force < 0 else "Compresión" if axial_force > 0 else "Fuerza
axial Nula"
    print(f"Elemento {ele_id}: {axial_force:+.2f} kN - {tipo}")
```

3. REACCIONES EN APOYOS

```
print("\n=== REACCIONES EN APOYOS ===")
ops.reactions() # Calcular reacciones (necesario después de usar constraints no
homogéneos)
```

Obtener reacciones en cada apoyo

```
R1_x = ops.nodeReaction(1, 1) # Reacción X en nodo 1
R1_y = ops.nodeReaction(1, 2) # Reacción Y en nodo 1
R4_x = ops.nodeReaction(4, 1) # Reacción X en nodo 4
R4_y = ops.nodeReaction(4, 2) # Reacción Y en nodo 4
R6_x = ops.nodeReaction(6, 1) # Reacción X en nodo 6
R6_y = ops.nodeReaction(6, 2) # Reacción Y en nodo 6
```

```
print(f"Apoyo 1 (Nodo 1): Fx = {R1_x:.2f} kN, Fy = {R1_y:.2f} kN")
print(f"Apoyo 2 (Nodo 4): Fx = {R4_x:.2f} kN, Fy = {R4_y:.2f} kN")
print(f"Apoyo 3 (Nodo 6): Fx = {R6_x:.2f} kN, Fy = {R6_y:.2f} kN")
```

4. VERIFICACIÓN DE EQUILIBRIO ESTÁTICO

```
print("\n=== VERIFICACIÓN DE EQUILIBRIO ===")
```

```
suma_Fx = R1_x + R4_x + R6_x # Sumatoria de fuerzas en X (todas las reacciones
horizontales)
suma_Fy = R1_y + R4_y + R6_y - 19.6133 # Sumatoria de fuerzas en Y (reacciones
verticales + carga externa)
suma_M4 = -R1_x*4 + R6_y*8 - 19.6133*12 # Sumatoria de momentos en el nodo 4 (reacciones
verticales + carga externa)
```

```
print(f"ΣFx = {suma_Fx:.6f} kN")
print(f"ΣFy = {suma_Fy:.6f} kN")
print(f"ΣM4 = {suma_M4:.6f} kN")
```

Verificar equilibrio (debe ser cercano a cero)

```
if abs(suma_Fx)<1e-6 and abs(suma_Fy)<1e-6 and abs(suma_M4)<1e-6:
    print("✓ Equilibrio verificado")
else:
    print("X Error en equilibrio")
```

===== GRÁFICAS ADICIONALES DE RESULTADOS =====

```
plt.figure(figsize=(6, 10))
```

===== GRÁFICA SISTEMA DEFORMADO =====

```
plt.subplot(2, 1, 1)
```

Dibujar sistema original (líneas discontinuas grises)

```
for element_data in elements:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_f"]

    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)
```

```
plt.plot([xi, xf], [yi, yf], 'k--', lw=1, alpha=0.5, label='Original' if
element_data["ID"] == 1 else "")

# Factor de amplificación para visualizar deformaciones
scale_factor = 10 # Los desplazamientos reales son muy pequeños

# Dibujar sistema deformado (amplificado)
for element_data in elements:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_f"]

    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Obtener desplazamientos reales
    disp_i = ops.nodeDisp(Ni)
    disp_f = ops.nodeDisp(Nf)

    # Calcular coordenadas deformadas (amplificadas)
    xi_def = xi + disp_i[0] * scale_factor
    yi_def = yi + disp_i[1] * scale_factor
    xf_def = xf + disp_f[0] * scale_factor
    yf_def = yf + disp_f[1] * scale_factor
    plt.plot([xi_def, xf_def], [yi_def, yf_def], 'g-', lw=2, label='Deformada' if
element_data["ID"] == 1 else "")

plt.xlabel('X (m)') plt.ylabel('Y (m)')
plt.title('Estructura Inicial vs Deformada')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.legend()

# ===== DIAGRAMA DE FUERZAS AXIALES =====
plt.subplot(2, 1, 2)

colors = [] # Asignar colores según el tipo de fuerza axial
for ele_id in range(1, 13):
    if axial_forces[ele_id] > 0:
        colors.append('blue') # Compresión (positivo)
    elif axial_forces[ele_id] < 0:
        colors.append('red') # Tracción (negativo)
    else:
        colors.append('orange') # Fuerza axial nula

# Graficar elementos con colores según fuerza axial
for i, element_data in enumerate(elements):
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_f"]
    xi, yi = ops.nodeCoord(Ni)
    xj, yj = ops.nodeCoord(Nf)
```

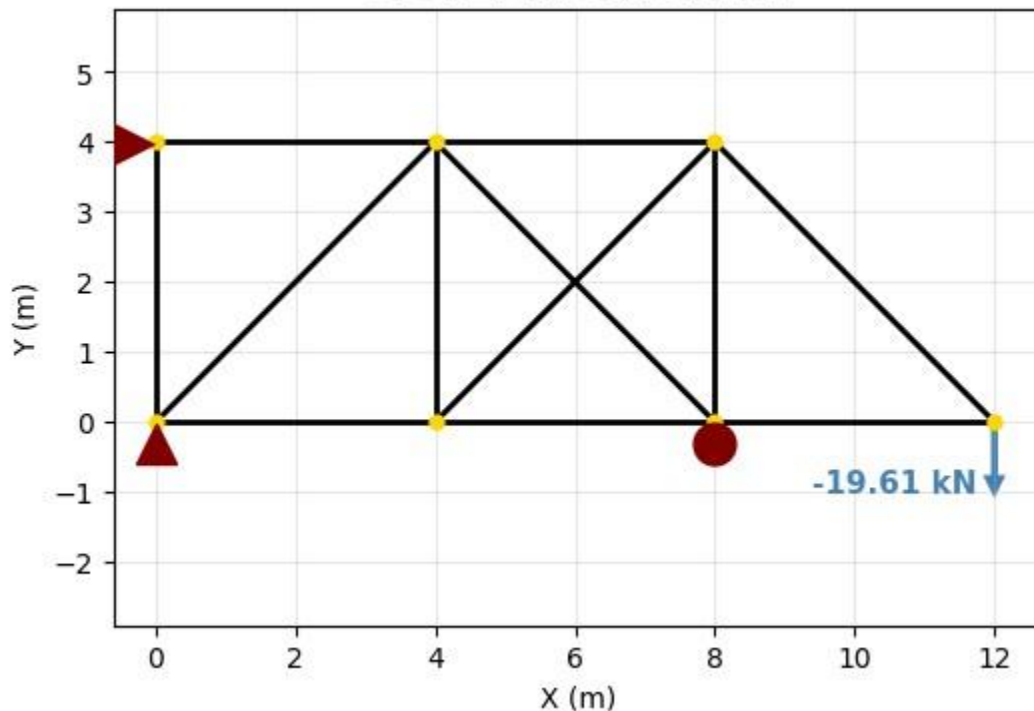
```
# Dibujar elemento con grosor y color según fuerza axial
plt.plot([xi, xj], [yi, yj], color=colors[i], linewidth=2, label=f'Ele {i+1}:
{axial_forces[i+1]:+.1f} kN')

# Dibujar nodos
for i in range(1, 8):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'ko', markersize=6)

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.title('Fuerzas Axiales (Rojo=Tracción, Azul=Compresión, Naranja=FA Nula)')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

MODELO EN OPENSEES: EJERCICIO 2 - ARMADURAS CON ASENTAMIENTOS EN APOYOS

SISTEMA ORIGINAL



=== RESUMEN DE RESULTADOS ===

=== DESPLAZAMIENTOS EN LOS NODOS ===

Nodo 1: $D_x = 0.000e+00$ m, $D_y = 0.000e+00$ m
 Nodo 2: $D_x = 4.021e-03$ m, $D_y = -9.337e-03$ m
 Nodo 3: $D_x = 5.055e-03$ m, $D_y = -1.949e-02$ m
 Nodo 4: $D_x = 0.000e+00$ m, $D_y = 0.000e+00$ m
 Nodo 5: $D_x = -2.141e-03$ m, $D_y = -1.011e-02$ m
 Nodo 6: $D_x = -3.510e-03$ m, $D_y = -2.000e-02$ m
 Nodo 7: $D_x = -3.772e-03$ m, $D_y = -2.905e-02$ m

=== FUERZAS AXIALES ===

Elemento 1: -301.56 kN - Tracción
 Elemento 2: -77.53 kN - Tracción
 Elemento 3: 160.58 kN - Compresión
 Elemento 4: 102.67 kN - Compresión
 Elemento 5: 19.61 kN - Compresión
 Elemento 6: -0.00 kN - Fuerza axial Nula
 Elemento 7: -57.92 kN - Tracción
 Elemento 8: -38.31 kN - Tracción
 Elemento 9: 199.36 kN - Compresión
 Elemento 10: -117.45 kN - Tracción
 Elemento 11: 81.91 kN - Compresión
 Elemento 12: -27.74 kN - Tracción

=== REACCIONES EN APOYOS ===

Apoyo 1 (Nodo 1): $F_x = -301.56 \text{ kN}$, $F_y = +0.00 \text{ kN}$

Apoyo 2 (Nodo 4): $F_x = +301.56 \text{ kN}$, $F_y = +140.97 \text{ kN}$

Apoyo 3 (Nodo 6): $F_x = +0.00 \text{ kN}$, $F_y = -121.36 \text{ kN}$

=== VERIFICACIÓN DE EQUILIBRIO ===

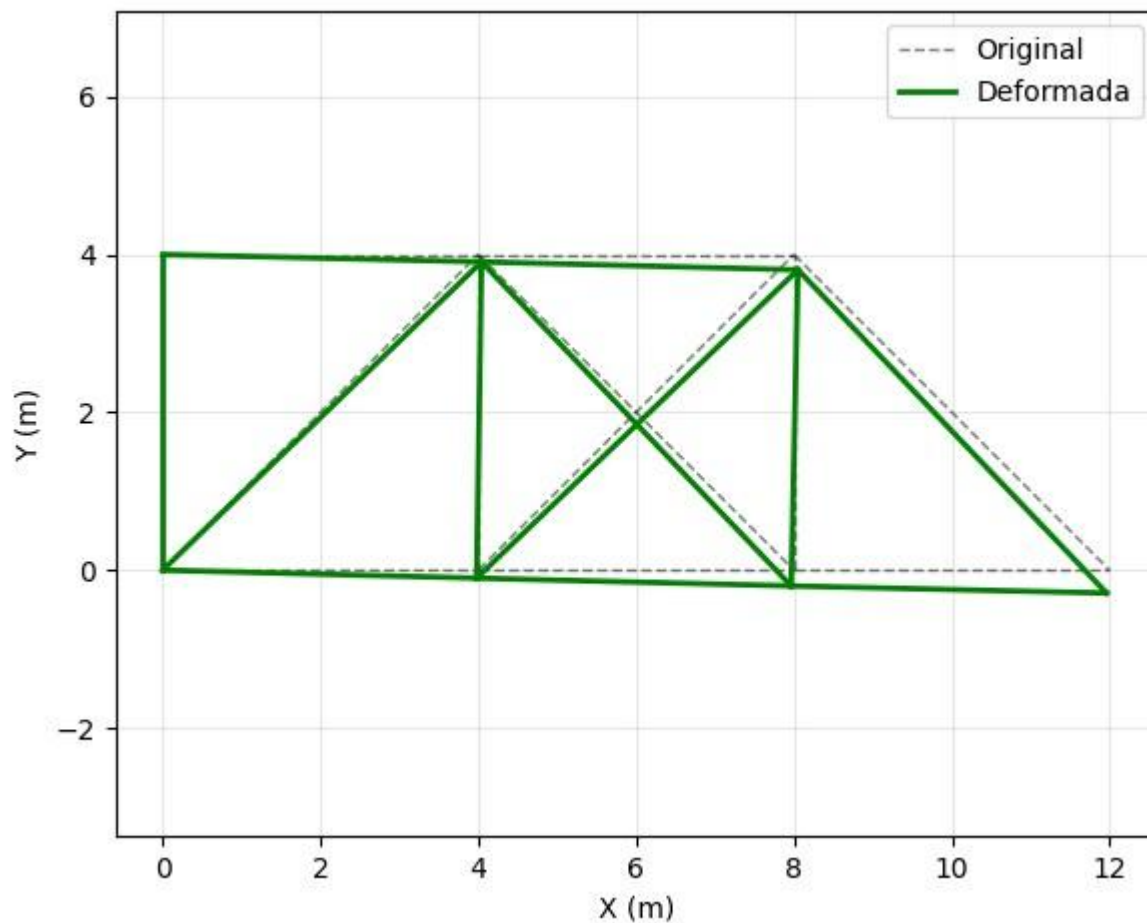
$\Sigma F_x = 0.000000 \text{ kN}$

$\Sigma F_y = -0.000000 \text{ kN}$

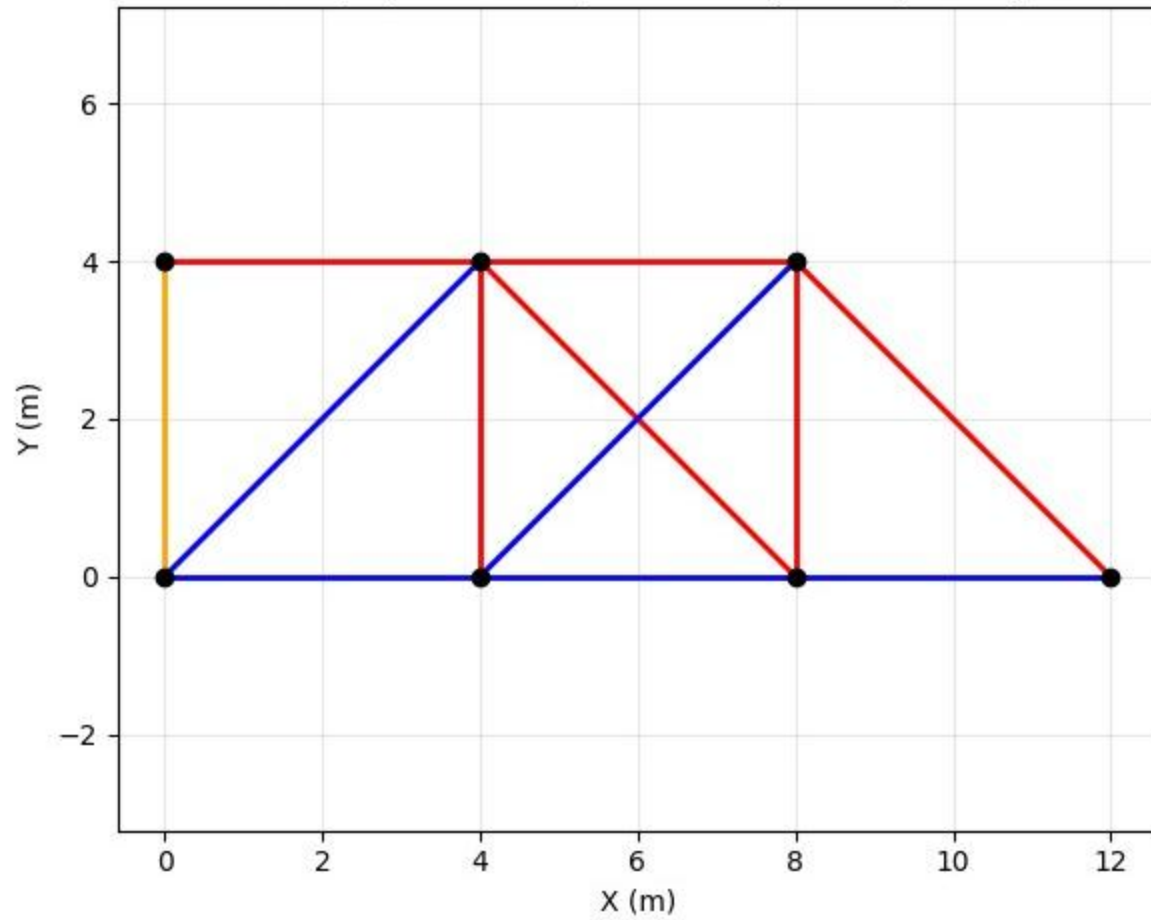
$\Sigma M_4 = -0.000000 \text{ kN}$

✓ Equilibrio verificado

Estructura Inicial vs Deformada



Fuerzas Axiales (Rojo=Tracción, Azul=Compresión, Naranja=FA Nula)



MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON ARMADURAS: EJERCICIO 3

```
# ===== IMPORTACIÓN DE BIBLIOTECAS =====
import numpy as np # Biblioteca para cálculos numéricos y operaciones matriciales
import math        # Biblioteca para funciones matemáticas básicas
import matplotlib.pyplot as plt # Biblioteca para visualización gráfica

print("MÉTODO MATRICIAL DE RIGIDEZ PARA ARMADURAS: EJERCICIO 3 (APOYOS ELÁSTICOS)")

# ===== DATOS DE ENTRADA =====
print("\n===== DATOS DE ENTRADA =====")
# Propiedades de los materiales y sección transversal
E = 2.1e7 # Módulo de elasticidad [Ton/m²]
A = ((5**2)-((5-1.6)**2))/(100**2) # Área transversal [m²]
L1 = ((0.8**2)+(1.0**2)**(1/2) # Longitud elementos diagonales [m]
a1 = math.atan(1/0.8)*(180/math.pi) # Ángulo de elementos diagonales [°]
L = np.array([0.8, 0.8, 0.8, 0.8, 1.0, 1.0, 1.0, L1, L1, L1, L1]) # Longitud de elementos [m]
a = np.array([0, 0, 0, 0, 90, 90, 90, a1, -a1, a1, -a1]) # Ángulo de elementos [°]
m = 11 # Número de elementos (barras)
n = 6 # Número de nodos
GL = 12 # Grados de libertad totales [6 Nodos x 2 gL/nodo = 12 GL]

print(f"Módulo de elasticidad: {E} Ton/m²")
print(f"Área sección transversal: {A:.3e} m²")
print(f"Longitud de los elementos: {np.round(L, 2)} m")
print(f"Ángulo de inclinación de los elementos: {np.round(a, 2)} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")

# ===== COORDENADAS DE NODOS =====
# Matriz de coordenadas [X, Y] para cada nodo (en metros)
coordenadas_nodos = np.array([
    [0.0, 0.0], # Nodo 1
    [0.8, 0.0], # Nodo 2
    [1.6, 0.0], # Nodo 3
    [0.0, 1.0], # Nodo 4
    [0.8, 1.0], # Nodo 5
    [1.6, 1.0]] # Nodo 6

# ===== CONECTIVIDAD DE ELEMENTOS =====
# Define qué nodos están conectados por cada elemento
Elementos = np.array([
    [1, 2], # Elemento 1: Nodo 1-2
    [2, 3], # Elemento 2: Nodo 2-3
    [4, 5], # Elemento 3: Nodo 4-5
    [5, 6], # Elemento 4: Nodo 5-6
    [1, 4], # Elemento 5: Nodo 1-4
    [2, 5], # Elemento 6: Nodo 4-2
```

```
[3, 6],      # Elemento 7: Nodo 3-6
[1, 5],      # Elemento 8: Nodo 1-5
[4, 2],      # Elemento 9: Nodo 4-2
[2, 6],      # Elemento 10: Nodo 2-6
[5, 3]]])    # Elemento 11: Nodo 5-3

# ===== GRADOS DE LIBERTAD POR ELEMENTO =====
# Mapeo de grados de libertad para cada elemento
#           E1 E2 E3 E4 E5 E6 E7 E8 E9 E10 E11
Nx = np.array([1, 3, 7, 9, 1, 3, 5, 1, 7, 3, 9]) # Dx GL inicial
Ny = np.array([2, 4, 8, 10, 2, 4, 6, 2, 8, 4, 10]) # Dy GL inicial
Fx = np.array([3, 5, 9, 11, 7, 9, 11, 9, 3, 11, 5]) # Dx GL final
Fy = np.array([4, 6, 10, 12, 8, 10, 12, 10, 4, 12, 6]) # Dy GL final

# ===== RIGIDECE DE APOYOS ELÁSTICOS =====
print("\n===== RIGIDECE DE APOYOS ELÁSTICOS =====")
# Vector de rigideces elásticas por grado de libertad [Ton/m]
#           GL:  1    2    3    4    5    6    7    8    9   10  11  12
K_elastica = np.array([8500, 5000, 8500, 5000, 8500, 5000, 0, 0, 0, 0, 0, 0])

# Mostrar rigideces elásticas por nodo
print("Rigideces elásticas por grado de libertad [Ton/m]:")
for i in range(n):
    gl_x = i * 2 # Grado de libertad en X (posiciones pares: 0, 2, 4, ...)
    gl_y = i * 2 + 1 # Grado de libertad en Y (posiciones impares: 1, 3, 5, ...)
    kx = K_elastica[gl_x]
    ky = K_elastica[gl_y]
    if kx>0 or ky>0:
        print(f"Nodo {i+1}: GL {gl_x+1}(X) = {kx} Ton/m, GL {gl_y+1}(Y) = {ky} Ton/m")

# ===== GRÁFICA SISTEMA ORIGINAL =====
# Crear gráfica de la estructura original
plt.figure(figsize=(9, 5))
plt.title('SISTEMA ORIGINAL', fontsize=14, fontweight='bold')

# Dibujar todos los elementos como líneas negras
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    # Convertir números de nodos en índices (restar 1 porque Python indexa desde 0)
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]
    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=2)

# Dibujar nodos con diferentes símbolos según si tienen apoyos elásticos
for i, (x, y) in enumerate(coordenadas_nodos):
    # Determinar grados de libertad de este nodo
    gl_x = i * 2
    gl_y = i * 2 + 1
    tiene_apoyo_elastico = K_elastica[gl_x]>0 or K_elastica[gl_y]>0
    if tiene_apoyo_elastico:
        plt.plot(x, y, 's', color='red', markersize=10, markerfacecolor='none',
        markeredgewidth=2) # Nodos con apoyos elásticos: cuadrado rojo vacío
```

```
# Mostrar valores de rigideces cerca del nodo
kx_text = f"Kx={K_elastica[gl_x]} Ton/m" if K_elastica[gl_x] > 0 else ""
ky_text = f"Ky={K_elastica[gl_y]} Ton/m" if K_elastica[gl_y] > 0 else ""
texto = f"{kx_text}\n{ky_text}".strip()
if texto:
    plt.text(x + 0.05, y + 0.05, texto, fontsize=8, color='red')
else:
    plt.plot(x, y, 'o', color='orange', markersize=7) # Nodos sin apoyos elásticos:
círculo naranja

# ===== DIBUJAR FUERZAS APLICADAS =====
# Nodo 4: 30 Ton horizontal (derecha) y 20 Ton vertical (abajo)
plt.arrow(-0.4, 1, 0.3, 0, head_width=0.05, head_length=0.1, fc='indigo', ec='indigo')
plt.text(-0.4, 1.05, '30 Ton', fontsize=10, color='indigo', fontweight='bold')
plt.arrow(0, 1.4, 0, -0.3, head_width=0.05, head_length=0.1, fc='indigo', ec='indigo')
plt.text(0.05, 1.2, '20 Ton', fontsize=10, color='indigo', fontweight='bold')

# Nodo 5: 20 Ton vertical (abajo)
plt.arrow(0.8, 1.4, 0, -0.3, head_width=0.05, head_length=0.1, fc='indigo', ec='indigo')
plt.text(0.8+0.05, 1.2, '20 Ton', fontsize=10, color='indigo', fontweight='bold')

# Nodo 6: 20 Ton vertical (abajo)
plt.arrow(1.6, 1.4, 0, -0.3, head_width=0.05, head_length=0.1, fc='indigo', ec='indigo')
plt.text(1.6+0.05, 1.2, '20 Ton', fontsize=10, color='indigo', fontweight='bold')

# Configurar aspecto del gráfico
plt.xlabel('X [m]')
plt.ylabel('Y [m]')
plt.grid(True, alpha=0.3)
plt.axis('equal') # Mantener relación de aspecto 1:1
plt.show()

# ===== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL =====
print("\n===== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL =====")
# Crear matriz global de rigidez (12x12 para 12 grados de libertad)
kG = np.zeros((GL, GL))

# Ensamblar matriz de rigidez elemento por elemento
for i in range(m):
    # Convertir ángulo de grados a radianes para funciones trigonométricas
    theta = math.radians(a[i])

    # Calcular coseno y seno del ángulo
    c = math.cos(theta)
    s = math.sin(theta)

    # Crear matriz de rigidez del elemento_i en coordenadas globales (4x4)
    kg_e = np.zeros((4, 4))

    # Factor constante para todos los términos de la matriz
    Factor = (A*E) / L[i]
```

```
# ===== CONSTRUIR MATRIZ DE RIGIDEZ DEL ELEMENTO =====
kg_e[0, 0] = Factor * c**2
kg_e[0, 1] = Factor * c * s
kg_e[0, 2] = -Factor * c**2
kg_e[0, 3] = -Factor * c * s

kg_e[1, 0] = Factor * c * s
kg_e[1, 1] = Factor * s**2
kg_e[1, 2] = -Factor * c * s
kg_e[1, 3] = -Factor * s**2

kg_e[2, 0] = -Factor * c**2
kg_e[2, 1] = -Factor * c * s
kg_e[2, 2] = Factor * c**2
kg_e[2, 3] = Factor * c * s

kg_e[3, 0] = -Factor * c * s
kg_e[3, 1] = -Factor * s**2
kg_e[3, 2] = Factor * c * s
kg_e[3, 3] = Factor * s**2

# ===== ENSAMBLAR EN MATRIZ GLOBAL =====
# Grados de libertad del elemento (restar 1 para convertir a índices de Python)
GL_elem = [Nx[i]-1, Ny[i]-1, Fx[i]-1, Fy[i]-1]

# Sumar la matriz del elemento a la matriz global en las posiciones correctas
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj]

# print(f"ELEMENTO {i+1}")
# print(f"Longitud: {L[i]} m, Ángulo: {a[i]}°")
# print(f"Matriz global del elemento: \n {np.round(kg_e, 0)}")
# print(f"Matriz global del sistema después del elemento {i+1}: \n {np.round(kG, 0)}")
print(f"Matriz de rigidez global de la estructura:\n {np.round(kG, 0)}")

# ===== MATRIZ DE APOYOS ELÁSTICOS =====
print("\n===== MATRIZ DE APOYOS ELÁSTICOS =====")
k_ap = np.zeros((GL, GL)) # Matriz diagonal con rigideces de apoyos

# Agregar rigideces elásticas a la diagonal de la matriz
for i in range(GL):
    if K_elastica[i] > 0:
        k_ap[i, i] += K_elastica[i] # Solo en la diagonal principal
print(f"Matriz de rigideces de apoyos elásticos:\n {np.round(k_ap, 0)}")

# Matriz de rigidez global incluyendo apoyos elásticos
kG_ap = kG + k_ap
```

```
# ===== DEFINICIÓN DE RESTRICCIONES =====
print("\n===== DEFINICIÓN DE RESTRICCIONES =====")
# Vector que indica qué grados de libertad están restringidos
# 0 = Libre (desplazamiento desconocido), 1 = Restringido (desplazamiento conocido)
#
#           GL: 1  2  3  4  5  6  7  8  9  10 11 12
restricciones = np.array([1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0])

# Mostrar estado de restricciones por nodo
for i in range(n):
    rtx = "[Restringido]" if restricciones[i*2] == 1 else "[Libre]"
    rty = "[Restringido]" if restricciones[i*2+1] == 1 else "[Libre]"
    print(f"Nodo {i+1}: Rx={restricciones[i*2]} {rtx}, Ry={restricciones[i*2+1]} {rty}")

# ===== VECTOR DE FUERZAS EXTERNAS =====
print("\n===== VECTOR DE FUERZAS EXTERNAS =====")
# Vector de fuerzas aplicadas en cada grado de libertad [Ton]
#           GL: 1  2  3  4  5  6  7  8  9  10 11 12
F = np.array([0, 0, 0, 0, 0, 0, 30, -20, 0, -20, 0, -20])
print(f"Vector de fuerzas externas: {F} Ton")

# ===== SOLUCIÓN DEL SISTEMA DE ECUACIONES =====
print("\n\n===== SOLUCIÓN DEL SISTEMA =====")
# Resolver el sistema de ecuaciones lineales: {F} = [K]{U}
# Calcular la inversa de la matriz de rigidez con apoyos elásticos
kG_ap_inv = np.linalg.inv(kG_ap)

# Calcular desplazamientos
U_totales = np.matmul(kG_ap_inv, F)

# ===== DESPLAZAMIENTOS EN LOS NODOS =====
print("\n=== DESPLAZAMIENTOS EN LOS NODOS ===")
for i in range(n):
    Ux = U_totales[i*2]      # Desplazamiento en X del nodo i+1
    Uy = U_totales[i*2+1]    # Desplazamiento en Y del nodo i+1
    print(f"Nodo {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m")

# ===== CÁLCULO DE REACCIONES =====
print("\n=== REACCIONES ===")
# Calcular las reacciones usando la matriz de rigidez original (sin apoyos elásticos)
Reacciones = np.matmul(kG, U_totales)

# Mostrar solo reacciones significativas (mayores que 1e-6)
for i in range(n):
    rx = Reacciones[i*2]
    ry = Reacciones[i*2+1]
    if abs(rx) > 1e-6 or abs(ry) > 1e-6:
        print(f"Nodo {i+1}: Rx = {rx:.3f} Ton, Ry = {ry:.3f} Ton")
```



```
# ===== CÁLCULO DE FUERZAS INTERNAS =====
print("\n=== FUERZAS AXIALES DE LOS ELEMENTOS ===")
# Vector para almacenar fuerzas axiales de cada elemento
F_elem = np.zeros(m)
for i in range(m):
    # Obtener desplazamientos de los nodos del elemento en coordenadas globales
    Ue = np.array([
        U_totales[Nx[i]-1], # Ux del nodo inicial
        U_totales[Ny[i]-1], # Uy del nodo inicial
        U_totales[Fx[i]-1], # Ux del nodo final
        U_totales[Fy[i]-1]] # Uy del nodo final

    # ===== CÁLCULO EN COORDENADAS LOCALES =====
    # Convertir ángulo a radianes
    theta = math.radians(a[i])
    c = math.cos(theta)
    s = math.sin(theta)

    # Factor constante para el elemento
    Factor = (A*E)/L[i]

    # Matriz de rigidez en coordenadas locales (2x2 para elemento axial)
    ke_loc = Factor * np.array([[1, -1], [-1, 1]])

    # Matriz de transformación de coordenadas
    Te = np.array([[c, s, 0, 0], [0, 0, c, s]])

    # Calcular fuerzas en coordenadas locales
    ke_loc_Te = np.matmul(ke_loc, Te) # k_local * T
    ke_loc_Te_Ue = np.matmul(ke_loc_Te, Ue) # (k_local * T) * U_global

    # La fuerza axial es la segunda componente del vector (fuerza en el nodo final)
    Fuerza_axial = ke_loc_Te_Ue[1]
    F_elem[i] = Fuerza_axial

    # Determinar tipo de fuerza axial: Tracción (+), Compresión (-)
    tipo = "Tracción" if Fuerza_axial > 0 else "Compresión" if Fuerza_axial < 0 else
"Fuerza axial Nula"
    print(f"Elemento {i+1}: {Fuerza_axial:.3f} Ton - {tipo}")

# ===== GRÁFICA SISTEMA DEFORMADO =====
FS = 10 # Factor de escala para amplificar desplazamientos (visualización)
plt.figure(figsize=(9, 5))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar elementos originales (no deformados) como líneas grises punteadas
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]
    plt.plot([xi, xf], [yi, yf], '--', color='gray', linewidth=2, alpha=0.5)
```

```
# Dibujar estructura deformada (amplificada)
for i, (nodo_i, nodo_f) in enumerate(Elementos):
    # Coordenadas originales de los nodos
    xi, yi = coordenadas_nodos[nodo_i-1]
    xf, yf = coordenadas_nodos[nodo_f-1]

    # Desplazamientos de los nodos (reales)
    desp_i = np.array([U_totales[(nodo_i-1)*2], U_totales[(nodo_i-1)*2+1]])
    desp_f = np.array([U_totales[(nodo_f-1)*2], U_totales[(nodo_f-1)*2+1]])

    # Coordenadas deformadas (amplificadas por factor FS)
    xi_def = xi + desp_i[0]*FS
    yi_def = yi + desp_i[1]*FS
    xf_def = xf + desp_f[0]*FS
    yf_def = yf + desp_f[1]*FS

    # Dibujar elemento deformado en verde
    plt.plot([xi_def, xf_def], [yi_def, yf_def], '-', color='green', linewidth=1)

# Marcar nodos con apoyos elásticos
for i, (x, y) in enumerate(coordenadas_nodos):
    gl_x = i * 2
    gl_y = i * 2 + 1
    if K_elastica[gl_x] > 0 or K_elastica[gl_y] > 0:
        plt.plot(x, y, 's', color='red', markersize=10, markerfacecolor='none',
        markeredgewidth=2)

plt.xlabel('X [m]')
plt.ylabel('Y [m]')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.show()
```

MÉTODO MATRICIAL DE RIGIDEZ PARA ARMADURAS: EJERCICIO 3 (APOYOS ELÁSTICOS)

===== DATOS DE ENTRADA =====

Módulo de elasticidad: 21000000.0 Ton/m²

Área sección transversal: 1.344e-03 m²

Longitud de los elementos: [0.8 0.8 0.8 0.8 1. 1. 1. 1.28 1.28 1.28 1.28] m

Ángulo de inclinación de los elementos: [0. 0. 0. 0. 90. 90. 90. 51.34 -51.34 51.34 -51.34] °

Número de elementos: 11

Número de nodos: 6

Número de grados de libertad: 12

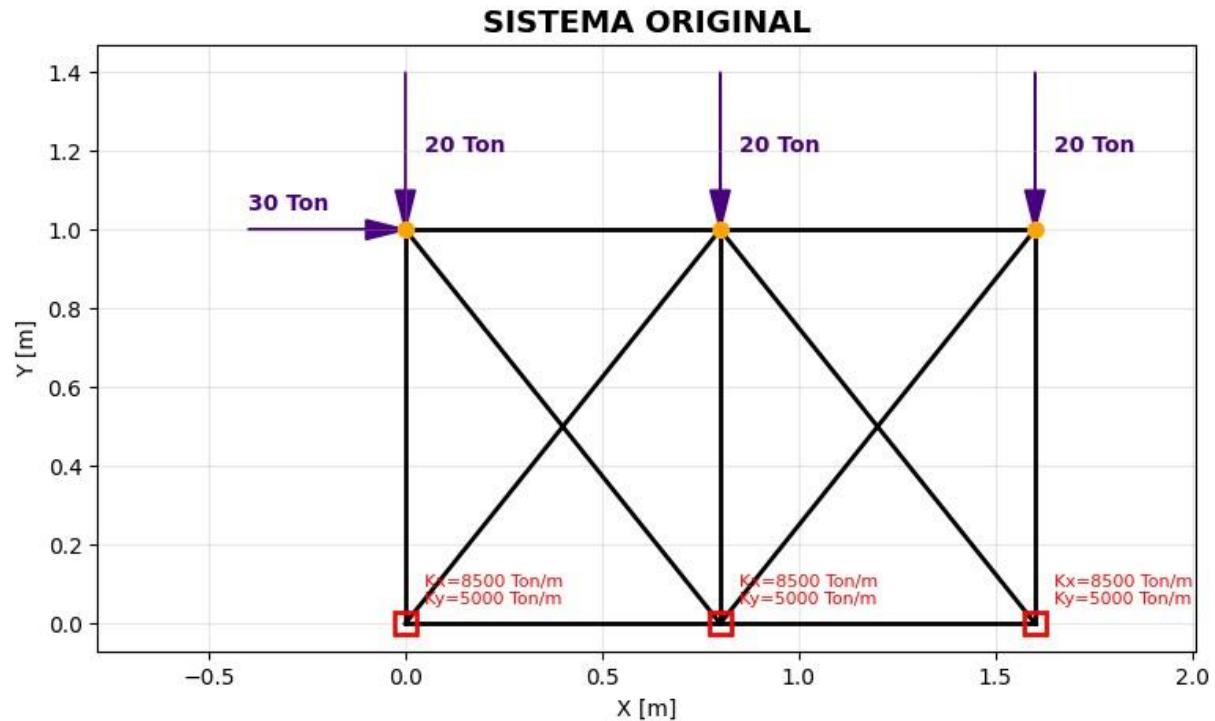
===== RIGIDECES DE APOYOS ELÁSTICOS =====

Rigideces elásticas por grado de libertad [Ton/m]:

Nodo 1: GL 1(X) = 8500 Ton/m, GL 2(Y) = 5000 Ton/m

Nodo 2: GL 3(X) = 8500 Ton/m, GL 4(Y) = 5000 Ton/m

Nodo 3: GL 5(X) = 8500 Ton/m, GL 6(Y) = 5000 Ton/m



===== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL =====

Matriz de rigidez global de la estructura:

```
[[ 43881. 10751. -35280.  0.  0.  0. -0. -0. -8601. -10751.  0.  0.]
 [ 10751. 41663.  0.  0.  0.  0. -0. -28224. -10751. -13439.  0.  0.]
 [-35280.  0. 87761.  0. -35280.  0. -8601. 10751. -0. -0. -8601. -10751.]
 [  0.  0.  0. 55101.  0.  0. 10751. -13439. -0. -28224. -10751. -13439.]
 [  0.  0. -35280.  0. 43881. -10751.  0.  0. -8601. 10751. -0. -0.]
 [  0.  0.  0.  0. -10751. 41663.  0.  0. 10751. -13439. -0. -28224.]
 [-0. -0. -8601. 10751.  0.  0. 43881. -10751. -35280.  0.  0.  0.]
 [-0. -28224. 10751. -13439.  0.  0. -10751. 41663.  0.  0.  0.  0.]
 [-8601. -10751. -0. -0. -8601. 10751. -35280.  0. 87761.  0. -35280.  0.]
 [-10751. -13439. -0. -28224. 10751. -13439.  0.  0.  0. 55101.  0.  0.]
 [  0.  0. -8601. -10751. -0. -0.  0.  0. -35280.  0. 43881. 10751.]
 [  0.  0. -10751. -13439. -0. -28224.  0.  0.  0.  0. 10751. 41663.]]
```

===== MATRIZ DE APOYOS ELÁSTICOS =====

Matriz de rigideces de apoyos elásticos:

```
[[8500.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0. 5000.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0. 8500.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0. 5000.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0. 8500.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0. 5000.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]
 [  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.]]
```

===== DEFINICIÓN DE RESTRICCIONES =====

Nodo 1: Rx=1 [Restringido], Ry=1 [Restringido]
 Nodo 2: Rx=1 [Restringido], Ry=1 [Restringido]
 Nodo 3: Rx=1 [Restringido], Ry=1 [Restringido]
 Nodo 4: Rx=0 [Libre], Ry=0 [Libre]
 Nodo 5: Rx=0 [Libre], Ry=0 [Libre]
 Nodo 6: Rx=0 [Libre], Ry=0 [Libre]

===== VECTOR DE FUERZAS EXTERNAS =====

Vector de fuerzas externas: [0 0 0 0 0 0 30 -20 0 -20 0 -20] Ton

===== SOLUCIÓN DEL SISTEMA =====

=== DESPLAZAMIENTOS EN LOS NODOS ===

Nodo 1: Ux = 1.053e-03 m, Uy = -1.703e-04 m
 Nodo 2: Ux = 1.207e-03 m, Uy = -4.159e-03 m
 Nodo 3: Ux = 1.270e-03 m, Uy = -7.670e-03 m
 Nodo 4: Ux = 7.329e-03 m, Uy = -3.573e-04 m
 Nodo 5: Ux = 6.812e-03 m, Uy = -4.448e-03 m
 Nodo 6: Ux = 6.765e-03 m, Uy = -8.452e-03 m

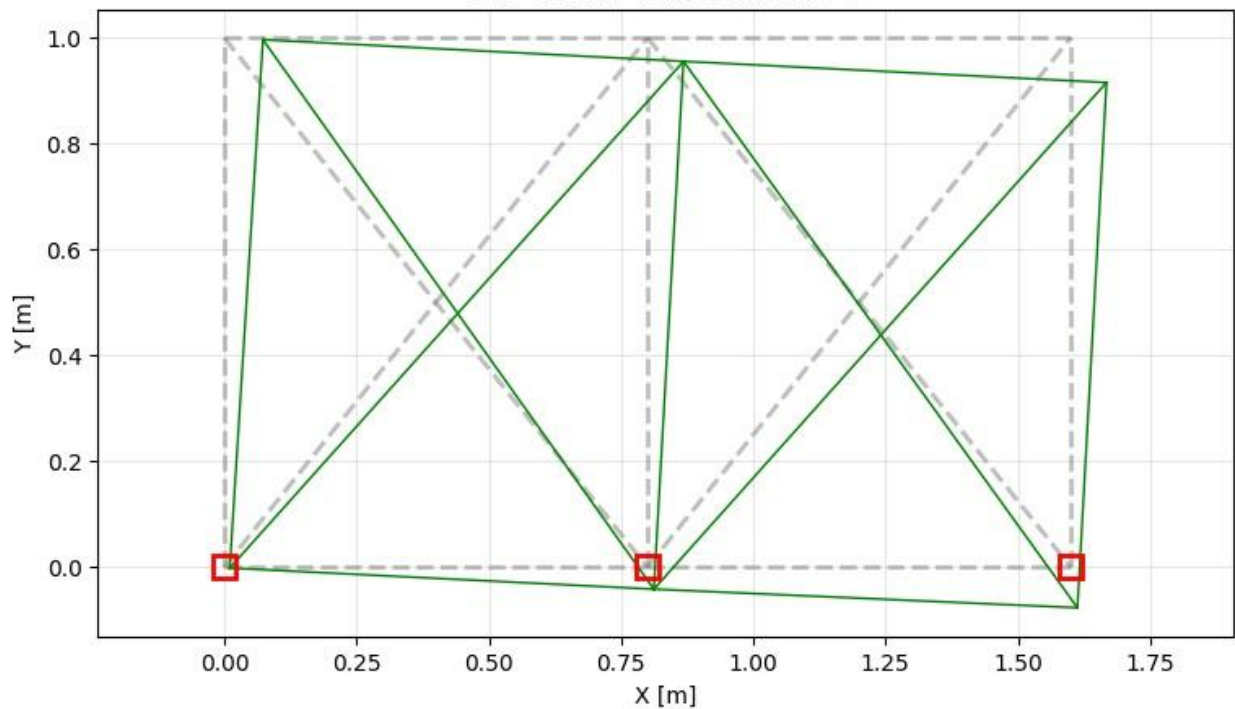
=== REACCIONES ===

Nodo 1: $R_x = -8.952$ Ton, $R_y = 0.851$ Ton
Nodo 2: $R_x = -10.255$ Ton, $R_y = 20.797$ Ton
Nodo 3: $R_x = -10.793$ Ton, $R_y = 38.351$ Ton
Nodo 4: $R_x = 30.000$ Ton, $R_y = -20.000$ Ton
Nodo 5: $R_x = -0.000$ Ton, $R_y = -20.000$ Ton
Nodo 6: $R_x = 0.000$ Ton, $R_y = -20.000$ Ton

=== FUERZAS AXIALES DE LOS ELEMENTOS ===

Elemento 1: 5.410 Ton - Tracción
Elemento 2: 2.232 Ton - Tracción
Elemento 3: -18.222 Ton - Compresión
Elemento 4: -1.656 Ton - Compresión
Elemento 5: -5.278 Ton - Compresión
Elemento 6: -8.145 Ton - Compresión
Elemento 7: -22.070 Ton - Compresión
Elemento 8: 5.669 Ton - Tracción
Elemento 9: -18.853 Ton - Compresión
Elemento 10: 2.651 Ton - Tracción
Elemento 11: -20.850 Ton - Compresión

SISTEMA DEFORMADO



MODELO OPENSEES

ARMADURAS: EJERCICIO 3

```
# ===== IMPORTACIÓN DE BIBLIOTECAS =====
import openseespy.opensees as ops # Biblioteca principal para análisis estructural OpenSees
import numpy as np                # Biblioteca para cálculos numéricos y matemáticos
import matplotlib.pyplot as plt   # Biblioteca para visualización y gráficos

print("MODELO EN OPENSEES PARA EL EJERCICIO 3 DE ARMADURAS (APOYOS ELÁSTICOS)")

# ===== GENERACIÓN DEL MODELO =====
ops.wipe() # Eliminar cualquier modelo existente en memoria

ops.model('basic', '-ndm', 2, '-ndf', 2) # Crear un nuevo modelo básico en 2D con 2 grados
de libertad por nodo (X, Y)

# ===== PROPIEDADES DEL MATERIAL =====
# Unidades utilizadas: Toneladas (fuerza) y metros (Longitud)
A = 0.001344 # Área transversal de los elementos en m²
E = 21000000 # Módulo de elasticidad del material en Ton/m²

# ===== CREACIÓN DE NODOS =====
# Definir coordenadas de los 6 nodos principales de la armadura
ops.node(1, 0.0, 0.0) # Nodo 1
ops.node(2, 0.8, 0.0) # Nodo 2
ops.node(3, 1.6, 0.0) # Nodo 3
ops.node(4, 0.0, 1.0) # Nodo 4
ops.node(5, 0.8, 1.0) # Nodo 5
ops.node(6, 1.6, 1.0) # Nodo 6

# ===== DEFINICIÓN DE MATERIALES =====
ops.uniaxialMaterial('Elastic', 1, E) # Material elástico para los elementos de la armadura

# ===== DEFINICIÓN DE APOYOS ELÁSTICOS =====
# Rigideces de los resortes que simulan apoyos elásticos
k_spring_x = 8500 # Rigidez en dirección horizontal (Ton/m)
k_spring_y = 5000 # Rigidez en dirección vertical (Ton/m)

# Crear materiales separados para resortes en X e Y
ops.uniaxialMaterial('Elastic', 2, k_spring_x) # Resorte en dirección X
ops.uniaxialMaterial('Elastic', 3, k_spring_y) # Resorte en dirección Y

# ===== CREACIÓN DE ELEMENTOS DE LA ARMADURA =====
elements = [] # Lista para almacenar información de todos los elementos

ops.element("Truss", 1, 1, 2, A, 1) # Elemento 1: Nodos 1-2
elements.append({"ID": 1, "Nodo_i": 1, "Nodo_j": 2})

ops.element("Truss", 2, 2, 3, A, 1) # Elemento 2: Nodos 2-3
elements.append({"ID": 2, "Nodo_i": 2, "Nodo_j": 3})
```

```
ops.element("Truss", 3, 4, 5, A, 1) # Elemento 3: Nodos 4-5
elements.append({"ID": 3, "Nodo_i": 4, "Nodo_j": 5})

ops.element("Truss", 4, 5, 6, A, 1) # Elemento 4: Nodos 5-6
elements.append({"ID": 4, "Nodo_i": 5, "Nodo_j": 6})

ops.element("Truss", 5, 1, 4, A, 1) # Elemento 5: Nodos 1-4
elements.append({"ID": 5, "Nodo_i": 1, "Nodo_j": 4})

ops.element("Truss", 6, 2, 5, A, 1) # Elemento 6: Nodos 2-5
elements.append({"ID": 6, "Nodo_i": 2, "Nodo_j": 5})

ops.element("Truss", 7, 3, 6, A, 1) # Elemento 7: Nodos 3-6
elements.append({"ID": 7, "Nodo_i": 3, "Nodo_j": 6})

ops.element("Truss", 8, 1, 5, A, 1) # Elemento 8: Nodos 1-5
elements.append({"ID": 8, "Nodo_i": 1, "Nodo_j": 5})

ops.element("Truss", 9, 4, 2, A, 1) # Elemento 9: Nodos 4-2
elements.append({"ID": 9, "Nodo_i": 4, "Nodo_j": 2})

ops.element("Truss", 10, 2, 6, A, 1) # Elemento 10: Nodos 2-6
elements.append({"ID": 10, "Nodo_i": 2, "Nodo_j": 6})

ops.element("Truss", 11, 5, 3, A, 1) # Elemento 11: Nodos 5-3
elements.append({"ID": 11, "Nodo_i": 5, "Nodo_j": 3})

# ===== CREACIÓN DE NODOS PARA APOYOS ELÁSTICOS =====
# Nodos adicionales que representan los puntos de anclaje fijos de los resortes
ops.node(101, 0.0, 0.0) # Nodo base para resorte en nodo 1
ops.node(102, 0.8, 0.0) # Nodo base para resorte en nodo 2
ops.node(103, 1.6, 0.0) # Nodo base para resorte en nodo 3

# ===== ELEMENTOS ZEROLENGTH PARA APOYOS ELÁSTICOS =====
# Elementos zeroLength conectan nodos base (fijos) con nodos estructurales
# Estos elementos tienen longitud cero y simulan resortes
ops.element('zeroLength', 101, 101, 1, '-mat', 2, 3, '-dir', 1, 2)
elements.append({"ID": 101, "Nodo_i": 101, "Nodo_j": 1, "type": "spring"})

ops.element('zeroLength', 102, 102, 2, '-mat', 2, 3, '-dir', 1, 2)
elements.append({"ID": 102, "Nodo_i": 102, "Nodo_j": 2, "type": "spring"})

ops.element('zeroLength', 103, 103, 3, '-mat', 2, 3, '-dir', 1, 2)
elements.append({"ID": 103, "Nodo_i": 103, "Nodo_j": 3, "type": "spring"})

# ===== CONDICIONES DE APOYO =====
# Fijar completamente los nodos base de los resortes (anclajes rígidos)
ops.fix(101, 1, 1) # Nodo base del resorte 1: fijo en X e Y
ops.fix(102, 1, 1) # Nodo base del resorte 2: fijo en X e Y
ops.fix(103, 1, 1) # Nodo base del resorte 3: fijo en X e Y
```

```
# ===== APLICACIÓN DE CARGAS =====
ops.timeSeries("Linear", 1) # Serie temporal lineal para aplicar cargas gradualmente
ops.pattern("Plain", 1, 1) # Patrón de carga estática asociado a la serie temporal 1
# Aplicar cargas verticales hacia abajo (-Y) en los nodos superiores
ops.load(4, 30.0, -20.0) # Nodo 4: 30 Ton → (derecha), 20 Ton ↓ (abajo)
ops.load(5, 0.0, -20.0) # Nodo 5: 0 Ton →, 20 Ton ↓
ops.load(6, 0.0, -20.0) # Nodo 6: 0 Ton →, 20 Ton ↓

# ===== CONFIGURACIÓN DEL ANÁLISIS ESTÁTICO =====
ops.system('BandSPD') # Solucionador: Banda Simétrica Positiva Definida (eficiente para
grandes sistemas)
ops.numberer('RCM') # Renumeración: Reverse Cuthill-McKee (optimiza ancho de banda de la
matriz)
ops.constraints('Plain') # Manejador de restricciones estándar
ops.integrator('LoadControl', 1.0) # Control de carga: aplicar 100% de la carga en un
paso
ops.algorithm('Linear') # Algoritmo lineal (adecuado para análisis elástico lineal)
ops.analysis('Static') # Análisis estático

# ===== CONFIGURACIÓN DE RECORDERS (GUARDAR RESULTADOS) =====
# Recorder para desplazamientos nodales (se guardan en archivo)
ops.recorder('Node', '-file', 'desplazamientos.out', '-time', '-node', 1, 2, 3, 4, 5, 6,
'-dof', 1, 2, 'disp')

# Recorder para fuerzas en los resortes
ops.recorder('Element', '-file', 'fuerzas_resortes.out', '-time', '-ele', 101, 102, 103,
'globalForce')

# ===== EJECUCIÓN DEL ANÁLISIS =====
ops.analyze(1) # Realizar análisis estático en un paso (carga completa)

# ===== VISUALIZACIÓN DE LA GEOMETRÍA Y CARGAS =====
plt.figure(figsize=(7, 5))

# Graficar todos los elementos
for element_data in elements:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Diferenciar elementos estructurales (negro) de resortes (rojo punteado)
    if element_data.get("type") == "spring":
        # Elementos zeroLength (resortes) se dibujan en rojo punteado
        plt.plot([xi, xf], [yi, yf], 'r--', lw=2, alpha=0.7)
    else:
        # Elementos truss (barras de la armadura) se dibujan en negro continuo
        plt.plot([xi, xf], [yi, yf], 'k-', lw=2)
```

```

# Graficar nodos principales de la estructura (círculos azules)
for i in range(1, 7):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'bo', markersize=8, label='Nodo estructural' if i == 1 else "")

# Graficar nodos base de los resortes (cuadrados rojos)
for i in [101, 102, 103]:
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'rs', markersize=6, label='Apoyo elástico' if i == 101 else "")

# ===== DIBUJAR CARGAS APLICADAS =====
# Carga horizontal en el nodo 4
plt.arrow(ops.nodeCoord(4)[0]-0.4, ops.nodeCoord(4)[1], 0.3, 0, head_width=0.05,
head_length=0.1, fc='purple', ec='purple')
plt.text(ops.nodeCoord(4)[0]-0.4, ops.nodeCoord(4)[1]+0.05, '30 Ton', fontsize=10,
color='indigo', fontweight='bold')

# Carga horizontal en el nodo 4
plt.arrow(ops.nodeCoord(4)[0], ops.nodeCoord(4)[1]+0.4, 0, -0.3, head_width=0.05,
head_length=0.1, fc='purple', ec='purple')
plt.text(ops.nodeCoord(4)[0]+0.05, ops.nodeCoord(4)[1]+0.2, '20 Ton', fontsize=10,
color='indigo', fontweight='bold')

# Carga vertical en el nodo 5
plt.arrow(ops.nodeCoord(5)[0], ops.nodeCoord(5)[1]+0.4, 0, -0.3, head_width=0.05,
head_length=0.1, fc='purple', ec='purple')
plt.text(ops.nodeCoord(5)[0]+0.05, ops.nodeCoord(5)[1]+0.2, '20 Ton', fontsize=10,
color='indigo', fontweight='bold')

# Carga vertical en el nodo 6
plt.arrow(ops.nodeCoord(6)[0], ops.nodeCoord(6)[1]+0.4, 0, -0.3, head_width=0.05,
head_length=0.1, fc='purple', ec='purple', label='Carga aplicada')
plt.text(ops.nodeCoord(6)[0]+0.05, ops.nodeCoord(6)[1]+0.2, '20 Ton', fontsize=10,
color='indigo', fontweight='bold')

# Configurar aspecto del gráfico
plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.title('Estructura con Apoyos Elásticos')
plt.grid(True, alpha=0.3)
plt.axis('equal') # Mantener relación de aspecto 1:1
plt.legend()
plt.show()

# ===== PRESENTACIÓN DE RESULTADOS =====
print("\n=== RESUMEN DE RESULTADOS ===")

# Desplazamientos nodales
print("\n=== DESPLAZAMIENTOS EN LOS NODOS ===")
for node_id in range(1, 7):
    disp = ops.nodeDisp(node_id)
    print(f"Nodo {node_id}: Ux = {disp[0]:.3e} m, Uy = {disp[1]:.3e} m")

```

```
# ===== CÁLCULO DE FUERZAS AXIALES EN BARRAS =====
print("\n=== FUERZAS AXIALES ===")
axial_forces = {} # Diccionario para almacenar fuerzas axiales por elemento

# Calcular fuerza axial para cada elemento truss (excluyendo resortes)
for element_data in elements:
    if element_data.get("type") != "spring": # Solo procesar elementos truss
        # Obtener coordenadas de los nodos
        ele_id = element_data["ID"]
        Ni = element_data["Nodo_i"]
        Nf = element_data["Nodo_j"]
        xi, yi = ops.nodeCoord(Ni)
        xf, yf = ops.nodeCoord(Nf)

        # Obtener fuerzas internas del elemento
        forces = ops.eleForce(ele_id)
        force_x = forces[0] # Fuerza en dirección X en el nodo inicial
        force_y = forces[1] # Fuerza en dirección Y en el nodo inicial

        # Calcular vector de dirección del elemento
        dir_x = xf - xi
        dir_y = yf - yi

        # Calcular longitud del elemento (módulo del vector)
        length = np.sqrt(dir_x**2 + dir_y**2)

        # Normalizar el vector de dirección (vector unitario)
        unit_dir_x = dir_x / length
        unit_dir_y = dir_y / length

        # Calcular fuerza axial: proyección de la fuerza sobre la dirección del elemento
        axial_force = force_x * unit_dir_x + force_y * unit_dir_y
        axial_forces[ele_id] = axial_force

        # Convención: positivo = compresión, negativo = tracción
        tipo = "Tracción" if axial_force < 0 else "Compresión" if axial_force > 0 else
"Fuerza Axial Nula"

        # Mostrar resultado
        print(f"Elemento {ele_id}: {axial_force:.3f} Ton - {tipo}")

# ===== REACCIONES EN LOS APOYOS ELÁSTICOS =====
print("\n=== REACCIONES EN APOYOS ELÁSTICOS ===")
ops.reactions() # Calcular reacciones en los apoyos

# Obtener reacciones en los nodos base de los resortes
R101_x = ops.nodeReaction(101, 1)
R101_y = ops.nodeReaction(101, 2)
R102_x = ops.nodeReaction(102, 1)
R102_y = ops.nodeReaction(102, 2)
R103_x = ops.nodeReaction(103, 1)
R103_y = ops.nodeReaction(103, 2)
```



```
print(f"Reacción en apoyo elástico 1 (Nodo 101): Rx = {R101_x:.2f} Ton, Ry = {R101_y:.2f} Ton")
print(f"Reacción en apoyo elástico 2 (Nodo 102): Rx = {R102_x:.2f} Ton, Ry = {R102_y:.2f} Ton")
print(f"Reacción en apoyo elástico 3 (Nodo 103): Rx = {R103_x:.2f} Ton, Ry = {R103_y:.2f} Ton")

# ===== FUERZAS EN LOS RESORTES =====
print("\n=== FUERZAS EN ELEMENTOS ELÁSTICOS ===")
for element_data in elements:
    if element_data.get("type") == "spring":
        ele_id = element_data["ID"]
        forces = ops.eleForce(ele_id)
        print(f"Fuerzas en resorte {ele_id}: Fx = {forces[0]:.2f} Ton, Fy = {forces[1]:.2f} Ton")

# ===== VERIFICACIÓN DE EQUILIBRIO ESTÁTICO =====
# Sumar todas las fuerzas en X y Y para verificar equilibrio
suma_fx = R101_x + R102_x + R103_x + 30.0 # Incluye carga horizontal de 30 Ton
suma_fy = R101_y + R102_y + R103_y - 20.0 - 20.0 - 20.0 # Incluye cargas verticales
suma_M1 = R102_y*0.8 + R103_y*1.6 - 30.0*1 - 20.0*0.8 - 20.0*1.6

print(f"\n=== VERIFICACIÓN DE EQUILIBRIO ===")
print(f"Suma Fx = {suma_fx:.6f} Ton")
print(f"Suma Fy = {suma_fy:.6f} Ton")
print(f"Suma M1 = {suma_M1:.6f} Ton")

# Verificar si las sumatorias son cercanas a cero (dentro de tolerancia)
if abs(suma_fx)<1e-6 and abs(suma_fy)<1e-6 and abs(suma_M1)<1e-6:
    print("✓ Equilibrio verificado")
else:
    print("X Error en equilibrio")

# ===== GRÁFICAS ADICIONALES =====
plt.figure(figsize=(10, 10))

# === SUBPLOT 1: ESTRUCTURA ORIGINAL VS DEFORMADA ===
plt.subplot(2, 1, 1)

# Dibujar estructura original (líneas punteadas grises)
for element_data in elements:
    ele_id = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    plt.plot([xi, xf], [yi, yf], 'k--', lw=1, alpha=0.5, label='Original' if element_data["ID"] == 1 else "")

scale_factor = 10 # Factor de amplificación para visualizar deformaciones
```



```
# Dibujar estructura deformada (amplificada)
for element_data in elements:
    ele_id = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Obtener desplazamientos reales de los nodos
    disp_i = ops.nodeDisp(Ni)    disp_f = ops.nodeDisp(Nf)

    # Calcular coordenadas deformadas (amplificadas)
    xi_def = xi + disp_i[0] * scale_factor
    yi_def = yi + disp_i[1] * scale_factor
    xf_def = xf + disp_f[0] * scale_factor
    yf_def = yf + disp_f[1] * scale_factor

    # Dibujar elemento deformado en verde continuo
    plt.plot([xi_def, xf_def], [yi_def, yf_def], 'g-', lw=2, label='Deformada' if
element_data["ID"] == 1 else "")

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.title('Estructura Inicial vs Deformada')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.legend()

# === SUBPLOT 2: DIAGRAMA DE FUERZAS AXIALES ===
plt.subplot(2, 1, 2)

# Filtrar solo elementos truss (excluir resortes)
truss_elements = [elem for elem in elements if elem.get("type") != "spring"]

# Asignar colores según tipo de fuerza axial
colors = []
for ele_id in range(1, 12):
    if axial_forces[ele_id] > 0:
        colors.append('blue')    # Compresión (positivo)
    elif axial_forces[ele_id] < 0:
        colors.append('red')     # Tracción (negativo)
    else:
        colors.append('orange')  # Fuerza axial cero

# Graficar elementos truss con colores según fuerza axial
for i, element_data in enumerate(truss_elements):
    ele_id = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)
```



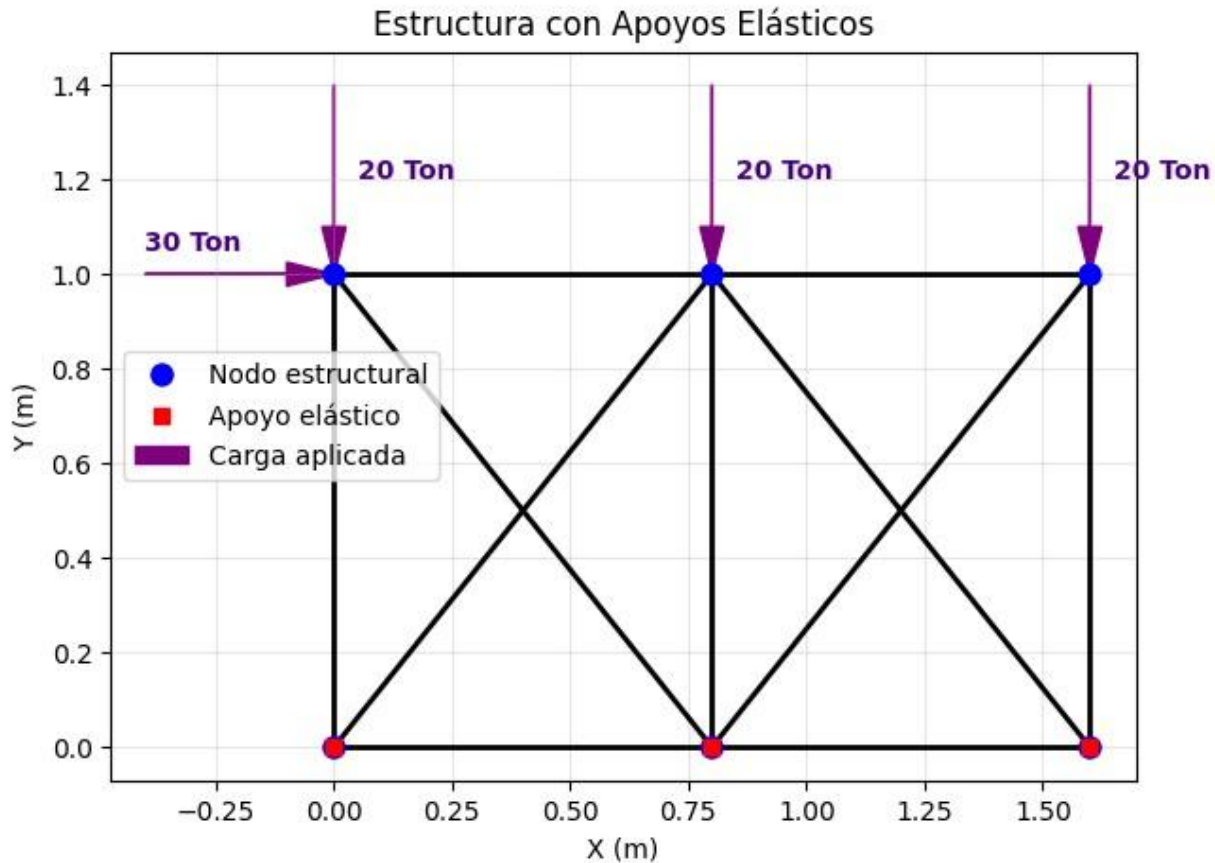
```
# Etiquetar solo los primeros 3 elementos para evitar superposición en Leyenda
plt.plot([xi, xf], [yi, yf], color=colors[i], linewidth=2, label=f'Ele {ele_id}: {axial_forces[ele_id]:+.1f} Ton' if i < 3 else "")
```

```
# Dibujar nodos principales
```

```
for i in range(1, 7):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'ko', markersize=6)
```

```
plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.title('Fuerzas Axiales (Rojo=Tracción, Azul=Compresión, Naranja=FA Nula)')
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

MODELO EN OPENSEES PARA EL EJERCICIO 3 DE ARMADURAS (APOYOS ELÁSTICOS)



=== RESUMEN DE RESULTADOS ===

=== DESPLAZAMIENTOS EN LOS NODOS ===

Nodo 1: $U_x = 1.053e-03$ m, $U_y = -1.703e-04$ m
 Nodo 2: $U_x = 1.207e-03$ m, $U_y = -4.159e-03$ m
 Nodo 3: $U_x = 1.270e-03$ m, $U_y = -7.670e-03$ m
 Nodo 4: $U_x = 7.329e-03$ m, $U_y = -3.573e-04$ m
 Nodo 5: $U_x = 6.812e-03$ m, $U_y = -4.448e-03$ m
 Nodo 6: $U_x = 6.765e-03$ m, $U_y = -8.452e-03$ m

=== FUERZAS AXIALES ===

Elemento 1: -5.410 Ton - Tracción
 Elemento 2: -2.232 Ton - Tracción
 Elemento 3: 18.222 Ton - Compresión
 Elemento 4: 1.656 Ton - Compresión
 Elemento 5: 5.278 Ton - Compresión
 Elemento 6: 8.145 Ton - Compresión
 Elemento 7: 22.070 Ton - Compresión
 Elemento 8: -5.669 Ton - Tracción
 Elemento 9: 18.853 Ton - Compresión
 Elemento 10: -2.651 Ton - Tracción
 Elemento 11: 20.850 Ton - Compresión

=== REACCIONES EN APOYOS ELÁSTICOS ===

Reacción en apoyo elástico 1 (Nodo 101): $R_x = -8.95 \text{ Ton}$, $R_y = 0.85 \text{ Ton}$

Reacción en apoyo elástico 2 (Nodo 102): $R_x = -10.26 \text{ Ton}$, $R_y = 20.80 \text{ Ton}$

Reacción en apoyo elástico 3 (Nodo 103): $R_x = -10.79$ Ton, $R_y = 38.35$ Ton

=== FUERZAS EN ELEMENTOS ELÁSTICOS ===

Fuerzas en resorte 101: $F_x = -8.95 \text{ Ton}$, $F_y = 0.85 \text{ Ton}$

Fuerzas en resorte 102: $F_x = -10.26 \text{ Ton}$, $F_y = 20.80 \text{ Ton}$

Fuerzas en resorte 103: $F_x = -10.79$ Ton, $F_y = 38.35$ Ton

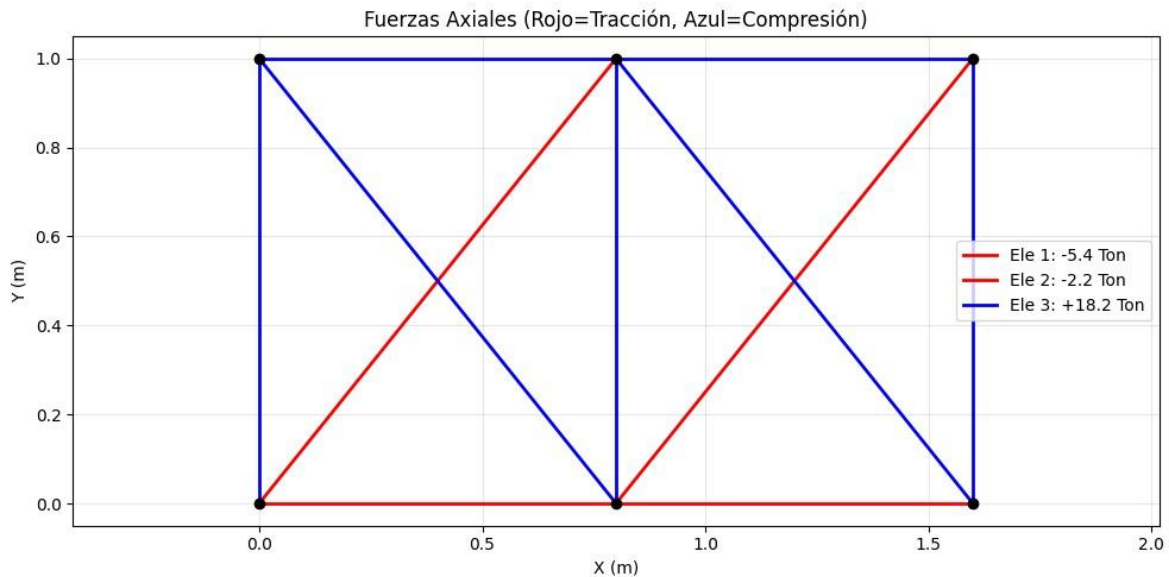
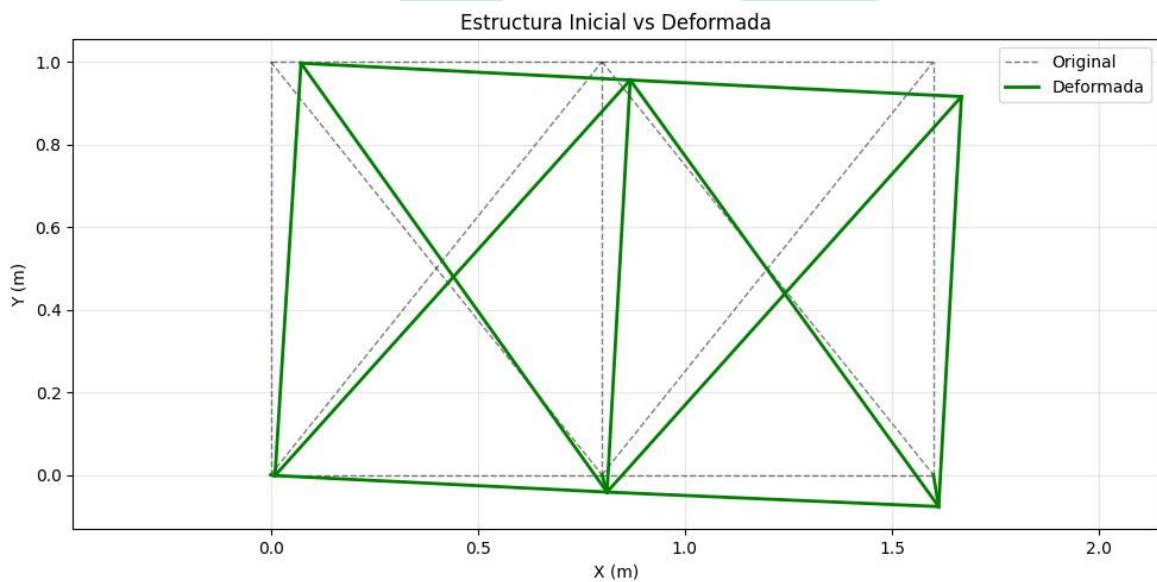
=== VERIFICACIÓN DE EQUILIBRIO ===

Suma Fx = -0.000000 Ton

Suma $F_y = 0.000000$ Ton

Suma M1 = 0.000000 Ton

✓ Equilibrio verificado



Pórticos

MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON PÓRTICOS: EJERCICIO 1

```
# =====
# IMPORTAR LIBRERÍAS
# =====
import numpy as np
import math
import matplotlib.pyplot as plt

print("MÉTODO MATRICIAL DE RIGIDEZ PARA PÓRTICOS - EJERCICIO 1")

# =====
# DATOS DE ENTRADA
# =====
print("\n=== DATOS DE ENTRADA ===")
E = 24870062.324 # Módulo de elasticidad (kN/m²)
A = np.array([0.09, 0.09, 0.09, 0.09, 0.105, 0.105]) # Áreas transversales (m²)
L = np.array([4, 3, 3, 4, 5, 5]) # Longitudes de elementos (m)
I = np.array([0.000675, 0.000675, 0.000675, 0.000675, 0.001071875, 0.001071875]) #
Momentos de inercia (m⁴)
a = np.array([90, 90, 90, 90, 0, 0]) # Ángulos de inclinación (°)
m = 6 # Número de elementos
n = 6 # Número de nodos
GL = n * 3 # Grados de Libertad totales (18: 6 nodos × 3 GL)

print(f"Módulo de elasticidad: {E} kN/m²")
print(f"Área de la sección transversal: {A} m²")
print(f"Longitud: {np.round(L, 2)} m")
print(f"Inercia: {np.round(I, 4)} m⁴")
print(f"Ángulo de inclinación: {np.round(a, 2)} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")

# =====
# COORDENADAS DE LOS NODOS
# =====
# Matriz de coordenadas nodales en el plano X-Y (unidades: metros)
# Cada fila corresponde a las coordenadas [x, y] de un nodo
coordenadas_nodos = np.array([
    [0, 0], # Nodo 1
    [0, 4], # Nodo 2
    [0, 7], # Nodo 3
    [5, 7], # Nodo 4
    [5, 4], # Nodo 5
    [5, 0]]) # Nodo 6
```

```
# =====
# CONECTIVIDAD DE LOS ELEMENTOS
# =====
# Cada fila define la conectividad de un elemento: [nodo_inicial, nodo_final]
# Los elementos 1-4 son las columnas verticales, 5-6 son las vigas horizontales
Elementos = np.array([
    [1, 2], # Elemento 1
    [2, 3], # Elemento 2
    [5, 4], # Elemento 3
    [6, 5], # Elemento 4
    [3, 4], # Elemento 5
    [2, 5]]) # Elemento 6

# =====
# NUMERACIÓN DE GRADOS DE LIBERTAD
# =====
# Se asigna un número único a cada grado de libertad (GL) del sistema
# - Nodo INICIAL: Dx = Nx[i], Dy = Ny[i], θz = Nz[i]
# - Nodo FINAL:   Dx = Fx[i], Dy = Fy[i], θz = Fz[i]
#
#      E1  E2  E3  E4  E5  E6
Nx = np.array([1, 4, 13, 16, 7, 4]) # GL Dx en nodo inicial
Ny = np.array([2, 5, 14, 17, 8, 5]) # GL Dy en nodo inicial
Nz = np.array([3, 6, 15, 18, 9, 6]) # GL θz en nodo inicial
Fx = np.array([4, 7, 10, 13, 10, 13]) # GL Dx en nodo final
Fy = np.array([5, 8, 11, 14, 11, 14]) # GL Dy en nodo final
Fz = np.array([6, 9, 12, 15, 12, 15]) # GL θz en nodo final

# =====
# DEFINICIÓN DE CARGAS APLICADAS
# =====
# Formato: [tipo, dirección, color, magnitud] para cargas distribuidas
Cargas = [[], [], [], []], # Elemento 1 al 4: sin carga
          ['Uniforme', 'Local_y', 'orange', -40], # Elemento 5: -40 kN/m (uniforme hacia abajo)
          ['Uniforme', 'Local_y', 'slateblue', -50]] # Elemento 6: -50 kN/m (uniforme hacia
abajo)

# =====
# PREPROCESAMIENTO GEOMÉTRICO DE TODOS LOS ELEMENTOS
# =====
# Calcula propiedades geométricas fundamentales para cada elemento
geom = [] # Lista para almacenar los datos

for e in range(m):
    ni, nf = Elementos[e][0], Elementos[e][1]
    xi, yi = coordenadas_nodos[ni-1]
    xf, yf = coordenadas_nodos[nf-1]
    theta = math.radians(a[e])
```

```
# Vector director unitario (apunta del nodo inicial al final)
ex = np.cos(theta)
ey = np.sin(theta)

# Vector normal unitario (perpendicular, rotado 90° antihorario)
nx = -ey
ny = ex

Le = L[e]
geom.append([ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny])

# =====
# GRÁFICA SISTEMA ORIGINAL
# =====

plt.figure(figsize=(7, 5.6))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos (barras)
for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=3, zorder=1)

# Dibujar nodos (puntos naranjas)
for i, (x, y) in enumerate(coordenadas_nodos):
    plt.plot(x, y, 'o', color='orange', markersize=10, zorder=2)

# Dibujar apoyos empotrados (cuadrados marrones)
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=2,
label='Empotramiento')
plt.plot(5, -0.15, 's', color='maroon', markersize=20, zorder=2)

# --- Cargas distribuidas ---
escala_visual = 0.03 # Factor de escala para visualización de flechas (solo efecto gráfico)

for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]

    # Saltar elementos sin carga
    if not Cargas[i]:
        continue

    # Extraer información de carga
    tipo, direccion, color, w = Cargas[i]

    # Puntos para dibujar flechas a lo largo del elemento
    t_vals = np.linspace(0, 1, 20)
```



```
x_superior = []
y_superior = []

for t in t_vals:
    # Punto a lo largo del elemento (coordenadas globales)
    x = xi + t * (xf - xi)
    y = yi + t * (yf - yi)

    # Para vigas horizontales con carga vertical (Local_y)
    # Las flechas se dibujan perpendiculares al elemento (dirección nx, ny)
    x_base = x + escala_visual * -w * nx
    y_base = y + escala_visual * -w * ny

    # Dibujar flecha individual
    plt.arrow(x_base, y_base, x - x_base, y - y_base, head_width=0.15,
              head_length=0.2, fc=color, ec=color, length_includes_head=True,
              zorder=3)

    x_superior.append(x_base)
    y_superior.append(y_base)

    # Dibujar línea que conecta las puntas de las flechas (contorno de la carga)
    plt.plot(x_superior, y_superior, '--', color=color, linewidth=2, zorder=3)

# Etiquetas de cargas distribuidas
plt.text(1.7, 8.3, '-40 kN/m', color='orange', fontsize=11, fontweight='bold',
         zorder=4)
plt.text(1.7, 5.7, '-50 kN/m', color='slateblue', fontsize=11, fontweight='bold',
         zorder=4)

# Cargas puntuales horizontales (flechas grises)
plt.arrow(-1.7, 4, 1.5, 0, head_width=0.1, head_length=0.2, fc='grey', ec='grey',
          linewidth=3, zorder=4)
plt.text(-1.5, 4.2, '150 kN', color='grey', fontweight='bold', fontsize=11,
         zorder=4)

plt.arrow(-1.7, 7, 1.5, 0, head_width=0.1, head_length=0.2, fc='grey', ec='grey',
          linewidth=3, zorder=4)
plt.text(-1.5, 7.2, '120 kN', color='grey', fontweight='bold', fontsize=11,
         zorder=4)

# Configuración final del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.show()
```

```
# =====
# ENSAMBLE DE LA MATRIZ DE RIGIDEZ GLOBAL
# =====
print("\n=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA ===")
kG = np.zeros((GL, GL)) # Inicializar matriz de rigidez global (18x18)

# Listas para almacenar matrices de cada elemento (para post-procesamiento)
kL_elementos = [] # Matrices Locales (6x6) en coordenadas Locales del elemento
T_elementos = [] # Matrices de transformación (6x6) de global a Local

for i in range(m): # Para cada elemento (0 a 5)
    # Parámetros de transformación
    theta = math.radians(a[i]) # Convertir ángulo a radianes
    c = math.cos(theta) # Coseno del ángulo
    s = math.sin(theta) # Seno del ángulo

    # Factores de rigidez (pre-calculados para eficiencia)
    AE = A[i] * E # Rigidez axial (EA)
    EI = E * I[i] # Rigidez flexional (EI)
    L2 = (L[i])**2 # Longitud al cuadrado
    L3 = (L[i])**3 # Longitud al cubo

    # Matriz de rigidez Local (6x6) en coordenadas Locales del elemento
    kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
          [0, (12*EI)/L3, (6*EI)/L2, 0, -(12*EI)/L3, (6*EI)/L2],
          [0, (6*EI)/L2, (4*EI)/L[i], 0, -(6*EI)/L2, (2*EI)/L[i]],
          [-AE/L[i], 0, 0, AE/L[i], 0, 0],
          [0, -(12*EI)/L3, -(6*EI)/L2, 0, (12*EI)/L3, -(6*EI)/L2],
          [0, (6*EI)/L2, (2*EI)/L[i], 0, -(6*EI)/L2, (4*EI)/L[i]]]
    kL_elementos.append(kL)

    # Matriz de transformación de coordenadas
    T = [[c, s, 0, 0, 0, 0],
          [-s, c, 0, 0, 0, 0],
          [0, 0, 1, 0, 0, 0],
          [0, 0, 0, c, s, 0],
          [0, 0, 0, -s, c, 0],
          [0, 0, 0, 0, 0, 1]]
    T_elementos.append(T)

    # Transformar matriz LOCAL a GLOBAL:  $K_{global} = T^T \cdot K_{local} \cdot T$ 
    T_T = np.transpose(T) # Transformación inversa (Local a global)
    kg_e = np.matmul(np.matmul(T_T, kL), T) # Matriz de rigidez del elemento en
    coordenadas globales
```

```
# Ensamblar en matriz global
# Mapeo: GL del elemento → posición en matriz global (índices 0-based)
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]

# Sumar contribución del elemento dentro de la matriz del sistema
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj]

#print(f"ELEMENTO {i+1}")
#print(f"Longitud: {L[i]} m, Ángulo: {a[i]:.3f}°")
#print(f"Matriz global del elemento: \n {np.round(kg_e, 0)}")
#print(f"Matriz global del sistema después del elemento {i+1}: \n {np.round(kG, 0)}")
print(f"Matriz global del sistema: \n {np.round(kG, 0)}")

# =====
# VECTOR DE FUERZAS DE EMPOTRAMIENTO PERFECTO (FEM)
# =====
print("\n=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===")
FEM_L_elementos = [] # FEM en coordenadas locales por elemento
FEM_G = np.zeros(GL) # FEM en coordenadas globales (ensamblado)

for i in range(m):
    # Verificar si el elemento tiene carga distribuida
    if not Cargas[i]:
        FEM_L = [0, 0, 0, 0, 0, 0]

    elif Cargas[i][0] == 'Uniforme' and Cargas[i][1] == 'Local_y':
        # Carga uniforme en dirección Y local (perpendicular al elemento)
        # Se cambia el signo porque la carga está definida como negativa en Cargas
        w = -Cargas[i][3] # Magnitud positiva de la carga hacia abajo

        # Fórmulas analíticas para viga biempotrada con carga uniforme
        # Convención: momentos positivos en sentido antihorario en el nodo
        Ryi = (w * L[i]) / 2 # Reacción vertical en nodo inicial
        Mzi = (w * (L[i]**2)) / 12 # Momento en nodo inicial (antihorario +)
        Ryf = (w * L[i]) / 2 # Reacción vertical en nodo final
        Mzf = -(w * (L[i]**2)) / 12 # Momento en nodo final (horario [-])

        FEM_L = [0, Ryi, Mzi, 0, Ryf, Mzf] # Vector FEM Local

    else:
        FEM_L = [0, 0, 0, 0, 0, 0]

FEM_L_elementos.append(FEM_L) # Almacenar en la lista el vector FEM Local
```

```
# Transformar FEM Local a global:  $FEM_{global} = T^T \cdot FEM_{Local}$ 
FEM_Ge = np.matmul(np.transpose(T_elementos[i]), FEM_L)

# Ensamblar en vector global
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]
for jj, gl_j in enumerate(GL_elem):
    FEM_G[gl_j] += FEM_Ge[jj] print(np.round(FEM_G, 2))

# =====
# CONDICIONES DE CONTORNO (RESTRICCIONES)
# =====
print("\n\n=== RESTRICCIONES ===")
# Vector de restricciones: 1 = restringido (desplazamiento conocido = 0)
#                               0 = libre (desplazamiento desconocido)
#                               GL: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
restricciones = np.array([1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1])

for i in range(n):
    rx = "Restringido" if restricciones[i*3] == 1 else "Libre"
    ry = "Restringido" if restricciones[i*3+1] == 1 else "Libre"
    rz = "Restringido" if restricciones[i*3+2] == 1 else "Libre"

    print(f"Node {i+1}: Dx={restricciones[i*3]} ({rx}), Dy={restricciones[i*3+1]} ({ry}), Dz={restricciones[i*3+2]} ({rz})")

# =====
# VECTOR DE FUERZAS EXTERNAS
# =====
print("\n\n=== VECTOR DE FUERZAS EXTERNAS ===")
# Fuerzas nodales aplicadas directamente (cargas puntuales en nodos)
#                               GL: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
F = np.array([0, 0, 0, 150, 0, 0, 120, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
print(f"Fuerzas externas: {F} kN")

# =====
# REDUCCIÓN DEL SISTEMA
# =====
print("\n\n=== REDUCCIÓN DEL SISTEMA ===")
# Identificar grados de libertad activos (no restringidos)
GL_activos = np.where(restricciones == 0)[0]
n_GL_activos = len(GL_activos)

print(f"Grados de libertad activos: {n_GL_activos}")
print(f"Índices: {GL_activos + 1}") # +1 para mostrar numeración en base 1

# Extraer submatrices correspondientes a GL activos
K_reducida = kG[np.ix_(GL_activos, GL_activos)] # Matriz de rigidez reducida
```

```
F_reducido = F[GL_activos]           # Vector de fuerzas reducido
FEM_reducido = FEM_G[GL_activos]     # Vector FEM reducido

print(f"\nMatriz global reducida:\n {np.round(K_reducida, 0)}")
print(f"\nVector de fuerzas externas reducido: {F_reducido} kN")
print(f"\nVector FEM reducido: {np.round(FEM_reducido, 2)} kN")

# =====
# SOLUCIÓN DEL SISTEMA
# =====
print("\n\n==== SOLUCIÓN DEL SISTEMA =====")
# Resolver sistema lineal:  $K \cdot U = F - FEM$  #  $U = K^{-1} \cdot (F - FEM)$  (desplazamientos desconocidos)
K_reducida_inv = np.linalg.inv(K_reducida)
U_desconocidos = np.matmul(K_reducida_inv, F_reducido - FEM_reducido)

# Reconstruir vector completo de desplazamientos
U_totales = np.zeros(GL)
U_totales[GL_activos] = U_desconocidos

# Calcular reacciones en apoyos:  $R = K \cdot U + FEM$ 
Reacciones = np.matmul(kG, U_totales) + FEM_G

print("\n=== REACCIONES ===")
# Mostrar reacciones solo en nodos con restricciones
for i in range(n):
    if np.any(restricciones[i*3:i*3+3] == 1):
        Rx = Reacciones[i*3]
        Ry = Reacciones[i*3+1]
        Mz = Reacciones[i*3+2]
        print(f"Node {i+1}: Rx = {Rx:.2f} kN, Ry = {Ry:.2f} kN, Mz = {Mz:.2f} kN-m")

print("\n=== DESPLAZAMIENTOS ===")
# Desplazamientos nodales (notación científica para valores pequeños)
for i in range(n):
    Ux = U_totales[i*3]
    Uy = U_totales[i*3+1]
    Tz = U_totales[i*3+2]
    print(f"Node {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m, Tz = {Tz:.3e} rad")

# =====
# GRÁFICA SISTEMA DEFORMADO
# =====
FS = 10 # Factor de escala para amplificar visualmente los desplazamientos

plt.figure(figsize=(6, 6))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')
```

```
# Dibujar sistema original (líneas punteadas grises) como referencia
for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '--', color='grey',
             linewidth=2, alpha=0.5, label='Original' if i == 0 else "")

# Dibujar sistema deformado (líneas verdes) con desplazamientos amplificados
for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]

    # Desplazamientos de los nodos (del vector solución)
    desp_i = np.array([U_totales[(ni-1)*3], U_totales[(ni-1)*3+1], U_totales[(ni-1)*3+2]])
    desp_f = np.array([U_totales[(nf-1)*3], U_totales[(nf-1)*3+1], U_totales[(nf-1)*3+2]])

    # Coordenadas deformadas (original + desplazamiento * factor de escala)
    xi_def = xi + desp_i[0] * FS
    yi_def = yi + desp_i[1] * FS
    xf_def = xf + desp_f[0] * FS
    yf_def = yf + desp_f[1] * FS

    # Dibujar elemento deformado
    plt.plot([xi_def, xf_def], [yi_def, yf_def], '-',
             color='g', linewidth=2, label='Deformada' if i == 0 else "")

# Dibujar empotramientos en posición original (referencia)
plt.plot(0, -0.3, 's', color='maroon', markersize=20, zorder=2) plt.plot(5, -0.3,
's', color='maroon', markersize=20, zorder=2)

# Configuración del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.5)
plt.axis('equal')
plt.tight_layout()
plt.show()

print("\n=== FUERZAS INTERNAS ===")
# Cálculo de fuerzas internas en los extremos de cada elemento
F_int_elementos = [] # Lista para almacenar fuerzas internas de todos los elementos
for i in range(m):
    # Extraer desplazamientos globales del elemento
    Ue = np.array([
        U_totales[Nx[i]-1], # Dx nodo inicial
        U_totales[Ny[i]-1], # Dy nodo inicial
        U_totales[Nz[i]-1], # Dz nodo inicial
```

```

    U_totales[Fx[i]-1], # Dx nodo final
    U_totales[Fy[i]-1], # Dy nodo final
    U_totales[Fz[i]-1])) # Dz nodo final

# Transformar desplazamientos a coordenadas Locales:  $U_{local} = T \cdot U_{global}$ 
Ue_L = np.matmul(T_elementos[i], Ue)

# Calcular fuerzas internas en coordenadas Locales:  $F = K_{local} \cdot U_{local} + FEM_{local}$ 
Fe_L = np.matmul(kL_elementos[i], Ue_L) + FEM_L_elementos[i]
F_int_elementos.append(Fe_L)

# Presentar resultados
print(f"Elemento {i+1}:")
print(f"  Nodo {Elementos[i,0]}: N = {Fe_L[0]:.3f} kN, V = {Fe_L[1]:.3f} kN, M = {Fe_L[2]:.3f} kN-m")
print(f"  Nodo {Elementos[i,1]}: N = {Fe_L[3]:.3f} kN, V = {Fe_L[4]:.3f} kN, M = {Fe_L[5]:.3f} kN-m\n")

# =====
# DIAGRAMA DE FUERZA AXIAL
# =====
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA DE FUERZA AXIAL (kN)", fontsize=14, fontweight="bold")
escala_axial = 0.01 # Factor de escala: controla el tamaño visual del diagrama

for i in range(m):
    # Extraer información geométrica del elemento i
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]

    # Dibujar la línea del elemento (en gris claro como referencia)
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Extraer fuerzas axiales en extremos (del cálculo de fuerzas internas)
    Ni = F_int_elementos[i][0] # Axial en nodo inicial (kN)
    Nf = F_int_elementos[i][3] # Axial en nodo final (kN)

    # Verificar si hay carga distribuida axial (en dirección local x)
    if len(Cargas[i]) > 0 and Cargas[i][0] == 'Uniforme' and Cargas[i][1] == 'Local_x':
        w = Cargas[i][3] # Magnitud de la carga axial distribuida (kN/m)

    else:
        w = 0 # Sin carga axial distribuida

    # Puntos de evaluación a lo largo del elemento (30 puntos para curva suave)
    x = np.linspace(0, Le, 30) # Coordenada local a lo largo del elemento (m)

```



```
# Cálculo de la fuerza axial variable N(x) integrando la carga distribuida axial
#  $N(x) = \int w_{axial}(x) dx + N_i$ 
N = w * x + Ni

# Coordenadas para dibujar el diagrama:
# - Se parte del punto sobre el elemento (xi + ex*x)
# - Se desplaza perpendicularmente (dirección nx, ny) según la magnitud de N
X_diag = xi + ex * x + escala_axial * N * nx
Y_diag = yi + ey * x + escala_axial * N * ny

# Línea base del elemento (para el relleno)
X_base = xi + ex * x
Y_base = yi + ey * x

# Dibujar la línea del diagrama y relleno con color semitransparente
plt.plot(X_diag, Y_diag, '-', linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:-1]]), np.concatenate([Y_base,
Y_diag[:-1]]), alpha=0.4)

# Texto en el punto medio indicando magnitud y tipo de esfuerzo
xm = xi + ex * Le / 2
ym = yi + ey * Le / 2
tipo = "Tracción" if Nf > 0 else "Compresión"

plt.text(xm, ym, f"{Nf:.2f} kN\n({tipo})", fontsize=8, ha='center',
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# =====
# DIAGRAMA DE FUERZA CORTANTE
# =====
escala_cortante = 0.008 # Factor de escala para visualización
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA DE FUERZA CORTANTE (kN)", fontsize=14, fontweight="bold")

for e in range(m):
    # Extraer información geométrica del elemento e
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[e]

    # Dibujar la línea del elemento como referencia
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)
```




```
# Extraer fuerzas cortantes en extremos
Vi = F_int_elementos[e][1] # Cortante en nodo inicial (kN)
Vf = F_int_elementos[e][4] # Cortante en nodo final (kN)

# Verificar si hay carga distribuida transversal (en dirección local y)
if len(Cargas[e]) > 0 and Cargas[e][0] == 'Uniforme' and Cargas[e][1]=='Local_y':
    w = Cargas[e][3] # Magnitud de la carga transversal distribuida

else:
    w = 0

# Puntos de evaluación a lo largo del elemento
x = np.linspace(0, Le, 30) # Coordenada local (m)

# Para carga uniforme, el cortante varía linealmente:  $V(x) = w \cdot x + V_i$ 
V = w * x + Vi

# Coordenadas para dibujar el diagrama (desplazamiento perpendicular)
X_diag = xi + ex * x + escala_cortante * V * nx
Y_diag = yi + ey * x + escala_cortante * V * ny

# Línea base del elemento
X_base = xi + ex * x
Y_base = yi + ey * x

# Dibujar diagrama con relleno
plt.plot(X_diag, Y_diag, '-', linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:, -1]]), np.concatenate([Y_base,
Y_diag[:, -1]]), alpha=0.4)

# Etiquetas con valores en extremos
plt.text(X_diag[0], Y_diag[0], f"{Vi:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.text(X_diag[-1], Y_diag[-1], f"{-Vf:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none')) # Signo
negativo por convención gráfica

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
# =====
# DIAGRAMA DE MOMENTO FLECTOR
# =====
escala_momento = 0.005 # Factor de escala para visualización (generalmente el más
pequeño)
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA DE MOMENTO FLECTOR (kN-m)", fontsize=14, fontweight="bold")

for e in range(m):
    # Extraer información geométrica del elemento e
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[e]

    # Extraer fuerzas internas necesarias
    Vi = F_int_elementos[e][1] # Cortante en nodo inicial (kN)
    Mi = F_int_elementos[e][2] # Momento en nodo inicial (kN-m)
    Mf = F_int_elementos[e][5] # Momento en nodo final (kN-m)

    # Dibujar línea del elemento como referencia
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Verificar si hay carga distribuida transversal
    if len(Cargas[e]) > 0 and Cargas[e][0] == 'Uniforme' and Cargas[e][1] == 'Local_y':
        w = Cargas[e][3] # Magnitud de la carga transversal (kN/m)

    else:
        w = 0

    # Puntos de evaluación a lo largo del elemento
    x = np.linspace(0, Le, 30) # Coordenada local (m)

    # Ecuación del momento flector para carga uniforme:  $M(x) = (w/2)*x^2 + V_i*x - M_i$ 
    M = (w / 2) * x**2 + Vi * x - Mi

    # Coordenadas para dibujar el diagrama parabólico (cuando hay carga)
    X_diag = xi + ex * x + escala_momento * M * nx
    Y_diag = yi + ey * x + escala_momento * M * ny
    X_base = xi + ex * x
    Y_base = yi + ey * x

    # Dibujar diagrama con relleno
    plt.plot(X_diag, Y_diag, '-', linewidth=1.5)
    plt.fill(np.concatenate([X_base, X_diag[:, -1]]),
            np.concatenate([Y_base, Y_diag[:, -1]]), alpha=0.4)

    # Calcular y marcar el punto de momento máximo (donde el cortante se anula)
    if abs(w) > 1e-9: # Evitar división por cero (solo si hay carga)
        x_max = -Vi / w # Punto donde el cortante  $V(x) = w*x + V_i = 0$ 
```



```
# Verificar que el punto está dentro del elemento
if 0 <= x_max <= Le:
    M_max = (w / 2) * x_max**2 + Vi * x_max - Mi

# Coordenadas del punto de momento máximo en el diagrama
xm = xi + ex * x_max + escala_momento * M_max * nx
ym = yi + ey * x_max + escala_momento * M_max * ny

# Marcar el punto y etiquetar
plt.scatter(xm, ym, s=30, zorder=5)
plt.text(xm, ym + 0.3, f"{M_max:.2f}", fontsize=9,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

# Etiquetas con valores de momento en extremos
# Nota: El signo negativo en Mi es por convención gráfica
plt.text(X_diag[0], Y_diag[0], f"{-Mi:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.text(X_diag[-1], Y_diag[-1], f"{Mf:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

MÉTODO MATRICIAL DE RIGIDEZ PARA PÓRTICOS - EJERCICIO 1

=== DATOS DE ENTRADA ===

Módulo de elasticidad: 24870062.324 kN/m²

Área de la sección transversal: [0.09 0.09 0.09 0.09 0.105 0.105] m²

Longitud: [4 3 3 4 5 5] m

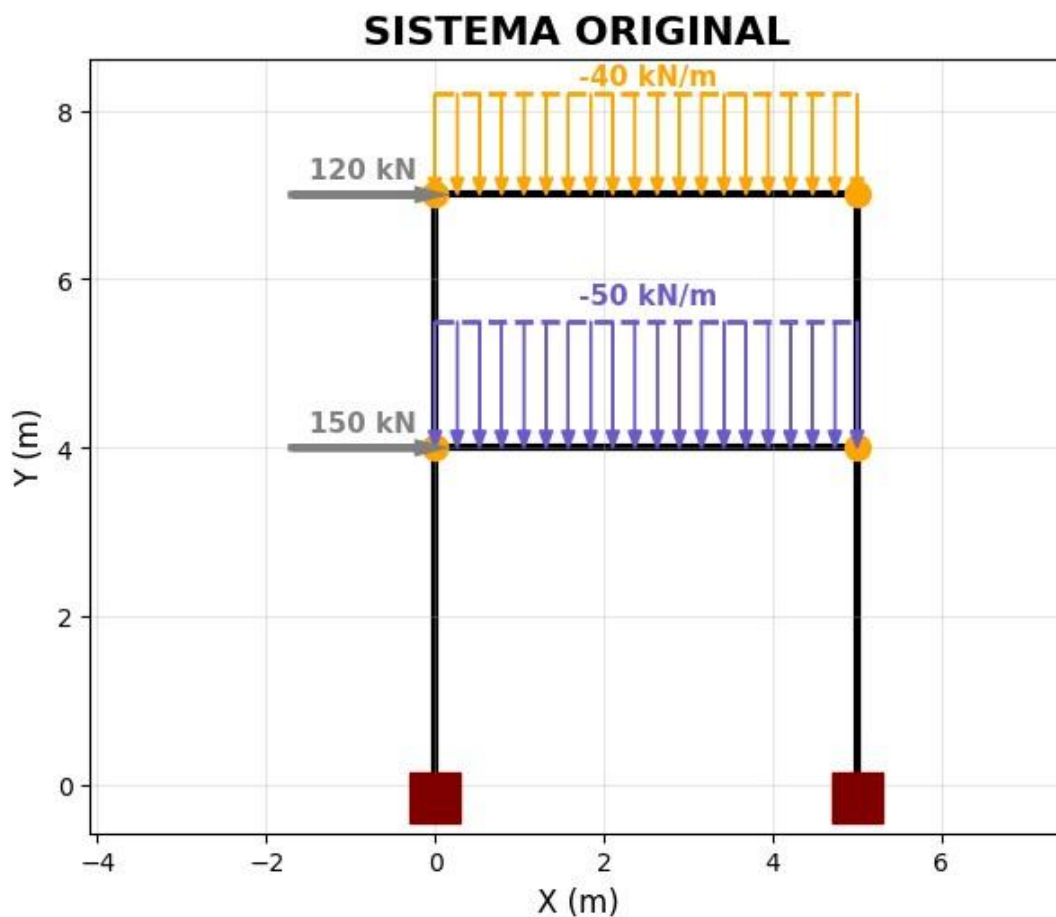
Inercia: [0.0007 0.0007 0.0007 0.0007 0.0011 0.0011] m

Ángulo de inclinación: [90 90 90 90 0 0] °

Número de elementos: 6

Número de nodos: 6

Número de grados de libertad: 18



=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA ===

Matriz global del sistema:

```
[[ 3148.      0. -6295. -3148.      0. -6295.      0.      0.      0.
  0.      0.      0.      0.      0.      0.      0.      0.      0.]
 [      0. 559576.      0.     -0. -559576.      0.      0.      0.      0.
  0.      0.      0.      0.      0.      0.      0.      0.      0.]
 [ -6295.      0. 16787.  6295.     -0.  8394.      0.      0.      0.
  0.      0.      0.      0.      0.      0.      0.      0.      0.]
 [ -3148.     -0.  6295. 532880.      0. -4896. -7461.     -0. -11192.
  0.      0.      0. -522271.      0.      0.      0.      0.      0.]
 [     -0. -559576.     -0.      0. 1308237.  6398.     -0. -746102.      0.
  0.      0.      0.      0. -2559.  6398.      0.      0.      0.]
 [ -6295.      0.  8394. -4896.  6398.  60496. 11192.     -0. 11192.
  0.      0.      0.      0. -6398. 10663.      0.      0.      0.]
 [      0.      0.      0.     -7461.     -0. 11192. 529732.      0. 11192.
 -522271.      0.      0.      0.      0.      0.      0.      0.      0.]
 [      0.      0.      0.     -0. -746102.     -0.      0. 748661.  6398.
  0. -2559.  6398.      0.      0.      0.      0.      0.      0.]
 [      0.      0.      0. -11192.      0. 11192. 11192.  6398.  43709.
  0. -6398. 10663.      0.      0.      0.      0.      0.      0.]
 [      0.      0.      0.      0.      0.      0.      0. -522271.      0.
 529732.      0. 11192. -7461.     -0. 11192.      0.      0.      0.]
 [      0.      0.      0.      0.      0.      0.      0.      0. -2559. -6398.
 0. 748661. -6398.     -0. -746102.     -0.      0.      0.      0.]
 [      0.      0.      0.      0.      0.      0.      0.      0.  6398. 10663.
 11192. -6398.  43709. -11192.      0. 11192.      0.      0.      0.]
 [      0.      0.      0. -522271.      0.      0.      0.      0.      0.      0.
 -7461.     -0. -11192. 532880.      0. -4896. -3148.     -0.  6295.]
 [      0.      0.      0.      0. -2559. -6398.      0.      0.      0.      0.
 -0. -746102.      0.      0. 1308237. -6398.     -0. -559576.     -0.]
 [      0.      0.      0.      0.  6398. 10663.      0.      0.      0.      0.
 11192.     -0. 11192. -4896. -6398.  60496. -6295.      0.  8394.]
 [      0.      0.      0.      0.      0.      0.      0.      0.      0.      0.
 0.      0.      0. -3148.     -0. -6295.  3148.      0. -6295.]
 [      0.      0.      0.      0.      0.      0.      0.      0.      0.      0.
 0.      0.      0.     -0. -559576.      0.      0. 559576.      0.]
 [      0.      0.      0.      0.      0.      0.      0.      0.      0.      0.
 0.      0.      0.  6295.     -0.  8394. -6295.      0. 16787.]]
```

=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===

```
[ 0.      0.      0.      0. 125. 104.17      0. 100.  83.33
  0. 100. -83.33      0. 125. -104.17      0.      0.      0. ]
```

=== RESTRICCIONES ===

Nodo 1: Dx=1 (Restringido), Dy=1 (Restringido), Dz=1 (Restringido)

Nodo 2: Dx=0 (Libre), Dy=0 (Libre), Dz=0 (Libre)

Nodo 3: Dx=0 (Libre), Dy=0 (Libre), Dz=0 (Libre)

Nodo 4: Dx=0 (Libre), Dy=0 (Libre), Dz=0 (Libre)

Nodo 5: Dx=0 (Libre), Dy=0 (Libre), Dz=0 (Libre)

Nodo 6: Dx=1 (Restringido), Dy=1 (Restringido), Dz=1 (Restringido)

=== VECTOR DE FUERZAS EXTERNAS ===

Fuerzas externas: [0 0 0 150 0 0 120 0 0 0 0 0 0 0 0 0
0] kN

=== REDUCCIÓN DEL SISTEMA ===

Grados de libertad activos: 12

Índices: [4 5 6 7 8 9 10 11 12 13 14 15]

Matriz global reducida:

```
[[ 532880.    0. -4896. -7461.    0. -11192.    0.    0.    0.
-522271.    0.    0.]
[    0. 1308237.  6398.    0. -746102.    0.    0.    0.    0.
 0. -2559.  6398.]
[ -4896.  6398.  60496.  11192.    0.  11192.    0.    0.    0.
 0. -6398. 10663.]
[ -7461.    0.  11192.  529732.    0.  11192. -522271.    0.    0.
 0.    0.    0.]
[    0. -746102.    0.    0.  748661.  6398.    0. -2559.  6398.
 0.    0.    0.]
[ -11192.    0.  11192.  11192.  6398.  43709.    0. -6398. 10663.
 0.    0.    0.]
[    0.    0.    0. -522271.    0.    0.  529732.    0.  11192.
-7461.    0.  11192.]
[    0.    0.    0.    0. -2559. -6398.    0.  748661. -6398.
-0. -746102.    0.]
[    0.    0.    0.    0.  6398. 10663.  11192. -6398.  43709.
-11192.    0.  11192.]
[-522271.    0.    0.    0.    0.    0. -7461.    0. -11192.
532880.    0. -4896.]
[    0. -2559. -6398.    0.    0.    0.    0. -746102.    0.
 0. 1308237. -6398.]
[    0.  6398. 10663.    0.    0.    0.  11192.    0.  11192.
-4896. -6398.  60496.]]
```

Vector de fuerzas externas reducido: [150 0 0 120 0 0 0 0 0 0 0 0] kN

Vector FEM reducido: [0. 125. 104.17 0. 100. 83.33 0. 100. -83.33
0. 125. -104.17] kN

===== SOLUCIÓN DEL SISTEMA =====

=== REACCIONES ===

Nodo 1: $R_x = -124.66$ kN, $R_y = 60.49$ kN, $M_z = 295.03$ kN-m

Nodo 6: $R_x = -145.34$ kN, $R_y = 389.51$ kN, $M_z = 322.42$ kN-m

=== DESPLAZAMIENTOS ===

Nodo 1: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $T_z = 0.000e+00$ rad

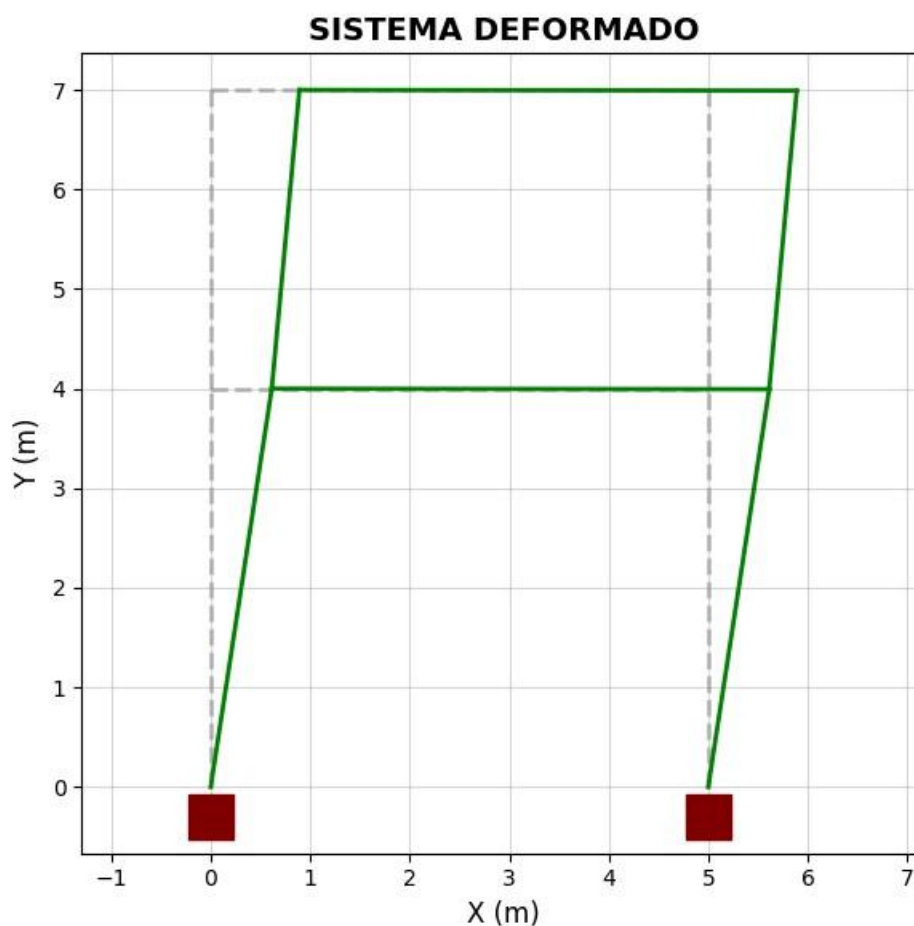
Nodo 2: $U_x = 6.139e-02$ m, $U_y = -1.081e-04$ m, $T_z = -1.089e-02$ rad

Nodo 3: $U_x = 8.916e-02$ m, $U_y = -1.779e-04$ m, $T_z = -5.867e-03$ rad

Nodo 4: $U_x = 8.897e-02$ m, $U_y = -8.944e-04$ m, $T_z = -1.914e-03$ rad

Nodo 5: $U_x = 6.130e-02$ m, $U_y = -6.961e-04$ m, $T_z = -7.563e-03$ rad

Nodo 6: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $T_z = 0.000e+00$ rad



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 60.490$ kN, $V = 124.661$ kN, $M = 295.031$ kN-m

Nodo 2: $N = -60.490$ kN, $V = -124.661$ kN, $M = 203.615$ kN-m

Elemento 2:

Nodo 2: $N = 52.054 \text{ kN}$, $V = 19.658 \text{ kN}$, $M = 1.371 \text{ kN-m}$

Nodo 3: $N = -52.054 \text{ kN}$, $V = -19.658 \text{ kN}$, $M = 57.603 \text{ kN-m}$

Elemento 3:

Nodo 5: $N = 147.946 \text{ kN}$, $V = 100.342 \text{ kN}$, $M = 118.901 \text{ kN-m}$

Nodo 4: $N = -147.946 \text{ kN}$, $V = -100.342 \text{ kN}$, $M = 182.125 \text{ kN-m}$

Elemento 4:

Nodo 6: $N = 389.510 \text{ kN}$, $V = 145.339 \text{ kN}$, $M = 322.420 \text{ kN-m}$

Nodo 5: $N = -389.510 \text{ kN}$, $V = -145.339 \text{ kN}$, $M = 258.935 \text{ kN-m}$

Elemento 5:

Nodo 3: $N = 100.342 \text{ kN}$, $V = 52.054 \text{ kN}$, $M = -57.603 \text{ kN-m}$

Nodo 4: $N = -100.342 \text{ kN}$, $V = 147.946 \text{ kN}$, $M = -182.125 \text{ kN-m}$

Elemento 6:

Nodo 2: $N = 44.996 \text{ kN}$, $V = 8.436 \text{ kN}$, $M = -204.985 \text{ kN-m}$

Nodo 5: $N = -44.996 \text{ kN}$, $V = 241.564 \text{ kN}$, $M = -377.836 \text{ kN-m}$

DIAGRAMA AXIAL

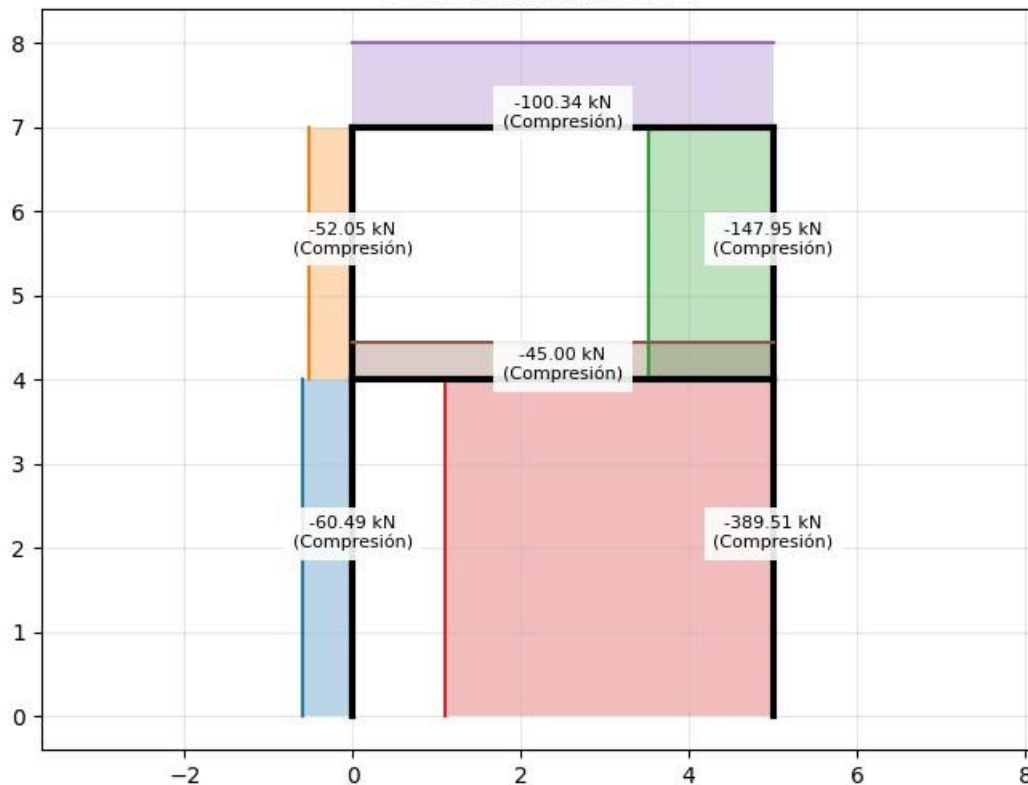


DIAGRAMA CORTANTE

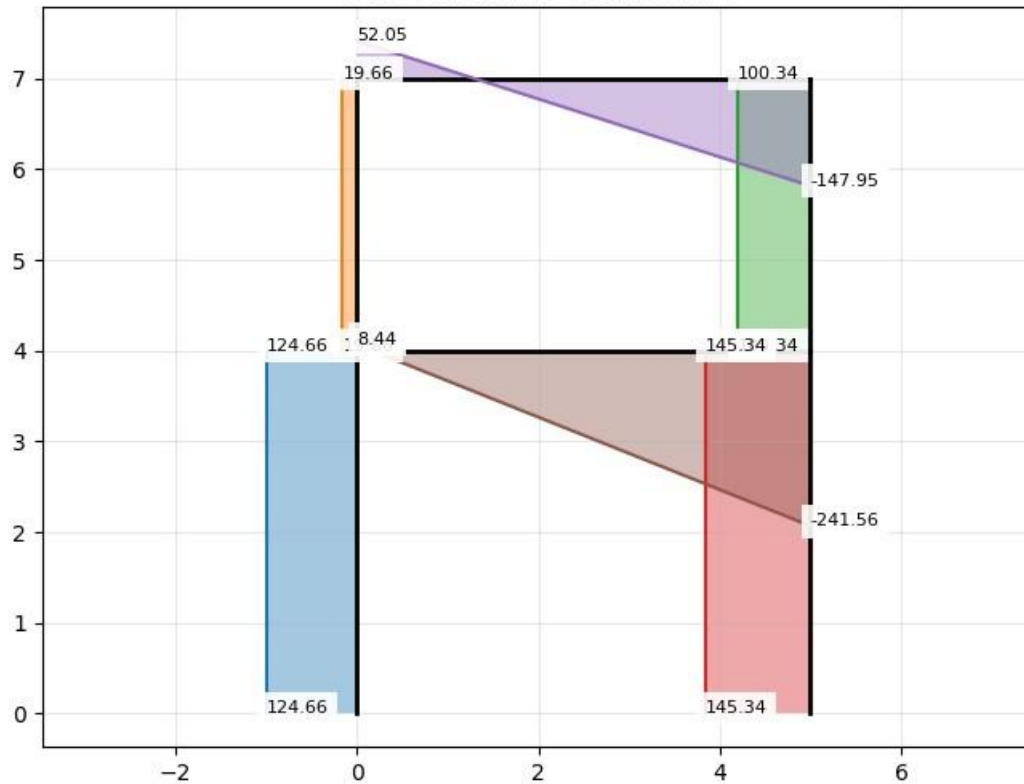
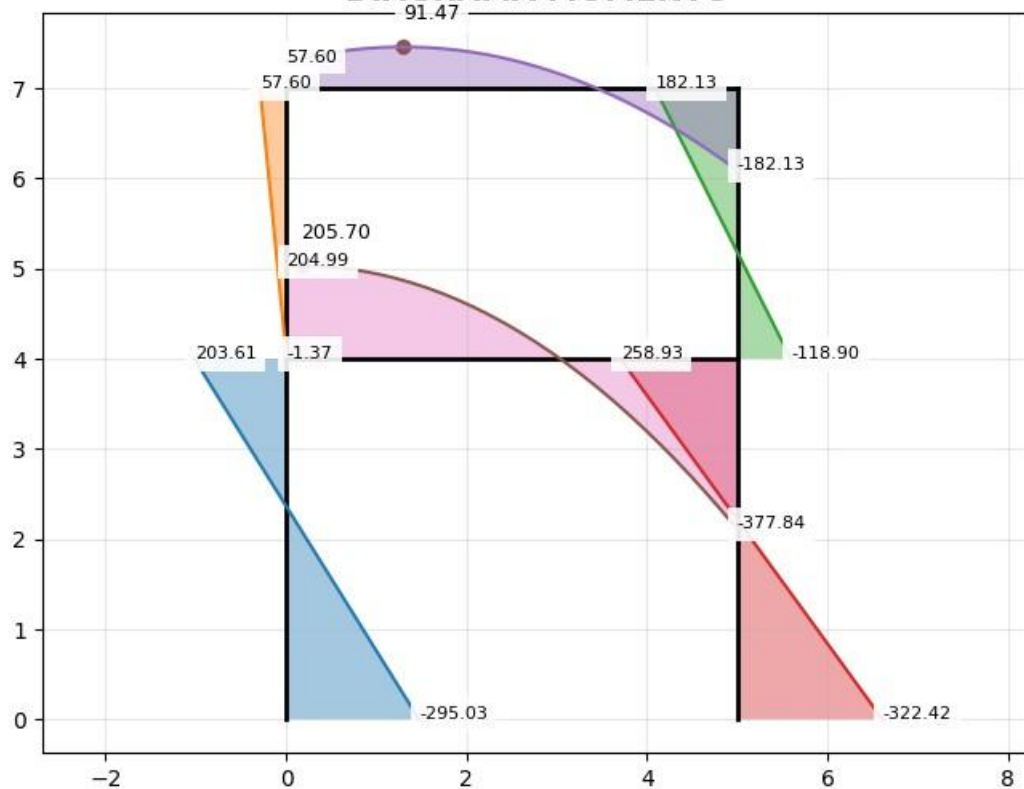


DIAGRAMA MOMENTO



MODELO OPENSEES

PÓRTICOS: EJERCICIO 1

```
# =====
# IMPORTACIÓN DE LIBRERÍAS
# =====
import openseespy.opensees as ops # Biblioteca principal de OpenSees para análisis estructural
import opsviz as opsv            # Librería para visualización de resultados y diagramas
import numpy as np               # Para operaciones numéricas y arreglos
import math                      # Para funciones matemáticas básicas (atan, sqrt)
import matplotlib.pyplot as plt  # Para visualización de resultados personalizada

print("MODELO EN OPENSEES DEL EJERCICIO 1 - PÓRTICOS")

# =====
# INICIALIZACIÓN DEL MODELO
# =====
ops.wipe() # Limpia cualquier modelo previo en memoria para empezar desde cero.

# Crear modelo básico 2D con 3 grados de libertad por nodo (dx, dy, rotación).
# '-ndm' es el número de dimensiones espaciales (2D) y '-ndf' los grados de libertad
# por nodo.
ops.model('basic', '-ndm', 2, '-ndf', 3)

# =====
# DEFINICIÓN DE PROPIEDADES DEL MATERIAL Y GEOMETRÍA
# =====
# Propiedades de la sección transversal (unidades: kN, m)
E = 24870062.324 # Módulo de elasticidad del material (kN/m²)

# --- Sección de vigas ---
Av = 0.3 * 0.35 # Área de la sección transversal (m²)
Iv = (1/12) * 0.3 * (0.35**3) # Momento de inercia alrededor del eje fuerte (m⁴)

# --- Sección de columnas ---
Ac = 0.3 * 0.3 # Área de la sección transversal (m²)
Ic = (1/12) * 0.3 * (0.3**3) # Momento de inercia alrededor del eje fuerte (m⁴)

# =====
# DEFINICIÓN DE NODOS
# =====
# Coordenadas en metros: [x, y]. Define la geometría del pórtico.
ops.node(1, 0, 0) # Nodo 1
ops.node(2, 0, 4) # Nodo 2
ops.node(3, 0, 7) # Nodo 3
ops.node(4, 5, 7) # Nodo 4
ops.node(5, 5, 4) # Nodo 5
ops.node(6, 5, 0) # Nodo 6
```

```
# =====
# CONDICIONES DE APOYO
# =====
# Empotramiento completo: restringir dx, dy, dz (1 = restringido, 0 = libre).
ops.fix(1, 1, 1, 1) # Nodo 1: Empotrado
ops.fix(6, 1, 1, 1) # Nodo 6: Empotrado

# =====
# DEFINICIÓN DE SECCIONES ELÁSTICAS
# =====
# Se define una sección transversal con comportamiento elástico-lineal.
# Se crea un 'tag' o identificador único para cada tipo de sección.
ops.section('Elastic', 1, E, Av, Iv) # Sección para vigas (tag=1)
ops.section('Elastic', 2, E, Ac, Ic) # Sección para columnas (tag=2)

# =====
# TRANSFORMACIÓN GEOMÉTRICA
# =====
# Transformación lineal, adecuada para análisis donde se asumen pequeñas
# deformaciones y desplazamientos. El 'tag' 1 identifica esta transformación.
ops.geomTransf('Linear', 1)

# =====
# ESQUEMA DE INTEGRACIÓN
# =====
# Define cómo se integran las propiedades a lo largo del elemento para obtener su
# matriz de rigidez. Integración de Lobatto con 10 puntos de integración para alta
# precisión. La integración 'Lobatto' asocia una sección (por su tag) a un elemento.
ops.beamIntegration('Lobatto', 1, 1, 10) # Para vigas (tag 1), usando la sección con
tag 1.
ops.beamIntegration('Lobatto', 2, 2, 10) # Para columnas (tag 2), usando la sección
con tag 2.

# =====
# CREACIÓN DE ELEMENTOS
# =====
# Lista para almacenar información de elementos (para post-procesamiento y
# visualización).
Elementos = []

# --- COLUMNAS (sección de columnas, integración 2) ---
# El comando 'dispBeamColumn' crea un elemento viga-columna basado en formulación de
# desplazamientos. ops.element(tag elemento, nodo inicial, nodo final, tag transf.
# geométrica, tag integración)

ops.element('dispBeamColumn', 1, 1, 2, 1, 2)
Elementos.append({"ID": 1, "Nodo_i": 1, "Nodo_j": 2}) # Guardamos info para graficar
```

```
ops.element('dispBeamColumn', 2, 2, 3, 1, 2)
Elementos.append({"ID": 2, "Nodo_i": 2, "Nodo_j": 3})

ops.element('dispBeamColumn', 3, 5, 4, 1, 2)
Elementos.append({"ID": 3, "Nodo_i": 5, "Nodo_j": 4})

ops.element('dispBeamColumn', 4, 6, 5, 1, 2)
Elementos.append({"ID": 4, "Nodo_i": 6, "Nodo_j": 5})

# --- VIGAS (sección de vigas, integración 1) ---
ops.element('dispBeamColumn', 5, 3, 4, 1, 1)
Elementos.append({"ID": 5, "Nodo_i": 3, "Nodo_j": 4})

ops.element('dispBeamColumn', 6, 2, 5, 1, 1)
Elementos.append({"ID": 6, "Nodo_i": 2, "Nodo_j": 5})

# =====
# APLICACIÓN DE CARGAS
# =====
# Configurar patrón de carga estática. 'timeSeries' define cómo varía la carga con el
# tiempo (o con el paso de análisis). 'Linear' significa que crece linealmente.
ops.timeSeries('Linear', 1) # Serie temporal lineal (tag 1) que va de 0 a 1 en un
# paso.
# 'pattern' agrupa un conjunto de cargas que se aplican con una misma serie temporal.
ops.pattern('Plain', 1, 1) # Patrón de carga simple (tag 1) asociado a la serie
# temporal tag 1.

# --- Cargas distribuidas en vigas ---
w5 = -40 # Viga superior (elemento 5): -40 kN/m
w6 = -50 # Viga inferior (elemento 6): -50 kN/m

# Aplicar cargas distribuidas. 'beamUniform' es una carga uniforme en el elemento.
ops.eleLoad('-ele', 5, '-type', 'beamUniform', w5, 0)
ops.eleLoad('-ele', 6, '-type', 'beamUniform', w6, 0)

# --- Cargas Puntuales en nodos ---
# ops.load aplica una carga directamente en un nodo. Los argumentos son: nodo, Fx,
# Fy, Mz.
ops.load(2, 150, 0, 0) # Nodo 2: 150 kN en dirección X
ops.load(3, 120, 0, 0) # Nodo 3: 120 kN en dirección X

# =====
# GRÁFICA SISTEMA ORIGINAL
# =====
# Definir información de cargas para visualización.
Cargas = [
    [], [], [], [], # Elementos 1-4 (columnas): sin carga distribuida aplicada
    ['Uniforme', 'Local_y', 'orangered', -40, -40], # Elemento 5: tipo, dir, color, wi, wf
    ['Uniforme', 'Local_y', 'sienna', -50, -50] # Elemento 6
```

```
# Configurar figura para visualización
plt.figure(figsize=(10, 8))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos estructurales
for element_data in Elementos:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Dibujar la barra (línea negra sólida)
    plt.plot([xi, xf], [yi, yf], 'k-', lw=2, zorder=1)

# Dibujar nodos principales
for i in range(1, 7):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'o', color='purple', markersize=8, zorder=2)

# Dibujar símbolos de apoyos empotrados (cuadrados)
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=3)
plt.plot(5, -0.15, 's', color='maroon', markersize=20, zorder=3)

# --- Dibujo cargas distribuidas ---
escala = 0.03 # Factor de escala para visualización de cargas

for i, element_data in enumerate(Elementos):
    # Saltar elementos sin carga
    if not Cargas[i]:
        continue

    # Extraer información de carga para este elemento
    tipo, direccion, color, w1, w2, *extra = Cargas[i]

    # Coordenadas de los nodos del elemento
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Vector del elemento y su longitud
    dx = xf - xi
    dy = yf - yi
    L_elem = (dx ** 2 + dy ** 2) ** 0.5
```

```
# Vectores unitarios
ex = dx / L_elem # Vector unitario en dirección del elemento
ey = dy / L_elem

# Vector normal (perpendicular) para cargas en dirección local y
nx = -ey
ny = ex

# Puntos a lo largo del elemento para dibujar flechas
t_vals = np.linspace(0, 1, 20) # 20 puntos uniformemente espaciados

# Listas para dibujar línea superior de carga
x_superior = []
y_superior = []

for t in t_vals:
    # Punto a lo largo del elemento
    x = xi + t * dx
    y = yi + t * dy

    # Magnitud de carga en este punto (interpolación lineal)
    w = -(w1 + (w2 - w1) * t) # Negativo para dirección correcta
    # Posición de la base de la flecha (carga en dirección local perpendicular)
    x_base = x + escala * w * nx
    y_base = y + escala * w * ny

    # Dibujar flecha individual
    plt.arrow(x_base, y_base, x - x_base, y - y_base, head_width=0.15,
head_length=0.2, fc=color, ec=color, length_includes_head=True, zorder=4)

    # Guardar puntos para línea superior
    x_superior.append(x_base)
    y_superior.append(y_base)

# Dibujar línea que une las puntas de las flechas
plt.plot(x_superior, y_superior, '--', color=color, linewidth=2, zorder=4)

# --- Etiquetas de cargas y anotaciones ---
# Etiquetas para cargas distribuidas
plt.text(2, 8.3, '-40 kN/m', color='orangered', fontsize=11, fontweight='bold',
zorder=5)
plt.text(2, 5.7, '-50 kN/m', color='sienna', fontsize=11, fontweight='bold',
zorder=5)

# Cargas concentradas horizontales
plt.arrow(-1.7, 4, 1.5, 0, head_width=0.1, head_length=0.2, fc='gold', ec='gold',
linewidth=3, zorder=5)
plt.text(-1.3, 4.2, '150 kN', color='gold', fontweight='bold', fontsize=11, zorder=5)
```

```
plt.arrow(-1.7, 7, 1.5, 0, head_width=0.1, head_length=0.2, fc='gold', ec='gold',
linewidth=3, zorder=5)
plt.text(-1.3, 7.2, '120 kN', color='gold', fontweight='bold', fontsize=11, zorder=5)

# Configuración final del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal') # Mantener proporciones 1:1
plt.tight_layout()
plt.show()

# =====
# CONFIGURACIÓN DEL ANÁLISIS
# =====
# Se configuran Los objetos que controlan cómo se ensambla y resuelve el sistema de
ecuaciones.
ops.system('BandSPD') # Almacena la matriz de rigidez en formato de banda simétrica
definida positiva.
ops.numberer('Plain') # Numeración simple de grados de libertad (el orden en que se
ingresan los nodos).
ops.constraints('Plain') # Método simple para imponer restricciones (como los apoyos
fijos).
ops.integrator('LoadControl', 1.0) # Controla el incremento de carga. '1.0' aplica
el 100% de la carga en un solo paso.

ops.algorithm('Linear') # Algoritmo de solución para sistemas de ecuaciones
lineales. Apropiado para análisis elástico-lineal.
ops.analysis('Static') # Especifica que se realizará un análisis estático.
ops.analyze(1) # Ejecuta el análisis con el número de pasos especificado (1 paso).

# =====
# RESULTADOS
# =====
print("\n===== RESULTADOS =====")
ops.reactions() # Calcular reacciones en apoyos. Esta función calcula las fuerzas
de reacción en los nodos restringidos.

print("\n=== REACCIONES EN APOYOS ===")
for nodo in [1, 6]:
    Rx = ops.nodeReaction(nodo, 1) # Reacción en dirección X
    Ry = ops.nodeReaction(nodo, 2) # Reacción en dirección Y
    Rz = ops.nodeReaction(nodo, 3) # Reacción momento

    print(f"Node {nodo}: Rx={Rx:.2f} kN, Ry={Ry:.2f} kN, Mz={Rz:.2f} kN-m")
```

```
print("\n=== DESPLAZAMIENTOS NODALES ===")
for i in range(1, 7):
    ux = ops.nodeDisp(i, 1) # Desplazamiento en X (gdl 1)
    uy = ops.nodeDisp(i, 2) # Desplazamiento en Y (gdl 2)
    uz = ops.nodeDisp(i, 3) # Rotación (θz, gdl 3)

    print(f"Node {i}: Ux={ux:.3e} m, Uy={uy:.3e} m, θz={uz:.3e} rad")

# =====
# GRÁFICA SISTEMA DEFORMADO
# =====
# Crear figura para la deformada con la librería opsv, que simplifica este proceso.
fig, ax = plt.subplots(figsize=(8, 6))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar la deformada. La función plot_defo usa los resultados del análisis.
opsv.plot_defo(ax=ax)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

print("\n=== FUERZAS INTERNAS EN ELEMENTOS ===")
for element_data in Elementos:
    eleTag = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nj = element_data["Nodo_j"]

    # Obtener fuerzas internas en el sistema de coordenadas locales del elemento.
    # El orden típico es: [N_i, V_i, M_i, N_j, V_j, M_j]
    F_int = ops.eleResponse(eleTag, 'localForces')

    print(f"Elemento {eleTag}:")
    print(f"Nodo {Ni}: N = {F_int[0]:.3f} kN, V = {F_int[1]:.3f} kN, M = {F_int[2]:.3f} kN-m")
    print(f"Nodo {Nj}: N = {F_int[3]:.3f} kN, V = {F_int[4]:.3f} kN, M = {F_int[5]:.3f} kN-m\n")
```



```
# =====
# DIAGRAMAS DE ESFUERZOS INTERNOS
# =====
# opsvis también proporciona funciones para dibujar diagramas de esfuerzos.
# El factor de escala (sfac) controla el tamaño de los diagramas para una mejor
# visualización.
# nep es el número de puntos en los que se evalúa el esfuerzo a lo largo del
# elemento.

# --- Diagrama de Fuerza Axial ---
fig_n = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA AXIAL (kN)', fontsize=14, fontweight='bold')
ax_n = plt.gca()
opsv.section_force_diagram_2d(sf_type='N', sfac=0.008, nep=20, ax=ax_n)
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

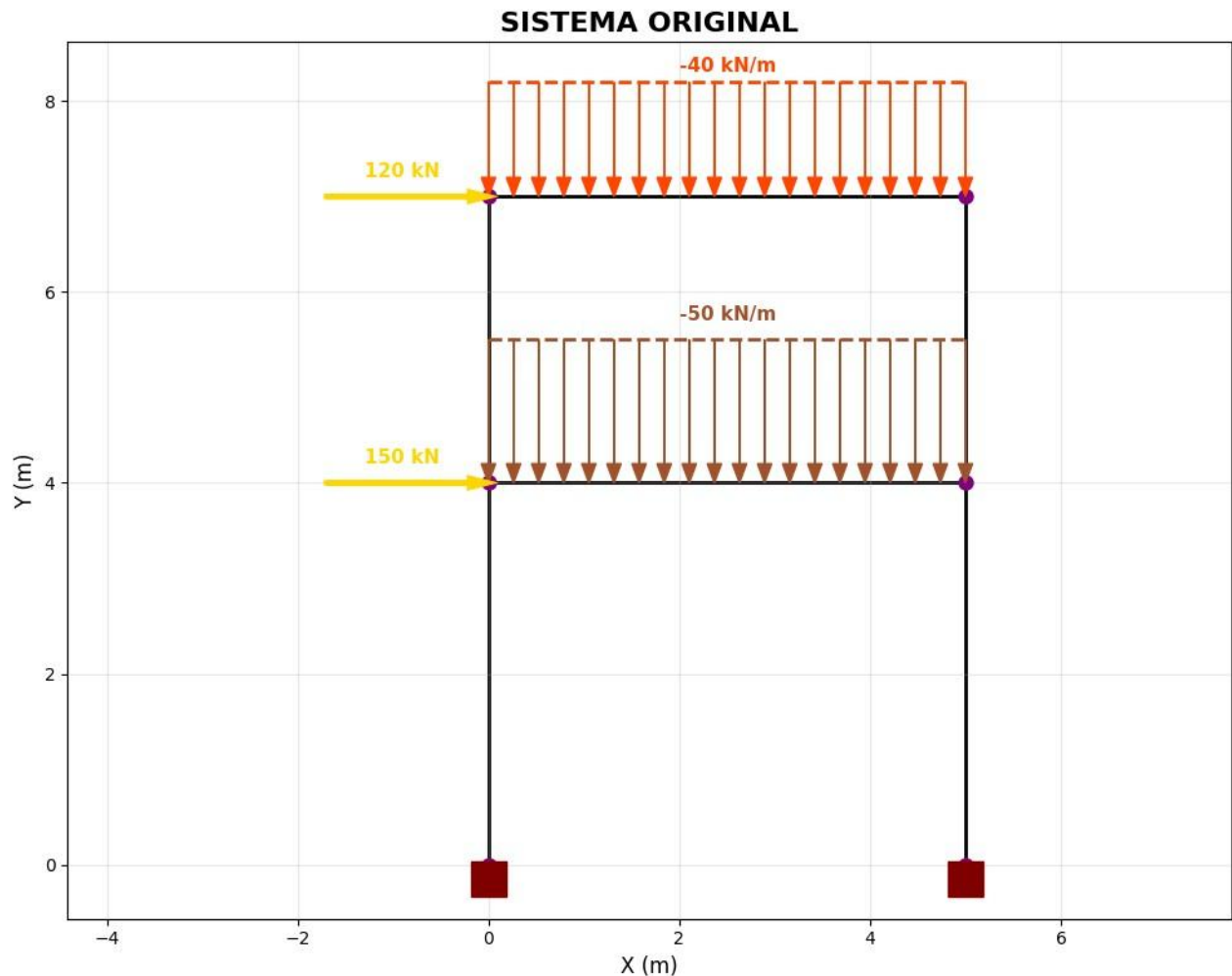
# --- Diagrama de Fuerza Cortante ---
fig_v = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA CORTANTE (kN)', fontsize=14, fontweight='bold')
ax_v = plt.gca()
opsv.section_force_diagram_2d(sf_type='V', sfac=0.01, nep=20, ax=ax_v)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

# --- Diagrama de Momento Flector ---
fig_m = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE MOMENTO FLECTOR (kN-m)', fontsize=14, fontweight='bold')
ax_m = plt.gca()
opsv.section_force_diagram_2d(sf_type='M', sfac=0.005, nep=20, ax=ax_m)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

MODELO EN OPENSEES DEL EJERCICIO 1 - PÓRTICOS



===== RESULTADOS =====

=== REACCIONES EN APOYOS ===

Nodo 1: $R_x = -124.66$ kN, $R_y = 60.49$ kN, $M_z = 295.03$ kN-m

Nodo 6: $R_x = -145.34$ kN, $R_y = 389.51$ kN, $M_z = 322.42$ kN-m

=== DESPLAZAMIENTOS NODALES ===

Nodo 1: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $\theta_z = 0.000e+00$ rad

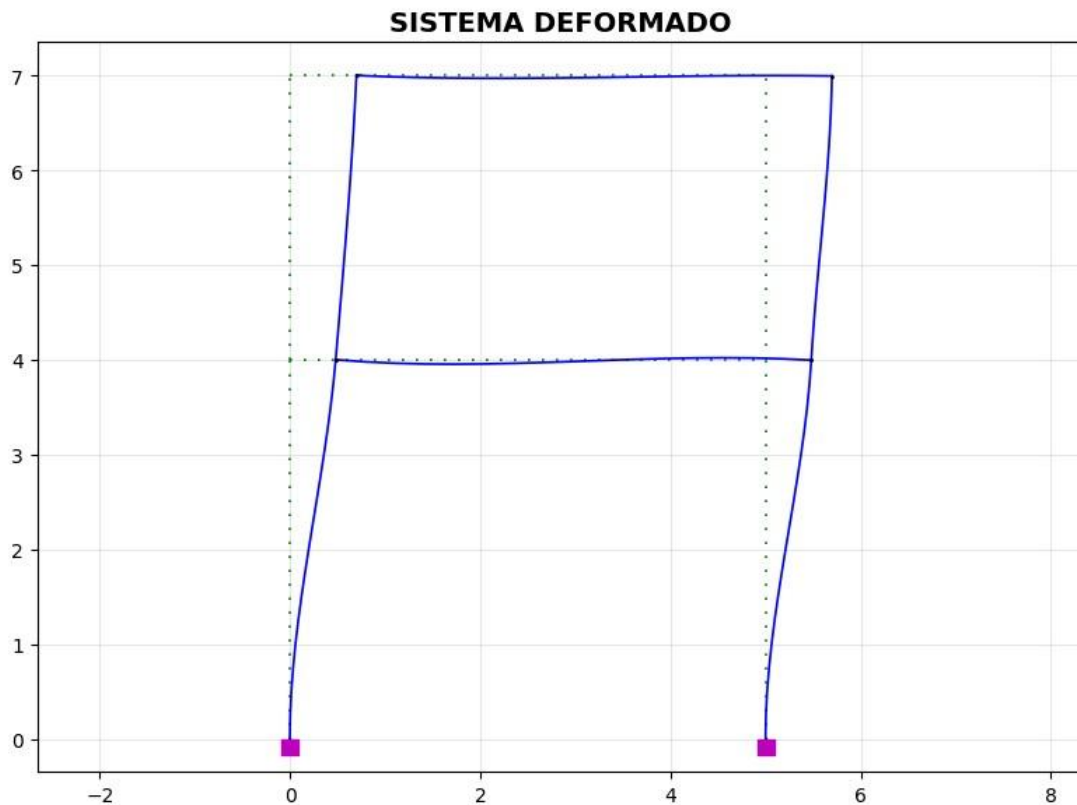
Nodo 2: $U_x = 6.139e-02$ m, $U_y = -1.081e-04$ m, $\theta_z = -1.089e-02$ rad

Nodo 3: $U_x = 8.916e-02$ m, $U_y = -1.779e-04$ m, $\theta_z = -5.867e-03$ rad

Nodo 4: $U_x = 8.897e-02$ m, $U_y = -8.944e-04$ m, $\theta_z = -1.914e-03$ rad

Nodo 5: $U_x = 6.130e-02$ m, $U_y = -6.961e-04$ m, $\theta_z = -7.563e-03$ rad

Nodo 6: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $\theta_z = 0.000e+00$ rad



=== FUERZAS INTERNAS EN ELEMENTOS ===

Elemento 1:

Nodo 1: $N = 60.490 \text{ kN}$, $V = 124.661 \text{ kN}$, $M = 295.031 \text{ kN-m}$

Nodo 2: $N = -60.490 \text{ kN}$, $V = -124.661 \text{ kN}$, $M = 203.615 \text{ kN-m}$

Elemento 2:

Nodo 2: $N = 52.054 \text{ kN}$, $V = 19.658 \text{ kN}$, $M = 1.371 \text{ kN-m}$

Nodo 3: $N = -52.054 \text{ kN}$, $V = -19.658 \text{ kN}$, $M = 57.603 \text{ kN-m}$

Elemento 3:

Nodo 5: $N = 147.946 \text{ kN}$, $V = 100.342 \text{ kN}$, $M = 118.901 \text{ kN-m}$

Nodo 4: $N = -147.946 \text{ kN}$, $V = -100.342 \text{ kN}$, $M = 182.125 \text{ kN-m}$

Elemento 4:

Nodo 6: $N = 389.510 \text{ kN}$, $V = 145.339 \text{ kN}$, $M = 322.420 \text{ kN-m}$

Nodo 5: $N = -389.510 \text{ kN}$, $V = -145.339 \text{ kN}$, $M = 258.935 \text{ kN-m}$

Elemento 5:

Nodo 3: $N = 100.342 \text{ kN}$, $V = 52.054 \text{ kN}$, $M = -57.603 \text{ kN-m}$

Nodo 4: $N = -100.342 \text{ kN}$, $V = 147.946 \text{ kN}$, $M = -182.125 \text{ kN-m}$

Elemento 6:

Nodo 2: $N = 44.996 \text{ kN}$, $V = 8.436 \text{ kN}$, $M = -204.985 \text{ kN-m}$

Nodo 5: $N = -44.996 \text{ kN}$, $V = 241.564 \text{ kN}$, $M = -377.836 \text{ kN-m}$

DIAGRAMA DE FUERZA AXIAL (kN)

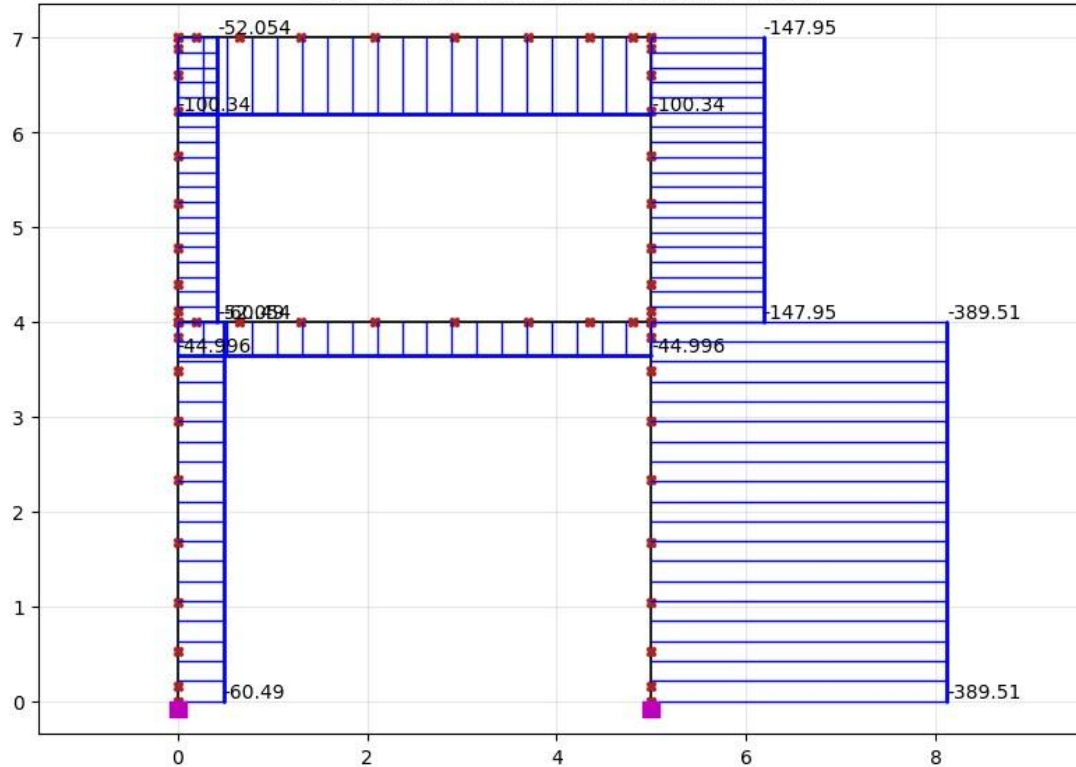
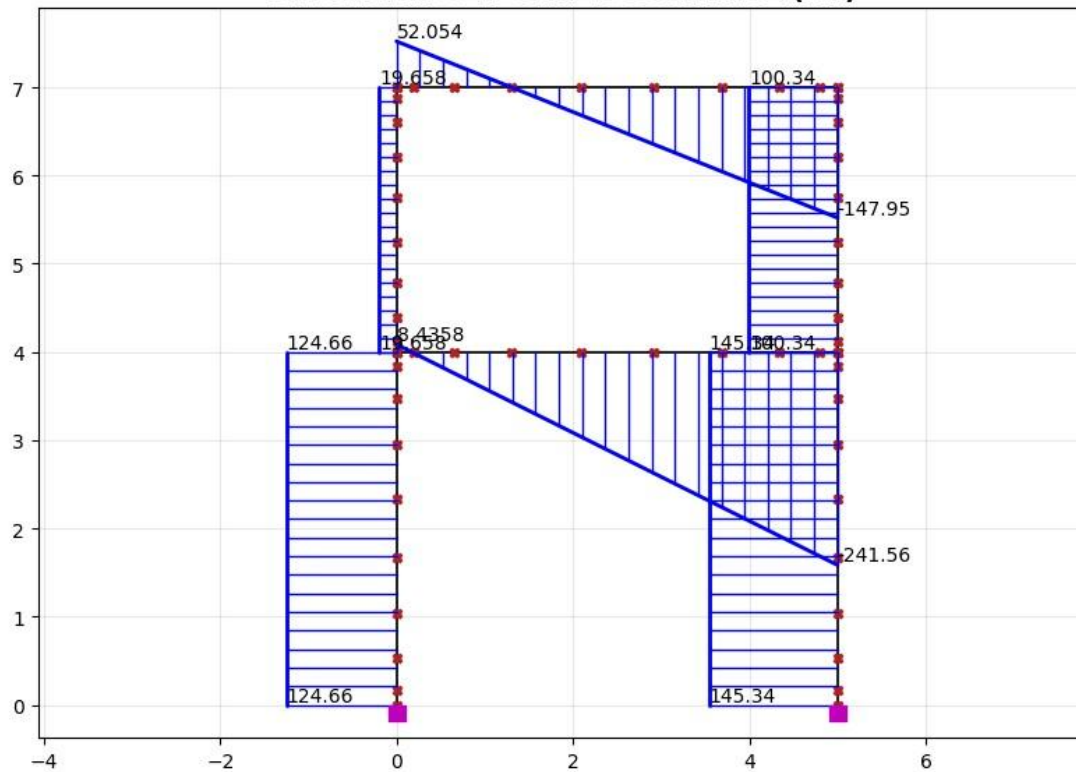
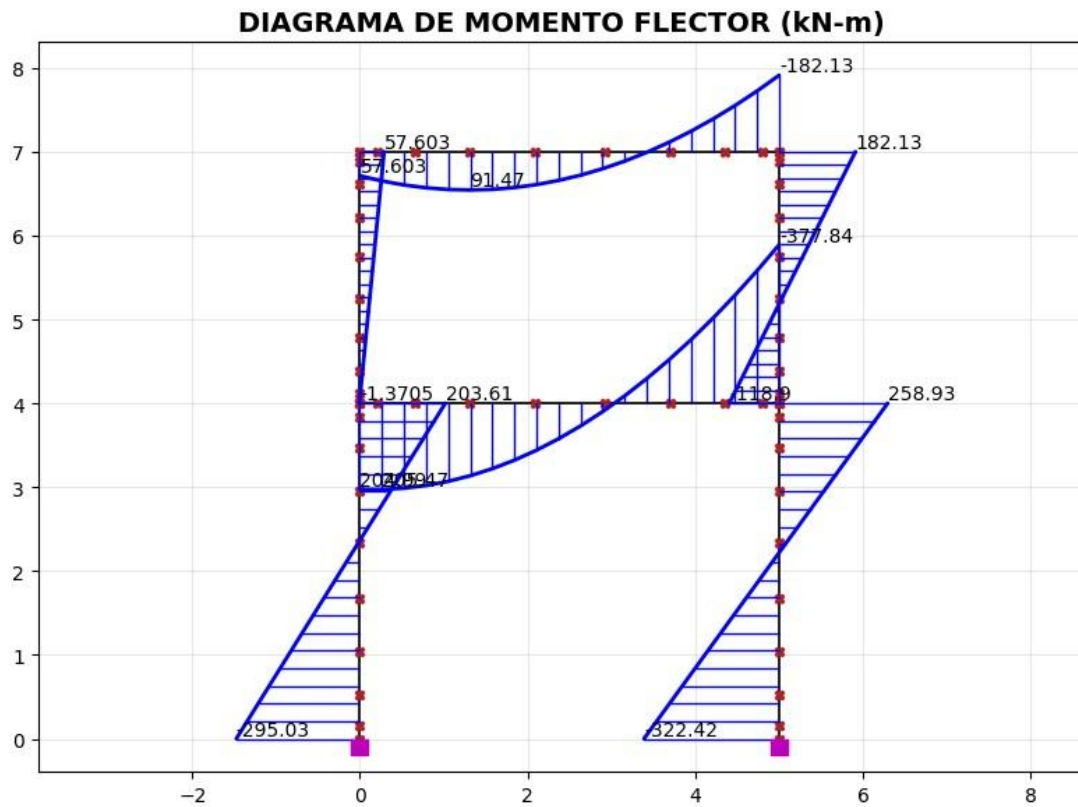


DIAGRAMA DE FUERZA CORTANTE (kN)





MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON

PÓRTICOS: EJERCICIO 2

```
import numpy as np
import math
import matplotlib.pyplot as plt

print("\033[1mMÉTODO MATRICIAL DE RIGIDEZ PARA PÓRTICOS - EJERCICIO 2\033[0m")

# =====
# DATOS DE ENTRADA
# =====
print("\n=== DATOS DE ENTRADA ===")
E = 2302520.3582 # Módulo de elasticidad del material (Ton/m²)
A = np.array([0.105, 0.105, 0.105]) # Áreas transversales (m²)
L3 = ((6**2)+(4**2))**(1/2) # Elemento 3: hipotenusa del triángulo 6x4
L = np.array([5, 4, L3]) # Longitudes de los elementos (m)
I = np.array([0.001071875, 0.001071875, 0.001071875]) # Momentos de inercia (m⁴)
a = np.array([math.degrees(math.atan(4/3)), 0, -math.atan(6/4)*(180/math.pi)]) #
Ángulos de inclinación (°)

m = 3 # Número de elementos
n = 4 # Número de nodos
GL = n*3 # Grados de Libertad totales (3 por nodo)

print(f"\nMódulo de elasticidad: {E} kN/m²")
print(f"Área de la sección transversal: {A} m²")
print(f"Longitud: {np.round(L, 2)} m")
print(f"Inercia: {np.round(I, 4)} m⁴")
print(f"Ángulo de inclinación: {np.round(a, 2)} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")

# =====
# COORDENADAS DE LOS NODOS
# =====
# Matriz de coordenadas: [x, y] para cada nodo (metros)
coordenadas_nodos = np.array([
    [0, 2], # Nodo 1
    [3, 6], # Nodo 2
    [7, 6], # Nodo 3
    [11, 0]]) # Nodo 4
```

```
# =====
# CONECTIVIDAD DE ELEMENTOS
# =====
# Cada fila define un elemento: [nodo_inicial, nodo_final]
Elementos = np.array([
    [1, 2], # Elemento 1
    [2, 3], # Elemento 2
    [3, 4]]) # Elemento 3

# =====
# NUMERACIÓN DE GRADOS DE LIBERTAD (GL)
# =====
# Se asigna un GL único a cada posible desplazamiento/rotación
# Grados de libertad en nodos iniciales
Nx = np.array([1, 4, 7]) # Desplazamientos en X
Ny = np.array([2, 5, 8]) # Desplazamientos en Y
Nz = np.array([3, 6, 9]) # Rotaciones en Z

# Grados de libertad en nodos finales
Fx = np.array([4, 7, 10]) # Desplazamientos en X
Fy = np.array([5, 8, 11]) # Desplazamientos en Y
Fz = np.array([6, 9, 12]) # Rotaciones en Z

# =====
# DEFINICIÓN DE CARGAS APLICADAS
# =====
# Formato para cada elemento: [tipo, dirección, color_visualización, w_inicial,
# w_final, *parámetros_extra]
Cargas = [
    ['Uniforme', 'Local_y', 'tomato', -40, -40], # E1: uniforme vertical -40 Ton/m
    ['Uniforme', 'Local_y', 'royalblue', -50, -30], # E2: trapezoidal -50 a -30 Ton/m
    ['Uniforme', 'Global_x', 'forestgreen', -30, -30, 6]] # E3: uniforme en X global
    -30 Ton/m, proyección=6m

# =====
# PREPROCESAMIENTO GEOMÉTRICO DE TODOS LOS ELEMENTOS
# =====
# Calcula y almacena las propiedades geométricas fundamentales para cada elemento.
geom = []
for e in range(m):
    ni, nf = Elementos[e][0], Elementos[e][1]
    xi, yi = coordenadas_nodos[ni-1]
    xf, yf = coordenadas_nodos[nf-1]

    dx = xf - xi
    dy = yf - yi
    Le = math.sqrt((dx**2)+(dy**2))
```

```
# Vector director unitario (ex, ey): apunta a lo largo del elemento.
ex = dx/Le
ey = dy/Le

# Vector normal unitario (nx, ny): perpendicular al elemento.
# Se obtiene rotando el vector director 90° en sentido antihorario.
nx = -ey
ny = ex

geom.append([ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny])

# =====
# GRÁFICA SISTEMA ORIGINAL
# =====

plt.figure(figsize=(10, 8))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=3, zorder=1)

# Dibujar nodos
for i, (x, y) in enumerate(coordenadas_nodos):
    plt.plot(x, y, 'o', color='orange', markersize=10, zorder=2)

# Dibujar apoyos fijos
plt.plot(0, 2-0.15, '^', color='maroon', markersize=20, zorder=2)
plt.plot(11, 0-0.15, '^', color='maroon', markersize=20, zorder=2)

# --- Cargas distribuidas ---
escala = 0.03 # Factor de escala para visualización

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Extraer información de carga para este elemento
    tipo, direccion, color, wi, wf, *extra = Cargas[i]

    # Puntos a lo largo del elemento para dibujar flechas
    t_vals = np.linspace(0, 1, 20) # 20 puntos uniformemente espaciados

    # Listas para almacenar puntos de la línea superior de carga
    x_superior = []
    y_superior = []
```



```

for t in t_vals:
    # Punto a lo largo del elemento (paramétrico 0-1)
    x = xi + t*dx
    y = yi + t*dy

    # Magnitud de carga en este punto (interpolación lineal)
    w = -(wi + (wf-wi)*t)

    # Calcular posición de la base de la flecha según dirección
    if direccion == "Local_y":
        # Carga en dirección local perpendicular (Y local)
        x_base = x + escala * w * nx
        y_base = y + escala * w * ny

    elif direccion == "Global_x":
        # Carga en dirección global X
        x_base = x + escala * w
        y_base = y

    # Dibujar flecha individual
    plt.arrow(x_base, y_base, x - x_base, y - y_base, head_width=0.15,
             head_length=0.2, fc=color, ec=color, length_includes_head=True, zorder=3)

    # Guardar punto para dibujar línea superior continua
    x_superior.append(x_base)
    y_superior.append(y_base)

    # Dibujar línea que une las puntas de todas las flechas
    plt.plot(x_superior, y_superior, '--', color=color, linewidth=2, zorder=3)

# --- Anotaciones del gráfico ---
# Etiquetas para cargas distribuidas
plt.text(2.1, 4, '-40 Ton/m', color='tomato', fontsize=11, fontweight='bold',
        zorder=4)
plt.text(4, 5.5, '-50 a -30 Ton/m', color='royalblue', fontsize=11,
        fontweight='bold', zorder=4)
plt.text(7, 3, f'{Cargas[2][3]*(Cargas[2][5]/L[2]):.2f} Ton/m', color='forestgreen',
        fontsize=11, fontweight='bold', zorder=4)

# Cargas puntuales (flechas moradas)
plt.arrow(3, 8.2, 0, -1.8, head_width=0.1, head_length=0.2, fc='purple', ec='purple',
        linewidth=3, zorder=4)
plt.text(1.9, 7, '90 Ton', color='purple', fontweight='bold', fontsize=11, zorder=4)

```

```
plt.arrow(9.2, 6, -1.8, 0, head_width=0.1, head_length=0.2, fc='purple', ec='purple',
linewidth=3, zorder=4)
plt.text(7.5, 6.2, '100 Ton', color='purple', fontweight='bold', fontsize=11,
zorder=4)
```

Configuración final del gráfico

```
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal') # Mantener relación de aspecto 1:1
plt.tight_layout()
plt.show()
```

```
# =====
# ENSAMBLE DE LA MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA
# =====
```

```
print("\n=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA ===")
# Inicializar matriz de rigidez global (12x12 para 4 nodos x 3 GL)
kG = np.zeros((GL, GL))
```

```
# Listas para almacenar matrices de cada elemento (para post-procesamiento)
kL_elementos = [] # Matrices Locales
T_elementos = [] # Matrices de transformación
```

```
for i in range(m): # Para cada elemento (i = 0,1,2...m)
    theta = math.radians(a[i]) # Convertir ángulo a radianes
    c = math.cos(theta) # coseno del ángulo
    s = math.sin(theta) # seno del ángulo

    AE = A[i] * E # Rigidez axial (EA)
    EI = E * I[i] # Rigidez flexional (EI)
    L2 = (L[i])**2 # Longitud al cuadrado
    L3 = (L[i])**3 # Longitud al cubo

    # Matriz de rigidez Local (6x6) en coordenadas Locales
    kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
          [0, (12*EI)/L3, (6*EI)/L2, 0, -(12*EI)/L3, (6*EI)/L2],
          [0, (6*EI)/L2, (4*EI)/L[i], 0, -(6*EI)/L2, (2*EI)/L[i]],
          [-AE/L[i], 0, 0, AE/L[i], 0, 0],
          [0, -(12*EI)/L3, -(6*EI)/L2, 0, (12*EI)/L3, -(6*EI)/L2],
          [0, (6*EI)/L2, (2*EI)/L[i], 0, -(6*EI)/L2, (4*EI)/L[i]]]

    kL_elementos.append(kL)
```

```
# Matriz de transformación de coordenadas
T = [[c, s, 0, 0, 0, 0],
      [-s, c, 0, 0, 0, 0],
      [0, 0, 1, 0, 0, 0],
      [0, 0, 0, c, s, 0],
      [0, 0, 0, -s, c, 0],
      [0, 0, 0, 0, 0, 1]]

T_elementos.append(T)

T_T = np.transpose(T) # Transpuesta de la matriz de transformación
kg_e = np.matmul(np.matmul(T_T, kL), T) # Matriz de rigidez del elemento en
coordenadas globales

# Ensamblar en matriz global
# Mapear grados de libertad del elemento a posición en matriz global
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]
# Sumar contribución del elemento a la matriz global
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj]

#print(f"ELEMENTO {i+1}")
#print(f"Longitud: {L[i]} m, Ángulo: {a[i]:.3f}°")
#print(f"Área: {A[i]} m, Inercia: {I[i]:.3f}°")
#print(f"Matriz global del elemento: \n {np.round(kg_e, 0)}")
#print(f"Matriz global del sistema después del elemento {i+1}: \n {np.round(kG,
0)}")

print(f"Matriz global del sistema: \n{np.round(kG, 0)}")

# =====
# VECTOR DE FUERZAS DE EMPOTRAMIENTO PERFECTO (FEM)
# =====
print("\n=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===")
FEM_L_elementos = [] # FEM en coordenadas Locales
FEM_G = np.zeros(GL) # FEM en coordenadas globales

for i in range(m): # Para cada elemento
    # Extraer información de carga para este elemento
    tipo, direccion, color, wi, wf, *extra = Cargas[i]

    # Calcular FEM según tipo de carga
    if tipo == 'Uniforme' and direccion == 'Local_y':
```

```
# Carga uniforme/trapezoidal en dirección Y Local (perpendicular al elemento)
wi = -wi # Carga en nodo inicial (Ton/m)
wf = -wf # Carga en nodo final (Ton/m)

# Fórmulas para viga con carga trapezoidal
Ryi = ((7*wi*L[i])/20) + ((3*wf*L[i])/20) # Reacción en nodo inicial
Mzi = ((wi*(L[i]**2))/20) + ((wf*(L[i]**2))/30) # Momento en nodo inicial
Ryf = ((3*wi*L[i])/20) + ((7*wf*L[i])/20) # Reacción en nodo final
Mzf = -(((wi*(L[i]**2))/30) + ((wf*(L[i]**2))/20)) # Momento en nodo final

FEM_L = [0, Ryi, Mzi, 0, Ryf, Mzf] # [Fx, Fy, Mz, Fx, Fy, Mz] en Locales

elif tipo == 'Uniforme' and direccion == 'Global_x':
    # Carga uniforme en dirección X global
    Theta = math.radians(a[i]) # Ángulo del elemento
    c = extra[0] # Proyección vertical sobre la que actúa la carga (6m)
    # Descomponer la carga global en componentes Locales (axial y perpendicular)
    # El factor (c/L[i]) distribuye la carga, que actúa en una proyección, a lo
    largo del elemento.

    wxi = wi * math.cos(Theta) * (c/L[i]) # Componente X en nodo i
    wyi = -wi * math.sin(Theta) * (c/L[i]) # Componente Y en nodo i
    wxf = wf * math.cos(Theta) * (c/L[i]) # Componente X en nodo f
    wyf = -wf * math.sin(Theta) * (c/L[i]) # Componente Y en nodo f

    # FEM para carga trapezoidal en coordenadas Locales
    Rxi = -(((7*wxi*L[i])/20) + ((3*wxf*L[i])/20)) # Fuerza X en nodo i
    Ryi = -(((7*wyi*L[i])/20) + ((3*wyf*L[i])/20)) # Fuerza Y en nodo i
    Mzi = -(((wyi*(L[i]**2))/20) + ((wyf*(L[i]**2))/30)) # Momento en nodo i
    Rxf = -(((3*wxi*L[i])/20) + ((7*wxf*L[i])/20)) # Fuerza X en nodo f
    Ryf = -(((3*wyi*L[i])/20) + ((7*wyf*L[i])/20)) # Fuerza Y en nodo f
    Mzf = (((wyi*(L[i]**2))/30) + ((wyf*(L[i]**2))/20)) # Momento en nodo f

    FEM_L = [Rxi, Ryi, Mzi, Rxf, Ryf, Mzf]

else:
    # Sin carga distribuida
    FEM_L = [0, 0, 0, 0, 0, 0]

FEM_L_elementos.append(FEM_L)

# Transformar FEM a coordenadas globales: FEM_global = T^T * FEM_Local
FEM_Ge = np.matmul(np.transpose(T_elementos[i]), FEM_L)

# Ensamblar en vector global
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]
```

```

for jj, gl_j in enumerate(GL_elem):
    FEM_G[gl_j] += FEM_Ge[jj]
print(np.round(FEM_G, 2))

# =====
# CONDICIONES DE CONTORNO (RESTRICCIONES)
# =====
print("\n=== RESTRICCIONES ===")
# Vector de restricciones: 1 = restringido, 0 = libre
#
#           GL: 1  2  3  4  5  6  7  8  9 10 11 12
restricciones = np.array([1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0])

for i in range(n): # Para cada nodo
    rx = "Restringido" if restricciones[i*3]==1 else "Libre" # Estado GL x
    ry = "Restringido" if restricciones[i*3+1]==1 else "Libre" # Estado GL y
    rz = "Restringido" if restricciones[i*3+2]==1 else "Libre" # Estado GL z

    print(f"Node {i+1}: Dx={restricciones[i*3]} ({rx}), Dy={restricciones[i*3+1]} ({ry}), Dz={restricciones[i*3+2]} ({rz}")

# =====
# VECTOR DE FUERZAS EXTERNAS
# =====
print("\n=== VECTOR DE FUERZAS EXTERNAS ===")
# Vector de fuerzas nodales: [Fx1, Fy1, Mz1, Fx2, Fy2, Mz2, ..., Fx4, Fy4, Mz4]
#
#           GL: 1  2  3  4  5  6  7  8  9 10 11 12
F = np.array([0, 0, 0, 0, -90, 0, -100, 0, 0, 0, 0, 0])
print(f"Fuerzas externas: {F}")

# =====
# REDUCCIÓN DEL SISTEMA (ELIMINACIÓN DE GL RESTRINGIDOS)
# =====
print("\n=== REDUCCIÓN DEL SISTEMA ===")

# Identificar grados de libertad activos (no restringidos)
GL_activos = np.where(restricciones == 0)[0]
n_GL_activos = len(GL_activos)
print(f"Grados de libertad activos: {n_GL_activos}")
print(f"Índices de GL activos: {GL_activos + 1}") # +1 porque Python indexa desde cero

# Extraer submatriz y subvectores correspondientes a GL activos
K_reducida = kG[np.ix_(GL_activos, GL_activos)] # Submatriz cuadrada
F_reducido = F[GL_activos] # Subvector de fuerzas
FEM_reducido = FEM_G[GL_activos] # Subvector de FEM
print(f"\nMatriz global reducida: \n {np.round(K_reducida, 0)}")
print(f"\nVector de fuerzas reducido: {F_reducido} Ton")
print(f"\nVector FEM reducido: {np.round(FEM_reducido,2)} Ton")

```

```
# =====
# SOLUCIÓN DEL SISTEMA
# =====
print("\n\n==== SOLUCIÓN DEL SISTEMA =====")
K_reducida_inv = np.linalg.inv(K_reducida) # Invertir matriz de rigidez reducida

# Calcular desplazamientos desconocidos:  $K \cdot U = F - FEM \rightarrow U = K^{-1} \cdot (F - FEM)$ 
U_desconocidos = np.matmul(K_reducida_inv, F_reducido - FEM_reducido)

# Reconstruir vector completo de desplazamientos
U_totales = np.zeros(GL) # Inicializar con ceros
U_totales[GL_activos] = U_desconocidos # Insertar valores calculados

# Calcular reacciones:  $R = K \cdot U + FEM$ 
Reacciones = np.matmul(kG, U_totales) + FEM_G

print("\n=== REACCIONES ===")
# Reacciones en apoyos
for i in range(n):
    Rx = Reacciones[i*3] # Reacción en X
    Ry = Reacciones[i*3+1] # Reacción en Y
    Mz = Reacciones[i*3+2] # Reacción momento

    # Mostrar solo en nodos con restricciones
    if (restricciones[i*3]==1 or restricciones[i*3+1]==1 or restricciones[i*3+2]==1):
        print(f"Node {i+1}: Rx = {Rx:.2f} Ton, Ry = {Ry:.2f} Ton, Mz = {Mz:.2f} Ton-m")

print("\n=== DESPLAZAMIENTOS ===")
# Desplazamientos nodales
for i in range(n):
    Ux = U_totales[i*3] # Desplazamiento en X
    Uy = U_totales[i*3+1] # Desplazamiento en Y
    Tz = U_totales[i*3+2] # Rotación en Z

    # Notación científica para valores pequeños
    print(f"Node {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m,  $\theta_z = {Tz:.3e}$  rad")

# =====
# GRÁFICA SISTEMA DEFORMADO
# =====
FS = 10 # Factor de escala para desplazamientos (amplifica para visualización)
plt.figure(figsize=(6, 5))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')
```

```
# Dibujar sistema original (líneas punteadas grises)
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '--', color='grey', linewidth=2, alpha=0.5,
label='Original' if i==0 else "")

# Dibujar sistema deformado (líneas verdes)
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Desplazamientos de los nodos
    desp_i = np.array([
        U_totales[(ni-1)*3],      # Ux nodo i
        U_totales[(ni-1)*3+1],    # Uy nodo i
        U_totales[(ni-1)*3+2]])   # Uz nodo i

    desp_f = np.array([
        U_totales[(nf-1)*3],      # Ux nodo f
        U_totales[(nf-1)*3+1],    # Uy nodo f
        U_totales[(nf-1)*3+2]])   # Uz nodo f

    # Coordenadas deformadas (amplificadas)
    xi_def = xi + desp_i[0] * FS
    yi_def = yi + desp_i[1] * FS
    xf_def = xf + desp_f[0] * FS
    yf_def = yf + desp_f[1] * FS

    # Dibujar elemento deformado
    plt.plot([xi_def, xf_def], [yi_def, yf_def], '-', color='g', linewidth=2,
label='Deformada' if i==0 else "")

# Dibujar apoyos (posición original)
plt.plot(0, 2-0.3, '^', color='maroon', markersize=20, zorder=2)
plt.plot(11, 0-0.3, '^', color='maroon', markersize=20, zorder=2)

# Configuración final del gráfico
plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid(True, alpha=0.5)
plt.axis('equal')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

```
print("\n=== FUERZAS INTERNAS ===")
F_int_elementos = [] # Lista para almacenar fuerzas internas

for i in range(m): # Para cada elemento
    # Extraer desplazamientos globales del elemento
    Ue = np.array([
        U_totales[Nx[i]-1], # Dx nodo inicial
        U_totales[Ny[i]-1], # Dy nodo inicial
        U_totales[Nz[i]-1], # Dz nodo inicial
        U_totales[Fx[i]-1], # Dx nodo final
        U_totales[Fy[i]-1], # Dy nodo final
        U_totales[Fz[i]-1]]) # Dz nodo final

    # Transformar a coordenadas locales: U_local = T · U_global
    Ue_L = np.matmul(T_elementos[i], Ue)

    # Calcular fuerzas internas en coordenadas locales: F_local = k_local · U_local + FEM_local
    Fe_L = np.matmul(kL_elementos[i], Ue_L) + FEM_L_elementos[i]
    F_int_elementos.append(Fe_L)

    # Presentar resultados
    print(f"Elemento {i+1}:\nNodo {Elementos[i][0]}: N = {Fe_L[0]:.3f} Ton, V = {Fe_L[1]:.3f} Ton, M = {Fe_L[2]:.3f} Ton-m")

    print(f"Nodo {Elementos[i][1]}: N = {Fe_L[3]:.3f} Ton, V = {Fe_L[4]:.3f} Ton, M = {Fe_L[5]:.3f} Ton-m\n")

# =====
# DIAGRAMA AXIAL
# =====
plt.figure(figsize=(8,6)) plt.title("DIAGRAMA AXIAL", fontsize=14, fontweight="bold")
escala_axial = 0.01 # Factor de escala para la magnitud de la fuerza axial en el dibujo

for i in range(m):
    # Desempaquetar información geométrica del elemento i
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Desempaquetar información de cargas del elemento i
    tipo, direccion, color, wi, wf, *extra = Cargas[i]

    # Dibujar la línea del elemento
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=3)

    # Fuerzas axiales en los extremos (del cálculo de fuerzas internas). Los índices:
    # 0=axial nodo i, 1=cortante i, 2=momento i, 3=axial j, 4=cortante j, 5=momento j
    Ni = F_int_elementos[i][0] # Axial en nodo inicial (kN)
    Nf = F_int_elementos[i][3] # Axial en nodo final (kN)
```


Caso especial: Carga horizontal global (Global_x) que causa una variación de la fuerza axial a lo largo del elemento. Esto ocurre cuando una carga horizontal se aplica sobre un elemento inclinado, generando componentes axiales variables.

```
if tipo == 'Uniforme' and direccion == 'Global_x':
    # Proyección de la carga global sobre la dirección axial del elemento
    wix = wi * ex * (extra[0]/Le) # Componente axial en nodo i (kN/m)
    wfx = wf * ex * (extra[0]/Le) # Componente axial en nodo j (kN/m)

else:
    wix = 0 # Sin carga axial distribuida
    wfx = 0
```

x = np.linspace(0, Le, 30) # Puntos a lo largo del elemento para evaluar la variación (coordenada local)

Cálculo de la fuerza axial variable N(x) integrando la carga distribuida axial. La integral resulta en: $N(x) = ((wfx - wix)/Le)(x^2/2) + wix*x + Ni$*

```
N = (((wfx - wix)/Le)*((x**2)/2)) + (wix*x) + Ni
```

Coordenadas para dibujar el diagrama: punto sobre el elemento desplazado perpendicularmente

```
X_diag = xi + ex*x + escala_axial * N * nx
Y_diag = yi + ey*x + escala_axial * N * ny
```

Línea base del elemento (para el relleno)

```
X_base = xi + ex*x
Y_base = yi + ey*x
```

Dibujar la línea del diagrama y el relleno semitransparente

```
plt.plot(X_diag, Y_diag, color=color, linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:-1]]),
        np.concatenate([Y_base, Y_diag[:-1]]), alpha=0.4, color=color)
```

Texto en el punto medio del elemento indicando el tipo y magnitud de la fuerza

```
xm = xi + ex * Le / 2
ym = yi + ey * Le / 2
tipo_texto = "Tracción" if Nf>0 else "Compresión" # Tracción(+) o compresión(-)
```

```
plt.text(xm, ym, f"{Nf:.2f} kN\n({tipo_texto})", fontsize=8, ha='center',
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
```

```
plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

```
# =====
# DIAGRAMA DE FUERZA CORTANTE
# =====
escala_cortante = 0.008 # Factor de escala para la magnitud del cortante en el
dibujo plt.figure(figsize=(8,6))
plt.title("DIAGRAMA CORTANTE", fontsize=14, fontweight="bold")

for i in range(m):
    # Desempaquetar información geométrica del elemento i
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Desempaquetar información de cargas
    tipo, direccion, color, wi, wf, *extra = Cargas[i]

    # Dibujar la línea del elemento como referencia
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Fuerzas cortantes en los extremos (del cálculo de fuerzas internas)
    Vi = F_int_elementos[i][1] # Cortante en nodo inicial (kN)
    Vf = F_int_elementos[i][4] # Cortante en nodo final (kN)

    # Determinación de la componente de carga perpendicular al elemento (dirección
    Local Y) que es la que genera esfuerzo cortante

    # Caso 1: Carga directamente perpendicular al elemento (Local_y)
    if tipo == 'Uniforme' and direccion == 'Local_y':
        wyi = wi # Componente perpendicular en nodo i (kN/m)
        wyf = wf # Componente perpendicular en nodo j (kN/m)

    # Caso 2: Carga horizontal global (Global_x) sobre elemento inclinado
    elif tipo == 'Uniforme' and direccion == 'Global_x':
        # La carga global debe proyectarse sobre la dirección perpendicular al
        elemento. a[i] es el ángulo de inclinación del elemento (en grados), extra[0] es la
        longitud de aplicación. El signo negativo ajusta la dirección según convención
        wyi = -wi * math.sin(math.radians(a[i])) * (extra[0]/Le)
        wyf = -wf * math.sin(math.radians(a[i])) * (extra[0]/Le)

    else:
        wyi = 0 # Sin carga distribuida
        wyf = 0

    x = np.linspace(0, Le, 30) # Puntos de evaluación a lo largo del elemento

    # Cálculo de la fuerza cortante variable  $V(x) = \int w_y(x) dx + V_i$ 
    # La integral resulta en:  $V(x) = ((wyf - wyi)/Le)*(x^2/2) + wyi*x + Vi$ 
    V = (((wyf - wyi)/Le)*((x**2)/2)) + (wyi*x) + Vi
```

```
# Coordenadas del diagrama (desplazamiento perpendicular)
X_diag = xi + ex*x + escala_cortante * V * nx
Y_diag = yi + ey*x + escala_cortante * V * ny

# Línea base del elemento
X_base = xi + ex*x
Y_base = yi + ey*x

# Dibujar diagrama con relleno
plt.plot(X_diag, Y_diag, color=color, linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:-1]]),
        np.concatenate([Y_base, Y_diag[:-1]]), alpha=0.4, color=color)

# Etiquetas con los valores de cortante en los extremos se muestra -Vf en el
# extremo j por convención gráfica
plt.text(X_diag[0], Y_diag[0], f"{Vi:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
plt.text(X_diag[-1], Y_diag[-1], f"{-Vf:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# =====
# DIAGRAMA DE MOMENTO FLECTOR
# =====
escala_momento = 0.007 # Factor de escala para la magnitud del momento en el dibujo
plt.figure(figsize=(8,6))
plt.title("DIAGRAMA MOMENTO", fontsize=14, fontweight="bold")

for i in range(m):
    # Desempaquetar información geométrica del elemento i
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Desempaquetar información de cargas
    tipo, direccion, color, wi, wf, *extra = Cargas[i]

    # Dibujar la línea del elemento como referencia
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Fuerzas y momentos en los extremos (necesarios para la integración)
    Vi = F_int_elementos[i][1] # Cortante en nodo inicial (kN)
    Vf = F_int_elementos[i][4] # Cortante en nodo final (kN)
    Mi = F_int_elementos[i][2] # Momento en nodo inicial (kN-m)
    Mf = F_int_elementos[i][5] # Momento en nodo final (kN-m)
```

```
# Determinación de la componente de carga perpendicular (igual que en cortante)
if tipo == 'Uniforme' and direccion == 'Local_y':
    wyi = wi
    wyf = wf

elif tipo == 'Uniforme' and direccion == 'Global_x':
    # Proyección de carga global sobre dirección perpendicular al elemento
    wyi = -wi * math.sin(math.radians(a[i])) * (extra[0]/Le)
    wyf = -wf * math.sin(math.radians(a[i])) * (extra[0]/Le)

else:
    wyi = 0
    wyf = 0

x = np.linspace(0, Le, 30) # Puntos de evaluación a lo largo del elemento

# Integración de V(x) (que ya incluye wyi, wyf) para obtener M(x)
M = (((wyf - wyi)/Le)*((x**3)/6)) + (wyi*((x**2)/2)) + Vi*x - Mi

# Coordenadas del diagrama (desplazamiento perpendicular)
X_diag = xi + ex*x + escala_momento * M * nx
Y_diag = yi + ey*x + escala_momento * M * ny

# Línea base del elemento
X_base = xi + ex*x
Y_base = yi + ey*x

# Dibujar diagrama con relleno
plt.plot(X_diag, Y_diag, color=color, linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:-1]]),
         np.concatenate([Y_base, Y_diag[:-1]]), alpha=0.4, color=color)

# Etiquetas con los valores de momento en los extremos.
# Se muestra -Mi en el extremo i para mantener la convención de que el momento se
# dibuja. El signo positivo en Mf se muestra directamente.
plt.text(X_diag[0], Y_diag[0], f"{-Mi:.2f}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
plt.text(X_diag[-1], Y_diag[-1], f"{Mf:.2f}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

MÉTODO MATRICIAL DE RIGIDEZ PARA PÓRTICOS - EJERCICIO 2

=== DATOS DE ENTRADA ===

Módulo de elasticidad: 2302520.3582 kN/m²

Área de la sección transversal: [0.105 0.105 0.105] m²

Longitud: [5. 4. 7.21] m

Inercia: [0.0011 0.0011 0.0011] m

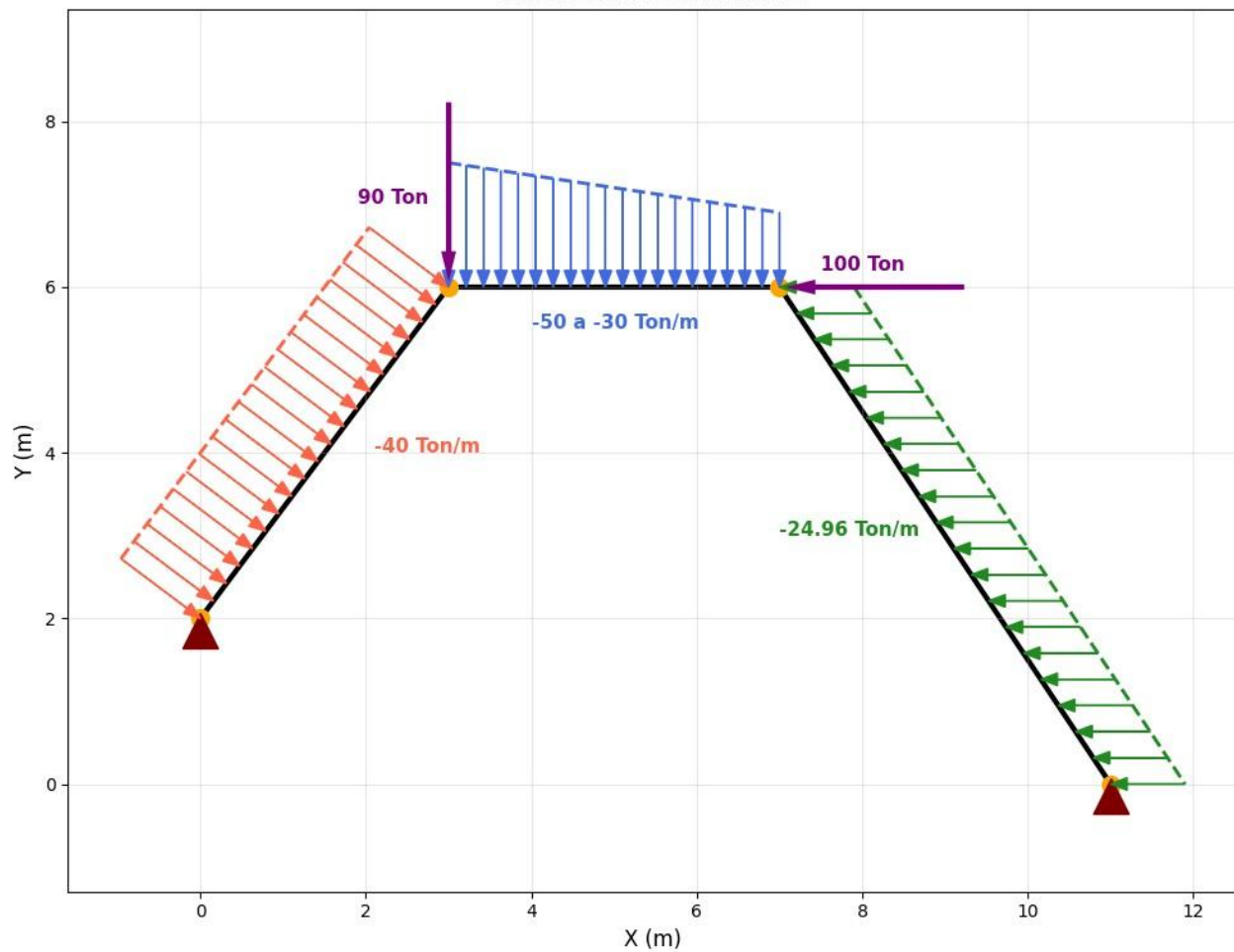
Ángulo de inclinación: [53.13 0. -56.31] °

Número de elementos: 3

Número de nodos: 4

Número de grados de libertad: 12

SISTEMA ORIGINAL



=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA ===

Matriz global del sistema:

```
[ [ 17559. 23096. -474. -17559. -23096. -474. 0. 0. 0. 0.
0. 0. ]
[ 23096. 31031. 355. -23096. -31031. 355. 0. 0. 0. 0.
0. 0. ]
[ -474. 355. 1974. 474. -355. 987. 0. 0. 0. 0.
0. 0. ]
[ -17559. -23096. 474. 78000. 23096. 474. -60441. 0. 0. 0.
0. 0. ]
[ -23096. -31031. -355. 23096. 31494. 570. 0. -463. 926. 0.
0. 0. ]
[ -474. 355. 987. 474. 570. 4442. 0. -926. 1234. 0.
0. 0. ]
[ 0. 0. 0. -60441. 0. 0. 70812. -15437. 237. -10371.
15437. 237. ]
[ 0. 0. 0. 0. -463. -926. -15437. 23698. -768. 15437. -
23235. 158. ]
[ 0. 0. 0. 0. 926. 1234. 237. -768. 3837. -237.
-158. 685. ]
[ 0. 0. 0. 0. 0. 0. -10371. 15437. -237. 10371. -
15437. -237. ]
[ 0. 0. 0. 0. 0. 0. 15437. -23235. -158. -15437.
23235. -158. ]
[ 0. 0. 0. 0. 0. 0. 237. 158. 685.
-237. -158. 1369. ] ]
```

=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===

```
[ -80. 60. 83.33 -80. 148. -27.33 90. 72. 39.33 90. 0. -90. ]
```

=== RESTRICCIONES ===

Nodo 1: Dx=1 (Restringido), Dy=1 (Restringido), Dz=0 (Libre)

Nodo 2: Dx=0 (Libre), Dy=0 (Libre), Dz=0 (Libre)

Nodo 3: Dx=0 (Libre), Dy=0 (Libre), Dz=0 (Libre)

Nodo 4: Dx=1 (Restringido), Dy=1 (Restringido), Dz=0 (Libre)

=== VECTOR DE FUERZAS EXTERNAS ===

```
Fuerzas externas: [ 0 0 0 0 -90 0 -100 0 0 0 0 0 ]
```

=== REDUCCIÓN DEL SISTEMA ===

Grados de libertad activos: 8

Índices de GL activos: [3 4 5 6 7 8 9 12]

Matriz global reducida:

```
[ [ 1974.  474. -355.  987.   0.   0.   0.   0.]
 [  474. 78000. 23096.  474. -60441.   0.   0.   0.]
 [ -355. 23096. 31494.  570.   0. -463.  926.   0.]
 [  987.  474.  570. 4442.   0. -926. 1234.   0.]
 [   0. -60441.   0.   0. 70812. -15437.  237.  237.]
 [   0.   0. -463. -926. -15437. 23698. -768.  158.]
 [   0.   0.  926. 1234.  237. -768. 3837.  685.]
 [   0.   0.   0.   0.  237.  158.  685. 1369.]]
```

Vector de fuerzas reducido: [0 0 -90 0 -100 0 0 0] Ton

Vector FEM reducido: [83.33 -80. 148. -27.33 90. 72. 39.33 -90.] Ton

===== SOLUCIÓN DEL SISTEMA =====

=== REACCIONES ===

Nodo 1: Rx = 105.30 Ton, Ry = 285.10 Ton, Mz = -0.00 Ton-m

Nodo 4: Rx = 14.70 Ton, Ry = 84.90 Ton, Mz = -0.00 Ton-m

=== DESPLAZAMIENTOS ===

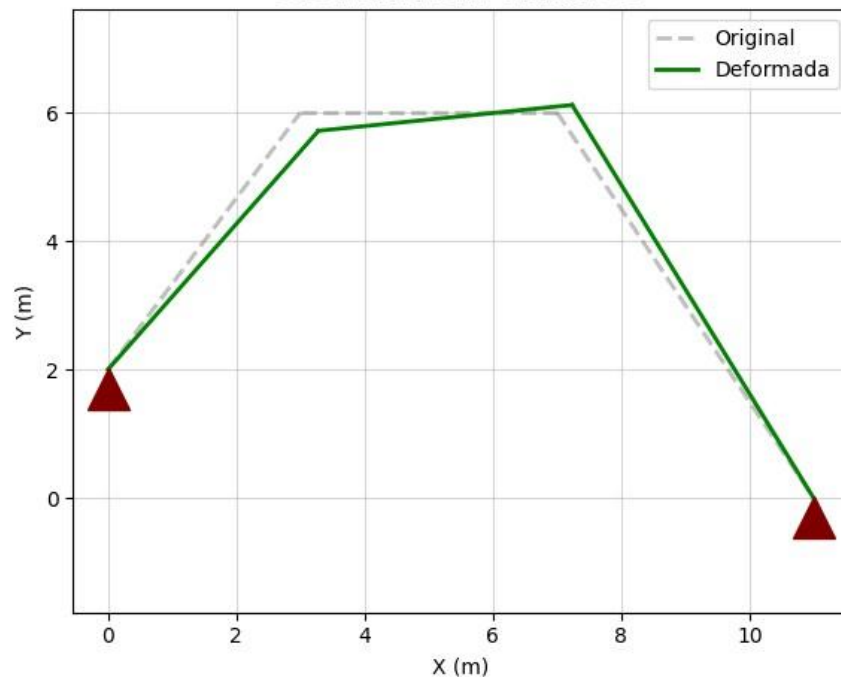
Nodo 1: Ux = 0.000e+00 m, Uy = 0.000e+00 m, $\theta_z = -7.006e-02$ rad

Nodo 2: Ux = 2.798e-02 m, Uy = -2.851e-02 m, $\theta_z = 3.201e-02$ rad

Nodo 3: Ux = 2.359e-02 m, Uy = 1.170e-02 m, $\theta_z = -2.585e-02$ rad

Nodo 4: Ux = 0.000e+00 m, Uy = 0.000e+00 m, $\theta_z = 7.323e-02$ rad

SISTEMA DEFORMADO



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 291.256 \text{ Ton}$, $V = 86.820 \text{ Ton}$, $M = -0.000 \text{ Ton-m}$

Nodo 2: $N = -291.256 \text{ Ton}$, $V = 113.180 \text{ Ton}$, $M = -65.898 \text{ Ton-m}$

Elemento 2:

Nodo 2: $N = 265.298 \text{ Ton}$, $V = 75.097 \text{ Ton}$, $M = 65.898 \text{ Ton-m}$

Nodo 3: $N = -265.298 \text{ Ton}$, $V = 84.903 \text{ Ton}$, $M = -112.175 \text{ Ton-m}$

Elemento 3:

Nodo 3: $N = 162.334 \text{ Ton}$, $V = 90.440 \text{ Ton}$, $M = 112.175 \text{ Ton-m}$

Nodo 4: $N = -62.488 \text{ Ton}$, $V = 59.329 \text{ Ton}$, $M = -0.000 \text{ Ton-m}$

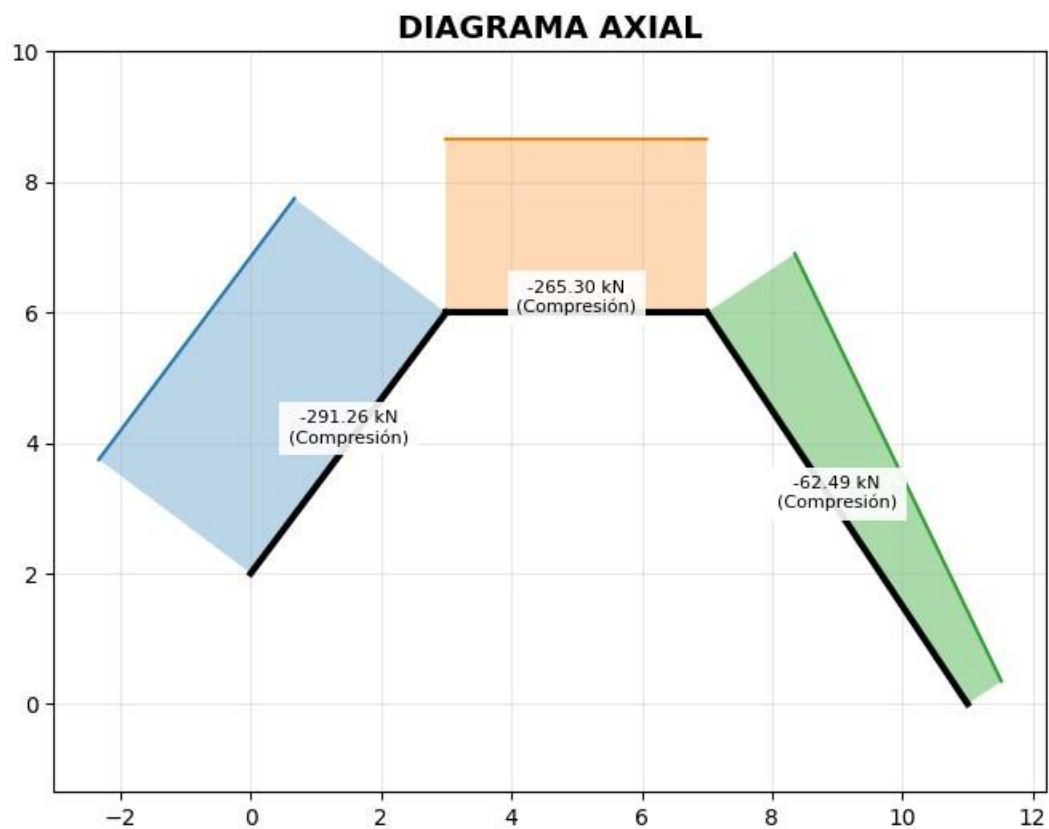


DIAGRAMA CORTANTE

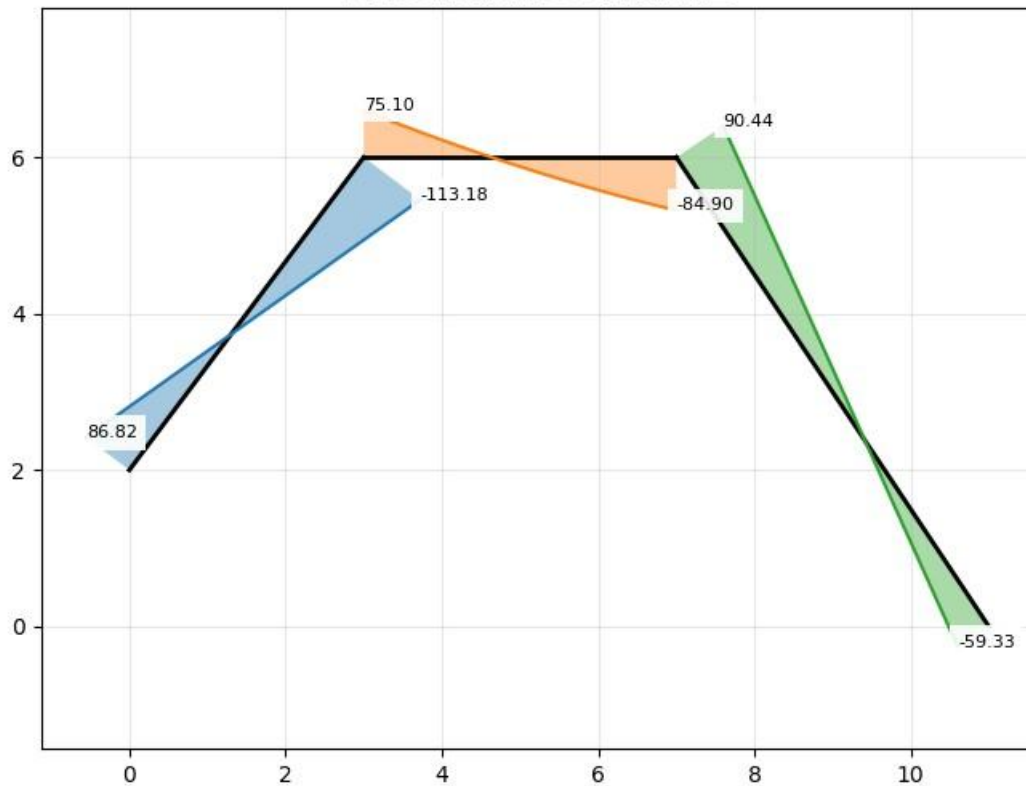
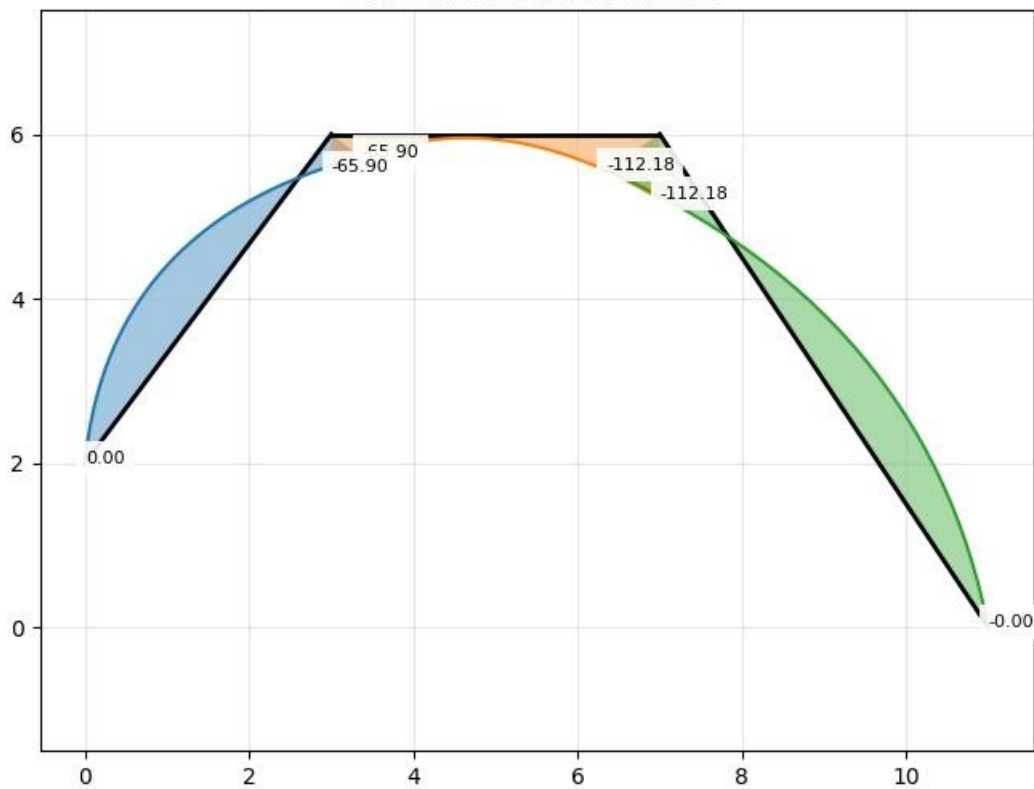


DIAGRAMA MOMENTO



MODELO OPENSEES

PÓRTICOS: EJERCICIO 2

```
# =====
# IMPORTACIÓN DE LIBRERÍAS
# =====
import openseespy.opensees as ops  # Biblioteca principal de OpenSees para análisis estructural
import opsviz as opsv              # Librería para visualización de resultados y diagramas
import numpy as np                 # Para operaciones numéricas y arreglos
import math                        # Para funciones matemáticas básicas (atan, sqrt)
import matplotlib.pyplot as plt    # Para visualización de resultados personalizada

print("MODELO EN OPENSEES DEL EJERCICIO 2 - PÓRTICOS")

# =====
# INICIALIZACIÓN DEL MODELO
# =====
ops.wipe() # Limpia cualquier modelo previo en memoria para empezar desde cero

# Crear modelo básico 2D con 3 grados de libertad por nodo (dx, dy, rotación)
# '-ndm': número de dimensiones espaciales (2D)
# '-ndf': número de grados de libertad por nodo (3:desplazamientos X, Y y rotación Z)
ops.model('basic', '-ndm', 2, '-ndf', 3)

# =====
# DEFINICIÓN DE PROPIEDADES DEL MATERIAL Y GEOMETRÍA
# =====
# Propiedades de la sección transversal (unidades: Ton, m)
E = 2302520.3582 # Módulo de elasticidad (Ton/m²)

# --- Propiedades de la sección transversal única para todos los elementos ---
A = 0.105 # Área de la sección (m²) - 0.30m x 0.35m
I = 0.001071875 # Momento de inercia (m⁴) - Para sección rectangular: (b*h³)/12

# =====
# DEFINICIÓN DE NODOS
# =====
# ops.node(etiqueta_nodo, coordenada_x, coordenada_y) en metros
ops.node(1, 0, 2) # Nodo 1
ops.node(2, 3, 6) # Nodo 2
ops.node(3, 7, 6) # Nodo 3
ops.node(4, 11, 0) # Nodo 4
```

```
# =====
# CONDICIONES DE APOYO
# =====
# ops.fix(etiqueta_nodo, restricción_x, restricción_y, restricción_rotación)
# 1 = restringido (empotrado), 0 = Libre (móvil)
ops.fix(1, 1, 1, 0) # Nodo 1: empotrado en X e Y, Libre en rotación (apoyo fijo)
ops.fix(4, 1, 1, 0) # Nodo 4: empotrado en X e Y, Libre en rotación (apoyo fijo)

# =====
# DEFINICIÓN DE SECCIÓN ELÁSTICA
# =====
# 'Elastic': material elástico lineal con propiedades E, A, I
# Parámetros: (tipo_sección, etiqueta, E, A, I)
ops.section('Elastic', 1, E, A, I) # Sección única con tag 1 para todos los
elementos

# =====
# TRANSFORMACIÓN GEOMÉTRICA
# =====
# 'Linear': transformación lineal para pequeñas deformaciones (asume pequeños
desplazamientos)
# Parámetros: (tipo_transformación, etiqueta)
ops.geomTransf('Linear', 1) # Transformación lineal con tag 1

# =====
# ESQUEMA DE INTEGRACIÓN
# =====
# 'Lobatto': integración de Lobatto con puntos en los extremos y interior
# Parámetros: (tipo_integración, etiqueta, etiqueta_sección,
número_puntos_integración) 10 puntos de integración para alta precisión en la
respuesta del elemento
ops.beamIntegration('Lobatto', 1, 1, 10)

# =====
# CREACIÓN DE ELEMENTOS
# =====
# Lista para almacenar información de elementos (para post-procesamiento y
visualización)
Elementos = []

# --- Elemento 1: Barra inclinada (nodo 1 a nodo 2) ---
# 'dispBeamColumn': elemento viga-columna basado en formulación por desplazamientos
# Parámetros: (tipo_elemento, etiqueta, nodo_i, nodo_j, tag_transformación,
tag_integración)
ops.element('dispBeamColumn', 1, 1, 2, 1, 1)
Elementos.append({"ID": 1, "Nodo_i": 1, "Nodo_j": 2}) # Guardar info para post-
proceso
```

```
# --- Elemento 2: Barra horizontal con carga trapezoidal ---
# Para modelar correctamente la carga variable, se discretiza el elemento en muchos
sub-elementos
nDiv2 = 50 # Número de subdivisiones (mayor número = mejor aproximación de carga variable)
L2 = 4.0   # Longitud total del elemento 2

# Crear nodos intermedios para la discretización
nodes2 = [2] # Lista de nodos, comenzando con el nodo 2 existente

for i in range(1, nDiv2):
    # Interpolan coordenadas entre nodo 2 (x=3, y=6) y nodo 3 (x=7, y=6)
    x = 3 + (7 - 3) * i / nDiv2 # División equiespaciada en X
    y = 6 # Altura constante (elemento horizontal)
    ops.node(100 + i, x, y)      # Crear nodo intermedio con etiqueta 101, 102, ...
    nodes2.append(100 + i)      # Agregar a la lista de nodos
nodes2.append(3) # Finalizar con nodo 3 existente

# Crear sub-elementos entre nodos consecutivos
elems2 = [] # Lista para almacenar etiquetas de sub-elementos
for i in range(nDiv2):
    eTag = 200 + i # Etiqueta única para cada sub-elemento (200, 201, 202, ...)

    # Crear elemento entre nodo i y nodo i+1 de la lista nodes2
    ops.element('dispBeamColumn', eTag, nodes2[i], nodes2[i + 1], 1, 1)

    elems2.append(eTag) # Guardar referencia del sub-elemento

# Registrar elemento 2 en la lista principal (solo como elemento conceptual para
post-proceso)
Elementos.append({"ID": 2, "Nodo_i": 2, "Nodo_j": 3})

# --- Elemento 3: Barra inclinada (nodo 3 a nodo 4) ---
ops.element('dispBeamColumn', 3, 3, 4, 1, 1)
Elementos.append({"ID": 3, "Nodo_i": 3, "Nodo_j": 4})

# =====
# APLICACIÓN DE CARGAS
# =====
# Configurar patrón de carga estática
# 'Linear': serie temporal lineal que va de 0 a 1 en un paso
ops.timeSeries('Linear', 1) # Serie temporal con tag 1

# 'Plain': patrón de carga simple asociado a una serie temporal
ops.pattern('Plain', 1, 1) # Patrón con tag 1 asociado a serie temporal tag 1
```

```
# --- Cargas distribuidas en elementos ---
# Elemento 1: Carga uniforme vertical en dirección Local Y
w1 = -40 # Carga distribuida en Ton/m (negativo = hacia abajo en sistema Local)

# '-beamUniform': carga uniforme en el elemento. Los valores son (carga_local_y,
carga_local_x)
ops.eleLoad('-ele', 1, '-type', '-beamUniform', w1, 0)

# Elemento 2: Carga trapezoidal (varía linealmente de nodo 2 a nodo 3)
w2i = -50 # Carga inicial (en nodo 2) en Ton/m
w2f = -30 # Carga final (en nodo 3) en Ton/m

# Aplicar carga uniforme por tramo para aproximar la variación trapezoidal
for idx, eTag in enumerate(elems2):
    # Posición Local del inicio del sub-elemento (desde nodo 2)
    x_local = idx * (L2 / nDiv2)

    # Calcular carga en extremos del sub-elemento (interpolación lineal)
    wa = w2i + (w2f - w2i) * x_local / L2 # Carga al inicio del sub-elemento
    wb = w2i + (w2f - w2i) * (x_local + L2 / nDiv2) / L2 # Carga al final del sub-
    elemento

    # Usar el promedio como carga uniforme en el sub-elemento (aproximación)
    w_prom = (wa + wb) / 2

    # Aplicar carga uniforme al sub-elemento
    ops.eleLoad('-ele', eTag, '-type', '-beamUniform', w_prom, 0)

# Elemento 3: Carga distribuida en dirección global X (barra inclinada)
w3 = -30 # Magnitud de carga distribuida en Ton/m
t3 = 6 # Longitud de proyección vertical de la carga (información adicional)
a3 = math.atan(6/4) # Ángulo de inclinación del elemento 3
L3 = math.sqrt((6**2) + (4**2)) # Longitud real del elemento 3

# Descomponer carga distribuida en componentes globales: Se calcula la componente en
cada dirección global basada en la orientación del elemento
w3_x = (w3 * t3 * math.cos(a3)) / L3 # Componente en dirección X global
w3_y = (w3 * t3 * math.sin(a3)) / L3 # Componente en dirección Y global

# Aplicar carga con componentes en ambos ejes globales
ops.eleLoad('-ele', 3, '-type', 'beamUniform', w3_y, w3_x)

# --- Cargas concentradas en nodos ---
# ops.load(etiqueta_nodo, fuerza_x, fuerza_y, momento_z)
ops.load(2, 0, -90, 0) # Nodo 2: 90 Ton hacia abajo (solo componente Y negativa)
ops.load(3, -100, 0, 0) # Nodo 3: 100 Ton hacia la izquierda (solo componente X
negativa)
```

```
# =====
# GRÁFICA SISTEMA ORIGINAL
# =====
# Definir información de cargas para visualización personalizada
# Cada elemento tiene: [tipo, dirección, color, valor_inicial, valor_final, *extra]
Cargas = [
    ['Uniforme', 'Local_y', 'navy', -40, -40], # Elemento 1: uniforme vertical
    ['Uniforme', 'Local_y', 'teal', -50, -30], # Elemento 2: trapezoidal vertical
    ['Uniforme', 'Global_x', 'darkorange', -30, -30, 6]] # Elemento 3: global X con
proyección 6

# Configurar figura para visualización
plt.figure(figsize=(10, 8))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos estructurales (líneas de los elementos)
for element_data in Elementos:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos desde OpenSees
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Dibujar la barra (línea negra sólida)
    plt.plot([xi, xf], [yi, yf], 'k-', lw=2, zorder=1)

# Dibujar nodos principales (puntos púrpura)
for i in range(1, 5):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'o', color='purple', markersize=7, zorder=2)

# Dibujar símbolos de apoyos fijos (triángulos marrones)
plt.plot(0, 2 - 0.15, '^', color='maroon', markersize=20, zorder=3)
plt.plot(11, 0 - 0.15, '^', color='maroon', markersize=20, zorder=3)

# --- Dibujo de cargas distribuidas con flechas ---
escala = 0.03 # Factor de escala para visualización (tamaño de las flechas)

for i, element_data in enumerate(Elementos):
    # Extraer información de carga para este elemento
    tipo, direccion, color, w1, w2, *extra = Cargas[i]

    # Coordenadas de los nodos del elemento
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)
```

```
# Vector del elemento y su longitud
dx = xf - xi
dy = yf - yi
L_elem = (dx ** 2 + dy ** 2) ** 0.5

# Vectores unitarios
ex = dx / L_elem # Vector unitario en dirección del elemento (local x)
ey = dy / L_elem # Vector unitario perpendicular? (realmente es el mismo)

# Vector normal (perpendicular) para cargas en dirección local y
# Rotación de 90°: (ex, ey) -> (-ey, ex)
nx = -ey
ny = ex

# Puntos a lo largo del elemento para dibujar flechas
t_vals = np.linspace(0, 1, 20) # 20 puntos uniformemente espaciados
# Listas para dibujar línea superior de carga
x_superior = []
y_superior = []

for t in t_vals:
    # Punto a lo largo del elemento (coordenada global)
    x = xi + t * dx
    y = yi + t * dy

    # Magnitud de carga en este punto (interpolación lineal entre w1 y w2)
    w = -(w1 + (w2 - w1) * t) # Negativo para dirección correcta en
visualización

# Calcular posición de la base de la flecha según dirección de la carga
if direccion == "Local_y":
    # Carga en dirección local perpendicular
    x_base = x + escala * w * nx
    y_base = y + escala * w * ny

elif direccion == "Global_x":
    # Carga en dirección global X (horizontal)
    x_base = x + escala * w # Desplazamiento solo en X
    y_base = y # Y sin cambio

# Dibujar flecha individual desde la base hasta el elemento
plt.arrow(x_base, y_base, x - x_base, y - y_base, head_width=0.15,
head_length=0.2, fc=color, ec=color, length_includes_head=True, zorder=4)

# Guardar puntos para línea superior (puntas de flechas)
x_superior.append(x_base)
y_superior.append(y_base)
```

```
# Dibujar línea que une las puntas de las flechas (contorno de la carga)
plt.plot(x_superior, y_superior, '--', color=color, linewidth=2, alpha=0.7,
zorder=4)

# --- Etiquetas de cargas y anotaciones ---

# Etiquetas para cargas distribuidas
plt.text(2.1, 4, '-40 Ton/m', color='navy', fontsize=11, fontweight='bold', zorder=5)
plt.text(4, 5.5, '-50 a -30 Ton/m', color='teal', fontsize=11, fontweight='bold',
zorder=5)
plt.text(7, 3, f'{Cargas[2][3] * (Cargas[2][5] / L3):.2f} Ton/m',
color='darkorange', fontsize=11, fontweight='bold', zorder=5)

# Cargas concentradas (flechas verdes)
# Carga vertical en nodo 2
plt.arrow(3, 8.2, 0, -1.8, head_width=0.1, head_length=0.2,
fc='green', ec='green', linewidth=3, zorder=5)
plt.text(1.9, 7, '90 Ton', color='green', fontweight='bold', fontsize=11, zorder=5)

# Carga horizontal en nodo 3
plt.arrow(9.2, 6, -1.8, 0, head_width=0.1, head_length=0.2,
fc='green', ec='green', linewidth=3, zorder=5)
plt.text(7.5, 6.2, '100 Ton', color='green', fontweight='bold', fontsize=11,
zorder=5)

# Configuración final del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal') # Mantener proporciones reales (misma escala en X e Y)
plt.legend()
plt.tight_layout()
plt.show()

# =====
# CONFIGURACIÓN DEL ANÁLISIS
# =====

# Configurar el sistema de ecuaciones y método de solución
ops.system('BandSPD') # Almacena la matriz de rigidez en formato de banda
simétrica definida positiva.
ops.numberer('RCM') # Algoritmo Reverse Cuthill-McKee: reordenamiento de nodos para
reducir el ancho de banda de la matriz de rigidez.
ops.constraints('Plain') # Método simple para imponer restricciones (apoyos)
ops.integrator('LoadControl', 1.0) # Control por carga: aplica el 100% de la carga
en un paso
ops.algorithm('Linear') # Algoritmo lineal para sistemas de ecuaciones (suficiente
para análisis elástico)
ops.analysis('Static') # Tipo de análisis: estático
ops.analyze(1) # Ejecutar análisis con 1 paso de carga
```



```
# =====
# RESULTADOS
# =====
print("\n===== RESULTADOS =====")
# Calcular reacciones en apoyos (debe llamarse antes de consultar reacciones)
ops.reactions()

print("\n=== REACCIONES EN APOYOS ===")
for nodo in [1, 4]:
    Rx = ops.nodeReaction(nodo, 1) # Reacción en dirección X (gdl 1)
    Ry = ops.nodeReaction(nodo, 2) # Reacción en dirección Y (gdl 2)
    Rz = ops.nodeReaction(nodo, 3) # Reacción momento (gdl 3)
    print(f"Node {nodo}: Rx={Rx:.2f} Ton, Ry={Ry:.2f} Ton, Mz={Rz:.2f} Ton-m")

print("\n=== DESPLAZAMIENTOS NODALES ===")
for i in range(1, 5):
    ux = ops.nodeDisp(i, 1) # Desplazamiento en X (gdl 1)
    uy = ops.nodeDisp(i, 2) # Desplazamiento en Y (gdl 2)
    uz = ops.nodeDisp(i, 3) # Rotación (θz, gdl 3)
    print(f"Node {i}: Ux={ux:.3e} m, Uy={uy:.3e} m, θz={uz:.3e} rad")

# =====
# GRÁFICA SISTEMA DEFORMADO
# =====
fig, ax = plt.subplots(figsize=(8, 6))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# El parámetro 'ax' le dice a plot_defo() que use nuestros ejes existentes
opsv.plot_defo(ax=ax)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

# =====
# FUERZAS INTERNAS EN ELEMENTOS
# =====
print("\n=== FUERZAS INTERNAS EN ELEMENTOS ===")
for element_data in Elementos:
    eleTag = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nj = element_data["Nodo_j"]

    # Elemento 2 requiere tratamiento especial por estar discretizado
    if eleTag == 2:
        # Obtener fuerzas en primer sub-elemento (cerca de nodo 2)
        F2_ini = ops.eleResponse(elems2[0], 'localForces')
```

```
# Obtener fuerzas en último sub-elemento (cerca de nodo 3)
F2_fin = ops.eleResponse(elems2[-1], 'localForces')

print("Elemento 2 :")
print(f"Nodo 2: N = {F2_ini[0]:.3f} Ton, V = {F2_ini[1]:.3f} Ton, M = {F2_ini[2]:.3f} Ton-m")
print(f"Nodo 3: N = {F2_fin[3]:.3f} Ton, V = {F2_fin[4]:.3f} Ton, M = {F2_fin[5]:.3f} Ton-m\n")

else:
    # Elementos 1 y 3 (completos, no discretizados). 'localForces' devuelve: [N_i, V_i, M_i, N_j, V_j, M_j] en coordenadas locales
    F_int = ops.eleResponse(eleTag, 'localForces')
    print(f"Elemento {eleTag}:")
    print(f"Nodo {Ni}: N = {F_int[0]:.3f} Ton, V = {F_int[1]:.3f} Ton, M = {F_int[2]:.3f} Ton-m")
    print(f"Nodo {Nj}: N = {F_int[3]:.3f} Ton, V = {F_int[4]:.3f} Ton, M = {F_int[5]:.3f} Ton-m\n")

# =====
# DIAGRAMAS DE ESFUERZOS
# =====
# --- Diagrama de Fuerza Axial ---
fig_n = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA AXIAL (Ton)', fontsize=14, fontweight='bold')
ax_n = plt.gca()

# sf_type='N': diagrama de axial
# sfac: factor de escala para visualización
# nep: número de puntos de evaluación por elemento
opsv.section_force_diagram_2d(sf_type='N', sfac=0.008, nep=20, ax=ax_n)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

# --- Diagrama de Fuerza Cortante ---
fig_v = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA CORTANTE (Ton)', fontsize=14, fontweight='bold')
ax_v = plt.gca()

# sf_type='V': diagrama de cortante
opsv.section_force_diagram_2d(sf_type='V', sfac=0.01, nep=20, ax=ax_v)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

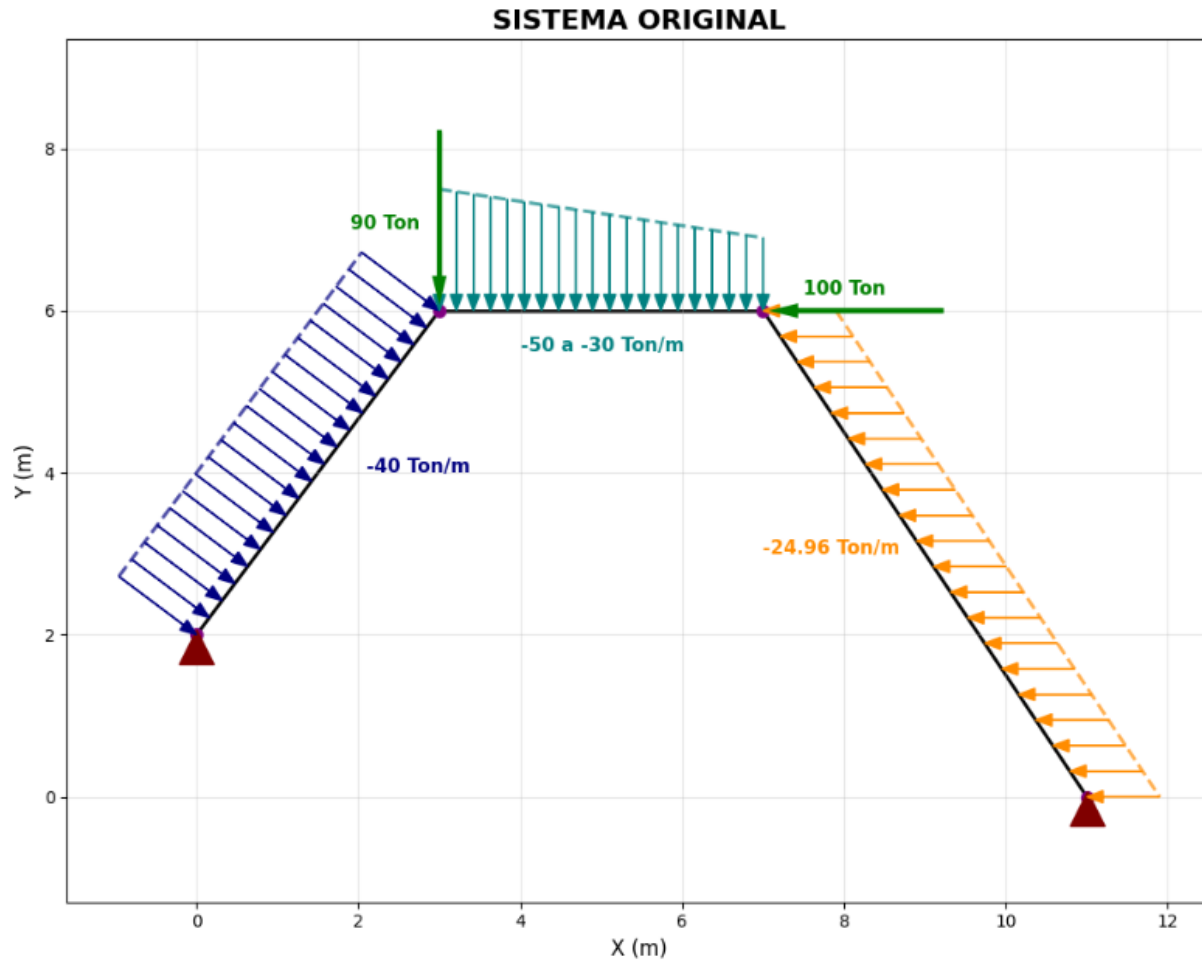
```
# --- Diagrama de Momento Flector ---
fig_m = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE MOMENTO FLECTOR (Ton-m)', fontsize=14, fontweight='bold')
ax_m = plt.gca()

# sf_type='M': diagrama de momento
opsv.section_force_diagram_2d(sf_type='M', sfac=0.005, nep=20, ax=ax_m)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```



MODELO EN OPENSEES DEL EJERCICIO 2 – PÓRTICOS



===== RESULTADOS =====

=== REACCIONES EN APOYOS ===

Nodo 1: $R_x=105.30$ Ton, $R_y=285.10$ Ton, $M_z=-0.00$ Ton-m

Nodo 4: $R_x=14.70$ Ton, $R_y=84.90$ Ton, $M_z=-0.00$ Ton-m

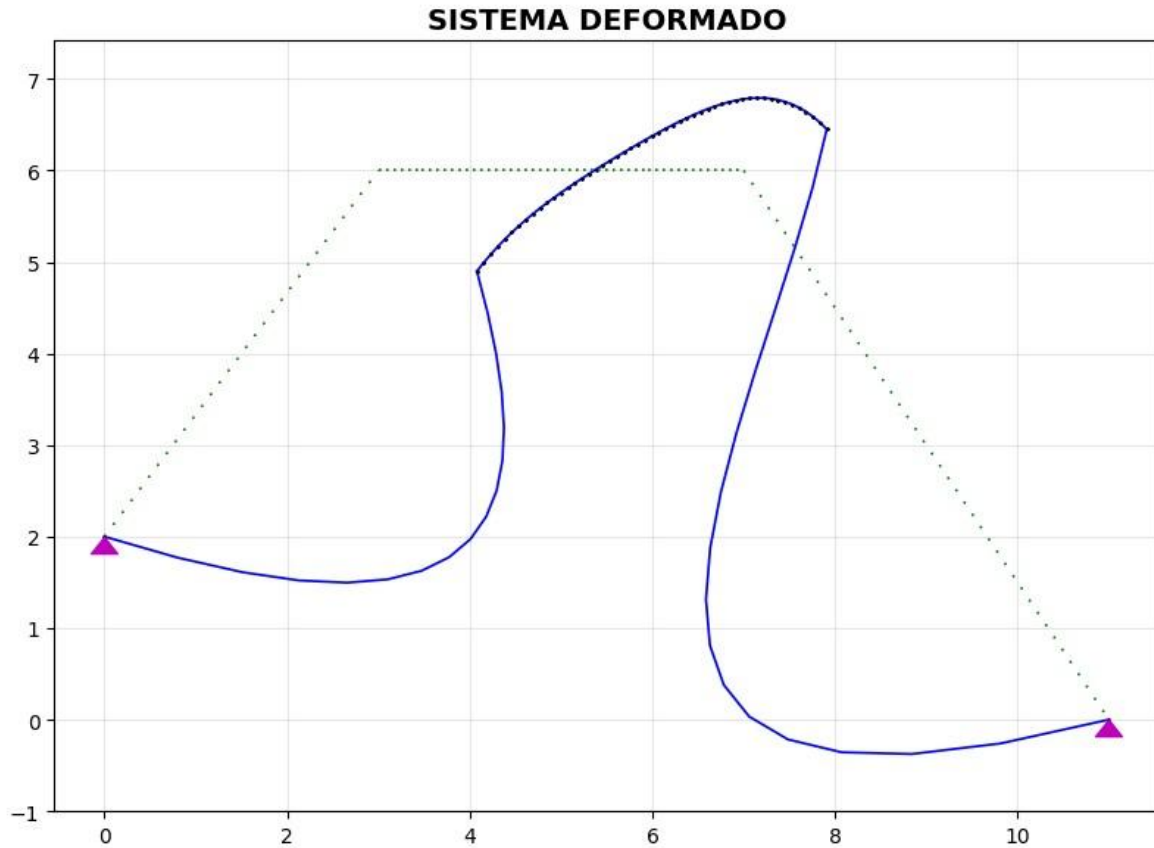
=== DESPLAZAMIENTOS NODALES ===

Nodo 1: $U_x=0.000e+00$ m, $U_y=0.000e+00$ m, $\theta_z=-7.006e-02$ rad

Nodo 2: $U_x=2.797e-02$ m, $U_y=-2.851e-02$ m, $\theta_z=3.201e-02$ rad

Nodo 3: $U_x=2.358e-02$ m, $U_y=1.169e-02$ m, $\theta_z=-2.585e-02$ rad

Nodo 4: $U_x=0.000e+00$ m, $U_y=0.000e+00$ m, $\theta_z=7.324e-02$ rad



=== FUERZAS INTERNAS EN ELEMENTOS ===

Elemento 1:

Nodo 1: $N = 291.256$ Ton, $V = 86.820$ Ton, $M = -0.000$ Ton-m

Nodo 2: $N = -291.256$ Ton, $V = 113.180$ Ton, $M = -65.902$ Ton-m

Elemento 2:

Nodo 2: $N = 265.298$ Ton, $V = 75.096$ Ton, $M = 65.902$ Ton-m

Nodo 3: $N = -265.298$ Ton, $V = 84.904$ Ton, $M = -112.172$ Ton-m

Elemento 3:

Nodo 3: $N = 162.335$ Ton, $V = 90.440$ Ton, $M = 112.172$ Ton-m

Nodo 4: $N = -62.489$ Ton, $V = 59.329$ Ton, $M = -0.000$ Ton-m

DIAGRAMA DE FUERZA AXIAL (Ton)

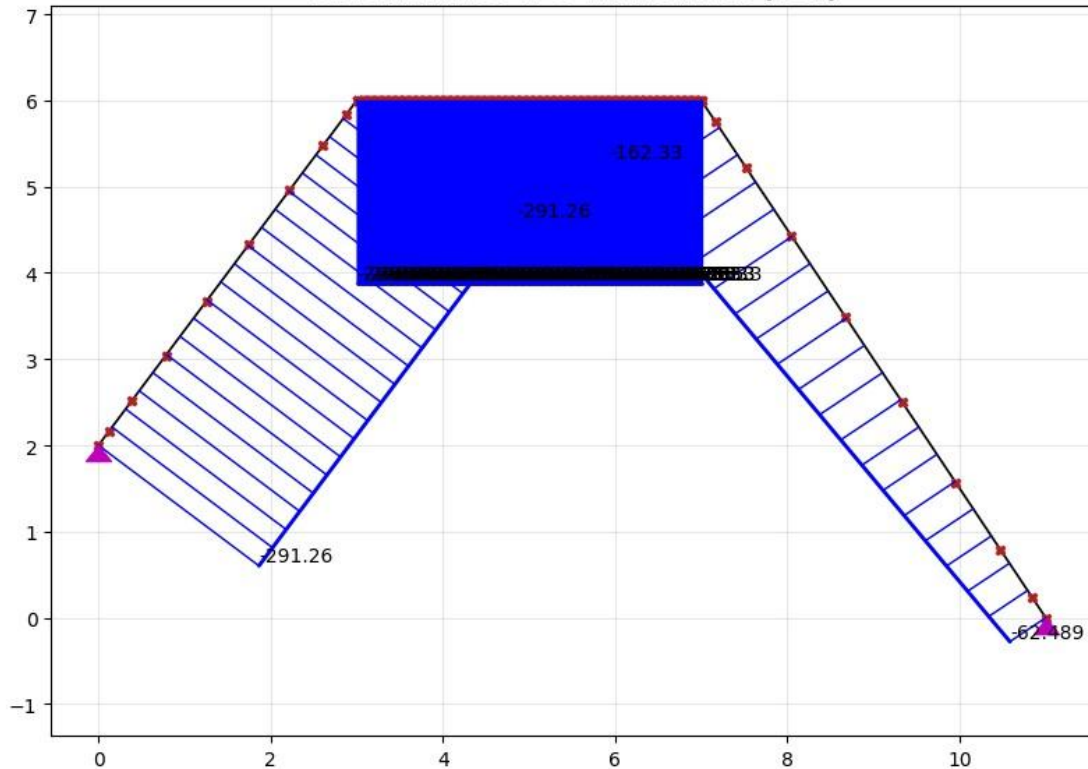
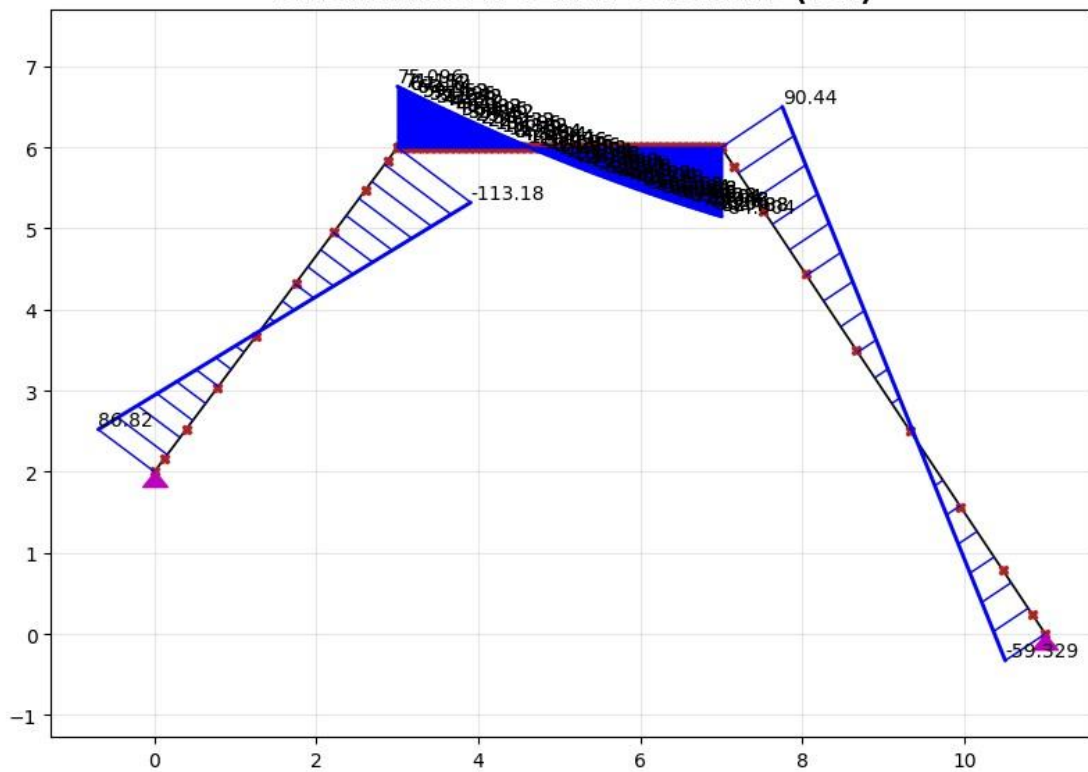
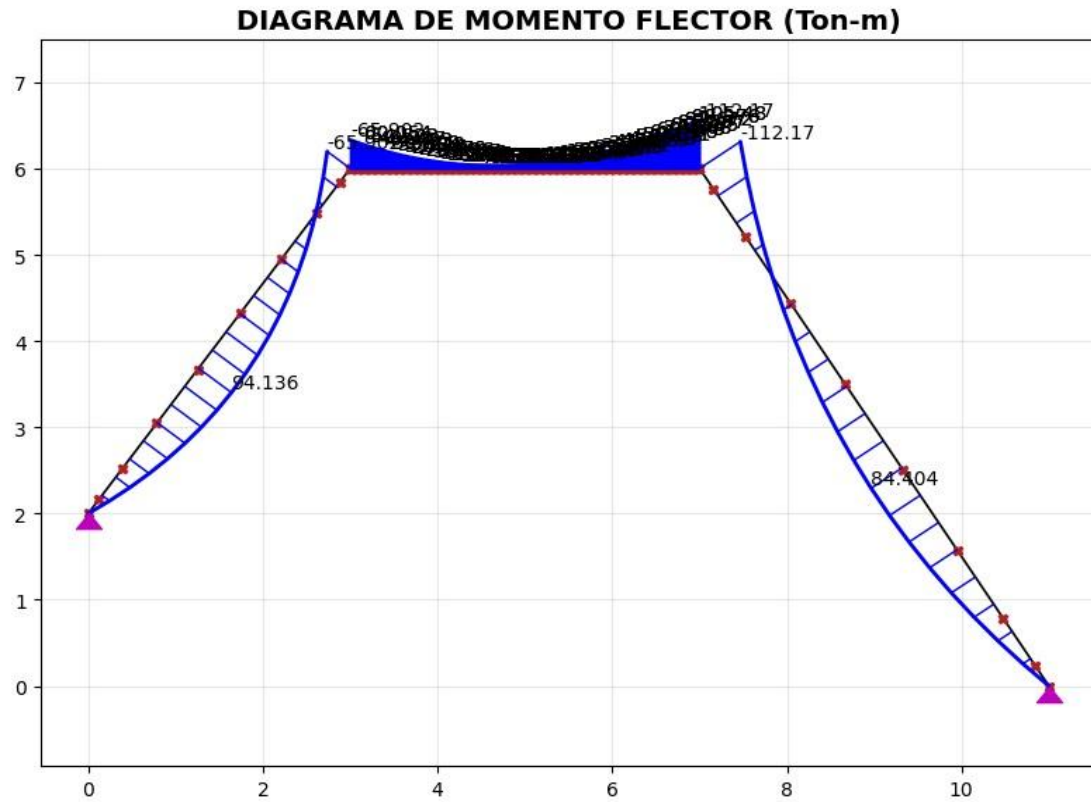


DIAGRAMA DE FUERZA CORTANTE (Ton)





MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON

PÓRTICOS: EJERCICIO 3

```
import numpy as np
import math
import matplotlib.pyplot as plt

print("MÉTODO MATRICIAL DE RIGIDEZ PARA PÓRTICOS - EJERCICIO 3")

# =====
# DATOS DE ENTRADA
# =====
print("\n=== DATOS DE ENTRADA ===")
E = 2e8 # Módulo de elasticidad (kN/m²)
A = np.array([0.0248, 0.0248, 0.0143, 0.0006, 0.0006]) # Áreas transversales (m²)
Ld = math.sqrt((4.5**2) + (9**2)) # Longitud del elemento diagonal
L = np.array([4.5, 4.5, 9, Ld/2, Ld/2]) # Longitudes (m)
I = np.array([6.360e-4, 6.360e-4, 5.492e-4, 1.8e-7, 1.8e-7]) # Inercia (m⁴)
a = np.array([90, 90, 0, math.atan(4.5/9)*(180/math.pi),
              math.atan(4.5/9)*(180/math.pi)]) # Ángulos de inclinación (°)

m = 5 # Número de elementos
n = 5 # Número de nodos
GL = n * 3 # Grados de libertad totales (15: 5 nodos × 3 GL)

# Impresión de datos de entrada
print(f"Módulo de elasticidad: {E:.1e} kN/m²")
print(f"Áreas: {np.round(A, 3)} m²")
print(f"Longitudes: {np.round(L, 3)} m")
print(f"Inercias: {np.round(I, 7)} m⁴")
print(f"Ángulos: {np.round(a, 2)}°")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Grados de libertad: {GL}")

# =====
# COORDENADAS DE LOS NODOS
# =====
# Matriz de coordenadas nodales en el plano X-Y (unidades: metros)
coordenadas_nodos = np.array([
    [0, 0], # Nodo 1
    [0, 4.5], # Nodo 2
    [9, 4.5], # Nodo 3
    [9, 0], # Nodo 4
    [4.5, 2.25]]) # Nodo 5
```



```
# =====
# DEFINICIÓN DE ELEMENTOS Y TIPOS DE CONEXIÓN
# =====
# Formato: [nodo_inicial, nodo_final, tipo_conexion_inicial, tipo_conexion_final]
# SR = Sin rótula - transmite momento flector (conexión rígida)
# R = Rótula - NO transmite momento flector (conexión articulada)
# Las rótulas afectan la matriz de rigidez local del elemento

Elementos = [
    [1, 2, 'SR', 'SR'], # Elemento 1
    [4, 3, 'SR', 'SR'], # Elemento 2
    [2, 3, 'R', 'R'],   # Elemento 3
    [1, 5, 'SR', 'R'],  # Elemento 4
    [5, 3, 'R', 'SR']] # Elemento 5

# =====
# GRADOS DE LIBERTAD POR ELEMENTO
# =====
# Asignación de GL globales para cada elemento
#
#      E1 E2 E3 E4 E5
Nx = np.array([1, 10, 4, 1, 13]) # GL desplazamiento X - nodo inicial
Ny = np.array([2, 11, 5, 2, 14]) # GL desplazamiento Y - nodo inicial
Nz = np.array([3, 12, 6, 3, 15]) # GL rotación Z - nodo inicial
Fx = np.array([4, 7, 7, 13, 7])  # GL desplazamiento X - nodo final
Fy = np.array([5, 8, 8, 14, 8])  # GL desplazamiento Y - nodo final
Fz = np.array([6, 9, 9, 15, 9])  # GL rotación Z - nodo final

# =====
# DEFINICIÓN DE CARGAS POR ELEMENTO
# =====
# Formato: [tipo, dirección, color, magnitud_inicial, magnitud_final]
Cargas = [[], []],
    ['Uniforme', 'Local_y', 'orchid', -10, -10], # Elemento 3: -10 kN/m uniforme
    [], []]

# =====
# PREPROCESAMIENTO GEOMÉTRICO DE TODOS LOS ELEMENTOS
# =====
# Calcula propiedades geométricas fundamentales para cada elemento
geom = []
for e in range(m):
    ni, nf = Elementos[e][0], Elementos[e][1]
    xi, yi = coordenadas_nodos[ni-1]
    xf, yf = coordenadas_nodos[nf-1]
    theta = math.radians(a[e])

    # Vector director unitario (apunta del nodo inicial al final)
    ex = np.cos(theta)
    ey = np.sin(theta)
```

```
# Vector normal unitario (perpendicular, rotado 90° antihorario)
nx = -ey
ny = ex

Le = L[e]
geom.append([ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny])

# =====
# GRÁFICA SISTEMA ORIGINAL
# =====

plt.figure(figsize=(7, 5.6))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')
# Dibujar elementos (barras) for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=3, zorder=1)

# Dibujar nodos con simbología especial
# Cuadrado blanco = nodo estructural
# Círculo negro sobre la barra = indicador de rótula
for i, (x, y) in enumerate(coordenadas_nodos):
    nodo_id = i + 1

    # Cuadrado blanco con borde negro para todos los nodos
    plt.plot(x, y, marker='s', markersize=14, markerfacecolor='white',
             markeredgewidth=2, color='black', zorder=3)

    # Buscar elementos conectados a este nodo que tengan rótula
    for elem in Elementos:
        ni, nj, tipo_i, tipo_j = elem

        # Determinar si el nodo tiene rótula en este elemento
        if ni == nodo_id and tipo_i == 'R':
            x2, y2 = coordenadas_nodos[nj-1]

        elif nj == nodo_id and tipo_j == 'R':
            x2, y2 = coordenadas_nodos[ni-1]

        else:
            continue

    # Vector dirección del elemento para posicionar el círculo
    dx = x2 - x
    dy = y2 - y
    L_elem = (dx**2 + dy**2)**0.5

    # Vector unitario a lo largo del elemento
    ex = dx / L_elem
    ey = dy / L_elem
```

```

# Pequeño desplazamiento sobre la barra para dibujar el círculo
offset = 0.2
xc = x + offset * ex
yc = y + offset * ey

# Círculo negro = indicador de rótula
plt.plot(xc, yc, marker='o', markersize=9, markerfacecolor='black',
markeredgecolor='black', zorder=4)
# Dibujar apoyos empotrados (cuadrados marrones)
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=2,
label='Empotramiento') plt.plot(9, -0.15, 's', color='maroon', markersize=20,
zorder=2)

# --- Dibujo de carga distribuida ---
escala_visual = 0.1 # Factor de escala para visualización de flechas

# Carga distribuida en viga superior (elemento 3)
xi, yi = coordenadas_nodos[1] # Nodo 2 (índice 1)
xf, yf = coordenadas_nodos[2] # Nodo 3 (índice 2)

dx = xf - xi
dy = yf - yi
Le = math.sqrt(dx**2 + dy**2)

# Puntos a lo largo de la viga para dibujar flechas
x_vals = np.linspace(xi, xf, 20)
y_vals = np.linspace(yi, yf, 20)

# Dibujar flechas de carga distribuida (hacia abajo)
for x, y in zip(x_vals, y_vals):
    plt.arrow(x, y+0.8, 0, -0.6, head_width=0.15, head_length=0.2, fc='orchid',
ec='orchid', zorder=4)

# Línea superior de la carga (contorno)
plt.plot([xi, xf], [yi+0.8, yf+0.8], '--', color='orchid', linewidth=2, zorder=3)
plt.text(4, 5.7, '-10 kN/m', color='orchid', fontsize=11, fontweight='bold',
zorder=5)

# Cargas puntuales
# Carga horizontal en nodo 2 (100 kN hacia la derecha)
plt.arrow(-1.7, 4.5, 1.5, 0, head_width=0.1, head_length=0.2, fc='gray', ec='gray',
linewidth=3, zorder=4) plt.text(-1.3, 4.7, '100 kN', color='gray', fontweight='bold',
fontsize=11, zorder=5)

# Carga vertical en nodo 2 (400 kN hacia abajo)
plt.arrow(0, 6.2, 0, -1.5, head_width=0.1, head_length=0.2, fc='gray', ec='gray',
linewidth=3, zorder=4) plt.text(0.2, 6, '-400 kN', color='gray', fontweight='bold',
fontsize=11, zorder=5)

```

```
# Carga vertical en nodo 3 (400 kN hacia abajo)
plt.arrow(9, 6.2, 0, -1.5, head_width=0.1, head_length=0.2, fc='gray', ec='gray',
linewidth=3, zorder=4) plt.text(7.5, 6, '-400 kN', color='gray', fontweight='bold',
fontsize=11, zorder=5)

# Configuración del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.tight_layout()
plt.show()

# =====
# ENSAMBLE DE LA MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA
# =====
print("\n=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA ===")
kG = np.zeros((GL, GL)) # Inicializar matriz de rigidez global (15x15)

# Listas para almacenar matrices de cada elemento (para post-procesamiento)
kL_elementos = [] # Matrices de rigidez local (6x6) en coordenadas locales
T_elementos = [] # Matrices de transformación (6x6) de global a local

for i in range(m):
    # Parámetros de transformación
    theta = math.radians(a[i])
    c = math.cos(theta)
    s = math.sin(theta)

    # Propiedades de rigidez
    AE = A[i]*E # Rigidez axial (EA)
    EI = E*I[i] # Rigidez flexional (EI)
    L2 = L[i]**2 # Longitud al cuadrado
    L3 = L[i]**3 # Longitud al cubo
    F = 1e-16 # Factor para simular conexiones articuladas

    # MATRIZ DE RIGIDEZ LOCAL SEGÚN TIPO DE CONEXIÓN
    # Las rótulas modifican la matriz de rigidez, eliminando grados de libertad
    # de rotación en los extremos donde hay articulación

    # CASO 1: Rótula en inicio, Sin rótula en fin (R-SR)
    # El momento en el inicio es cero, se condensa esa fila/columna
    if Elementos[i][2] == 'R' and Elementos[i][3] == 'SR':
        kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
              [0, (3*EI)/L3, F, 0, -(3*EI)/L3, (3*EI)/L2],
              [0, F, F, 0, F, F],
              [-AE/L[i], 0, F, AE/L[i], 0, 0],
              [0, -(3*EI)/L3, F, 0, (3*EI)/L3, -(3*EI)/L2],
              [0, (3*EI)/L2, F, 0, -(3*EI)/L2, (3*EI)/L[i]]]
```

```
# CASO 2: Sin rótula en inicio, rótula en fin (SR-R)
# El momento en el final es cero, se condensa esa fila/columna
elif Elementos[i][2] == 'SR' and Elementos[i][3] == 'R':
    kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
          [0, (3*EI)/L3, (3*EI)/L2, 0, -(3*EI)/L3, F],
          [0, (3*EI)/L2, (3*EI)/L[i], 0, -(3*EI)/L2, F],
          [-AE/L[i], 0, 0, AE/L[i], 0, 0],
          [0, -(3*EI)/L3, -(3*EI)/L2, 0, (3*EI)/L3, F],
          [0, F, F, 0, F, F]]

# CASO 3: Rótulas al inicio y al fin (R-R)
# Ambos momentos son cero, solo transmite carga axial (como una armadura)
elif Elementos[i][2] == 'R' and Elementos[i][3] == 'R':
    kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
          [0, F, F, 0, F, F],
          [0, F, F, 0, F, F],
          [-AE/L[i], 0, 0, AE/L[i], 0, 0],
          [0, F, F, 0, F, F],
          [0, F, F, 0, F, F]]

# CASO 4: Sin rótulas (SR-SR)
# Elemento de pórtico completo con rigidez a flexión en ambos extremos
elif Elementos[i][2] == 'SR' and Elementos[i][3] == 'SR':
    kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
          [0, (12*EI)/L3, (6*EI)/L2, 0, -(12*EI)/L3, (6*EI)/L2],
          [0, (6*EI)/L2, (4*EI)/L[i], 0, -(6*EI)/L2, (2*EI)/L[i]],
          [-AE/L[i], 0, 0, AE/L[i], 0, 0],
          [0, -(12*EI)/L3, -(6*EI)/L2, 0, (12*EI)/L3, -(6*EI)/L2],
          [0, (6*EI)/L2, (2*EI)/L[i], 0, -(6*EI)/L2, (4*EI)/L[i]]]
    kL_elementos.append(kL)

# Matriz de transformación de coordenadas
T = [[c, s, 0, 0, 0, 0],
      [-s, c, 0, 0, 0, 0],
      [0, 0, 1, 0, 0, 0],
      [0, 0, 0, c, s, 0],
      [0, 0, 0, -s, c, 0],
      [0, 0, 0, 0, 0, 1]]
T_elementos.append(T)

# Transformar matriz LOCAL a GLOBAL:  $K_{global} = T^T \cdot K_{local} \cdot T$ 
T_T = np.transpose(T) # Transformación inversa (local a global)
kg_e = np.matmul(np.matmul(T_T, kL), T) # Matriz de rigidez del elemento en
coordenadas globales

# Ensamblar en matriz global - Ajuste por indexación Python (0-based)
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]
```

```
# Sumar contribución del elemento a la matriz global
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj]

#print(f"ELEMENTO {i+1}")
#print(f"Longitud: {L[i]} m, Ángulo: {a[i]:.3f}°")
#print(f"Matriz global del elemento: \n {np.round(kg_e, 0)}")
#print(f"Matriz global del sistema después del elemento {i+1}: \n {np.round(kG, 0)}")
print(f"Matriz global del sistema: \n {np.round(kG, 0)}")

# =====
# VECTOR DE FUERZAS DE EMPOTRAMIENTO PERFECTO (FEM)
# =====
print("\n=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===")
FEM_G = np.zeros(GL)
# FEM precalculados para cada elemento en coordenadas Locales:
# Para el elemento 3 (viga con rótulas en ambos extremos) con carga uniforme:
# - Reacciones verticales:  $wL/2 = 10 \cdot 9/2 = 45$  kN en cada extremo
# - Momentos: 0 (por las rótulas)
FEM_L_elementos = [
    [0, 0, 0, 0, 0, 0], # Elem 1: sin carga
    [0, 0, 0, 0, 0, 0], # Elem 2: sin carga
    [0, 45, 0, 0, 45, 0], # Elem 3: viga - 45 kN en cada extremo
    [0, 0, 0, 0, 0, 0], # Elem 4: sin carga
    [0, 0, 0, 0, 0, 0]] # Elem 5: sin carga

# Transformar y ensamblar FEM global
for i in range(m):
    # Transformar FEM Local a global:  $FEM_{global} = T^T \cdot FEM_{local}$ 
    FEM_Ge = np.matmul(np.transpose(T_elementos[i]), FEM_L_elementos[i])

    # Ensamblar en vector global
    GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]

    for jj, gl_j in enumerate(GL_elem):
        FEM_G[gl_j] += FEM_Ge[jj] print(np.round(FEM_G, 2))

# =====
# CONDICIONES DE CONTORNO (RESTRICCIONES)
# =====
print("\n=== RESTRICCIONES ===")
# Vector de restricciones: 1 = restringido (desplazamiento conocido = 0) #
# 0 = Libre (desplazamiento desconocido)
# Nodos 1 y 4: EMPOTRADOS (dx, dy, dz = 1)
# Nodos 2, 3, 5: TODOS LIBRES (dx, dy, dz = 0)
restricciones = np.array([1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0])
```

```
print("Estado de cada grado de libertad:")
for i in range(n):
    gl_base = i * 3
    print(f"Nodo {i+1}: "
          f"Dx={'Restringido' if restricciones[gl_base] else 'Libre'}, "
          f"Dy={'Restringido' if restricciones[gl_base+1] else 'Libre'}, "
          f"θz={'Restringido' if restricciones[gl_base+2] else 'Libre'}")

# =====
# VECTOR DE FUERZAS EXTERNAS
# =====
print("\n=== VECTOR DE FUERZAS EXTERNAS ===")
# Fuerzas aplicadas directamente en nodos (cargas puntuales)
#      GL: 1  2  3  4    5  6  7    8  9  10 11 12 13 14 15
F = np.array([0, 0, 0, 100, -400, 0, 0, -400, 0, 0, 0, 0, 0, 0, 0])
print(f"Fuerzas externas: {F} kN")

# =====
# REDUCCIÓN DEL SISTEMA
# =====
print("\n=== REDUCCIÓN DEL SISTEMA ===")
# Identificar grados de libertad activos (no restringidos)
GL_activos = np.where(restricciones == 0)[0]
n_GL_activos = len(GL_activos) print(f"Grados de libertad activos: {n_GL_activos}")
print(f"Índices: {GL_activos+1}") # +1 para mostrar numeración 1-based

# Extraer submatrices para GL activos (partición de la matriz global)
kG_reducida = kG[np.ix_(GL_activos, GL_activos)] # Matriz de rigidez reducida
F_reducido = F[GL_activos] # Vector de fuerzas reducido
FEM_reducido = FEM_G[GL_activos] # Vector FEM reducido
print(f"\nMatriz global reducida: \n {np.round(kG_reducida, 0)}")
print(f"Vector de fuerzas reducido: {F_reducido} kN")
print(f"Vector FEM reducido: {np.round(FEM_reducido,2)} kN")

# =====
# SOLUCIÓN DEL SISTEMA
# =====
print("\n\n=== SOLUCIÓN DEL SISTEMA ===")
# Verificar si la matriz es singular o mal condicionada
# Si es singular, usar pseudoinversa (pinv) en lugar de inversa

if np.linalg.matrix_rank(kG_reducida) < len(kG_reducida):
    kG_r_inv = np.linalg.pinv(kG_reducida)
    print("Matriz pseudoinversa")
else:
    kG_r_inv = np.linalg.inv(kG_reducida)
    print("Matriz inversa")
```

```
# Calcular desplazamientos desconocidos:  $U = K^{-1} \cdot (F - FEM)$ 
U_desconocidos = np.matmul(kG_r_inv, F_reducido - FEM_reducido)

# Reconstruir vector completo de desplazamientos (incluyendo ceros en GL restringidos)
U_totales = np.zeros(GL)
U_totales[GL_activos] = U_desconocidos

# Calcular reacciones en apoyos:  $R = K \cdot U + FEM - F$ 
Reacciones = np.matmul(kG, U_totales) + FEM_G

print("\n=== REACCIONES ===")
# Mostrar reacciones solo en nodos con restricciones
for i in range(n):
    if np.any(restricciones[i*3:i*3+3] == 1):
        Rx = Reacciones[i*3]
        Ry = Reacciones[i*3+1]
        Mz = Reacciones[i*3+2]
        print(f"Nodo {i+1}: Rx = {Rx:.2f} kN, Ry = {Ry:.2f} kN, Mz = {Mz:.2f} kN-m")

print("\n=== DESPLAZAMIENTOS ===")
# Desplazamientos nodales (notación científica para valores pequeños)
for i in range(n):
    Ux = U_totales[i*3]
    Uy = U_totales[i*3+1]
    Tz = U_totales[i*3+2]
    print(f"Nodo {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m,  $\theta_z = {Tz:.3e}$  rad")

# =====
# GRÁFICA SISTEMA DEFORMADO
# =====
FS = 100 # Factor de escala único para visualización de desplazamientos
plt.figure(figsize=(6, 4))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar sistema original (líneas punteadas grises) como referencia
for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '--', color='grey',
             linewidth=2, alpha=0.5, label='Original' if i == 0 else "")

# Dibujar sistema deformado (líneas verdes) con desplazamientos amplificados
for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]

    # Desplazamientos de los nodos (del vector solución)
    desp_i = np.array([U_totales[(ni-1)*3], U_totales[(ni-1)*3+1],
                      U_totales[(ni-1)*3+2]))
```



```

desp_f = np.array([U_totales[(nf-1)*3], U_totales[(nf-1)*3+1],
                  U_totales[(nf-1)*3+2]])

# Coordenadas deformadas (original + desplazamiento × factor de escala)
xi_def = xi + desp_i[0] * FS
yi_def = yi + desp_i[1] * FS
xf_def = xf + desp_f[0] * FS
yf_def = yf + desp_f[1] * FS

# Dibujar elemento deformado
plt.plot([xi_def, xf_def], [yi_def, yf_def], '-',
         color='g', linewidth=2, label='Deformada' if i == 0 else "")

# Dibujar apoyos empotrados en posición original (referencia)
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=2)
plt.plot(9, -0.15, 's', color='maroon', markersize=20, zorder=2)

# Configuración del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.5)
plt.axis('equal')
plt.legend()
plt.tight_layout()
plt.show()

print("\n=== FUERZAS INTERNAS ===")
# Cálculo de fuerzas internas en los extremos de cada elemento
F_int_elementos = [] # Lista para almacenar las fuerzas internas

for i in range(m):
    # Extraer desplazamientos globales del elemento (del vector U_totales)
    Ue = np.array([
        U_totales[Nx[i]-1], # Dx nodo inicial
        U_totales[Ny[i]-1], # Dy nodo inicial
        U_totales[Nz[i]-1], # Dz nodo inicial
        U_totales[Fx[i]-1], # Dx nodo final
        U_totales[Fy[i]-1], # Dy nodo final
        U_totales[Fz[i]-1]]) # Dz nodo final

    # Transformar desplazamientos a coordenadas locales: U_local = T · U_global
    Ue_L = np.matmul(T_elementos[i], Ue)

    # Calcular fuerzas internas en coordenadas locales: F=K_local·U_local+ FEM_local
    Fe_L = np.matmul(kL_elementos[i], Ue_L) + FEM_L_elementos[i]
    F_int_elementos.append(Fe_L)

# Presentar resultados
print(f"Elemento {i+1}:")

```

```

print(f"  Nodo {Elementos[i][0]}: N = {Fe_L[0]:.3f} kN, V = {Fe_L[1]:.3f} kN, M =
{Fe_L[2]:.3f} kN-m")
print(f"  Nodo {Elementos[i][1]}: N = {Fe_L[3]:.3f} kN, V = {Fe_L[4]:.3f} kN, M =
{Fe_L[5]:.3f} kN-m\n")

# =====
# DIAGRAMA AXIAL
# =====

plt.figure(figsize=(8,6))
plt.title("DIAGRAMA AXIAL", fontsize=14, fontweight="bold")

escala_axial = 0.002  # Factor de escala para visualización

for i in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[i]

    # Extraer fuerzas axiales en extremos (del cálculo de fuerzas internas)
    Ni = F_int_elementos[i][0]
    Nf = F_int_elementos[i][3]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Verificar si hay carga distribuida axial (en dirección local x)
    if len(Cargas[i]) > 0 and Cargas[i][0] == 'Uniforme' and Cargas[i][1]=='Local_x':
        w = Cargas[i][3]  # Magnitud de La carga axial distribuida (kN/m)
    else:
        w = 0  # Sin carga axial distribuida

    # Puntos de evaluación a lo largo del elemento (30 puntos para curva suave)
    x = np.linspace(0, Le, 30)  # Coordenada local a lo largo del elemento (m)

    # Cálculo de la fuerza axial variable N(x) integrando la carga distribuida axial
    #  $N(x) = \int w_{axial}(x) dx + N_i$  (para carga constante:  $N = w*x + N_i$ )
    N = w * x + Ni

    # Coordenadas para dibujar el diagrama:
    # - Se parte del punto sobre el elemento (xi + ex*x)
    # - Se desplaza perpendicularmente (dirección nx, ny) según la magnitud de N
    X_diag = xi + ex * x + escala_axial * N * nx
    Y_diag = yi + ey * x + escala_axial * N * ny

    # Línea base del elemento (para el relleno)
    X_base = xi + ex * x
    Y_base = yi + ey * x

    # Dibujar la línea del diagrama y relleno con color semitransparente
    plt.plot(X_diag, Y_diag, '-', linewidth=1.5)
    plt.fill(np.concatenate([X_base, X_diag[:, -1]]),

```

```

np.concatenate([Y_base, Y_diag[::-1]]), alpha=0.4)

# Texto en punto medio indicando magnitud y tipo de esfuerzo
xm = xi + ex * Le / 2
ym = yi + ey * Le / 2
tipo = "Nula" if abs(Nf)<1e-6 "Tracción" if Nf>0 else "Compresión" if Nf<0
      else ""
plt.text(xm, ym, f"{Nf:.2f} kN\n({tipo})", fontsize=8, ha='center',
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()

# =====
# DIAGRAMA CORTANTE
# =====
escala_cortante = 0.03 # Factor de escala para visualización
plt.figure(figsize=(8,6))
plt.title("DIAGRAMA CORTANTE", fontsize=14, fontweight="bold")
for e in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[e]

    # Extraer fuerzas cortantes en extremos
    Vi = F_int_elementos[e][1]
    Vf = F_int_elementos[e][4]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Verificar si hay carga distribuida transversal (en dirección Local y)
    if len(Cargas[e]) > 0 and Cargas[e][0] == 'Uniforme' and Cargas[e][1]=='Local_y':
        w = Cargas[e][3] # Magnitud de la carga transversal distribuida

    else:
        w = 0

    # Puntos de evaluación a lo largo del elemento
    x = np.linspace(0, Le, 30)

    # Para carga uniforme, el cortante varía linealmente: V(x) = w*x + Vi
    V = w * x + Vi

    # Coordenadas para dibujar el diagrama (desplazamiento perpendicular)
    X_diag = xi + ex * x + escala_cortante * V * nx
    Y_diag = yi + ey * x + escala_cortante * V * ny

    # Línea base del elemento
    X_base = xi + ex * x
    Y_base = yi + ey * x

```

```
# Dibujar diagrama con relleno
plt.plot(X_diag, Y_diag, '-', linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:, -1]]),
         np.concatenate([Y_base, Y_diag[:, -1]]), alpha=0.4)

# Etiquetas con valores en extremos
plt.text(X_diag[0], Y_diag[0], f"{{Vi:.2f}}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
plt.text(X_diag[-1], Y_diag[-1], f"{{-Vf:.2f}}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
# Vf negativo por convención gráfica

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()

# =====
# DIAGRAMA MOMENTO
# =====
escala_momento = 0.007 # Factor de escala para visualización
plt.figure(figsize=(8,6))
plt.title("DIAGRAMA MOMENTO", fontsize=14, fontweight="bold")
for e in range(m):
    ni, nf, xi, yi, xf, yf, Le, ex, ey, nx, ny = geom[e]
    # Extraer fuerzas internas necesarias
    Vi = F_int_elementos[e][1] # Cortante inicial (para ecuaciones)
    Mi = F_int_elementos[e][2] # Momento en nodo inicial
    Mf = F_int_elementos[e][5] # Momento en nodo final

    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Verificar si hay carga distribuida transversal
    if len(Cargas[e]) > 0 and Cargas[e][0] == 'Uniforme' and Cargas[e][1] == 'Local_y':
        w = Cargas[e][3] # Magnitud de la carga transversal
    else:
        w = 0

    # Puntos de evaluación a lo largo del elemento
    x = np.linspace(0, Le, 30) # Coordenada Local

    # Ecuación del momento flector para carga uniforme:
    # M(x) = (w/2)*x^2 + V_i*x - M_i
    M = (w / 2) * x**2 + Vi * x - Mi

    # Coordenadas para dibujar el diagrama parabólico (cuando hay carga)
    X_diag = xi + ex * x + escala_momento * M * nx
    Y_diag = yi + ey * x + escala_momento * M * ny
```

```
X_base = xi + ex * x
Y_base = yi + ey * x

# Dibujar diagrama con relleno
plt.plot(X_diag, Y_diag, '-', linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:-1]]),
         np.concatenate([Y_base, Y_diag[:-1]]), alpha=0.4)

# Calcular y marcar el punto de momento máximo (donde el cortante se anula)
if abs(w) > 1e-9: # Evitar división por cero (solo si hay carga)
    x_max = -Vi / w # Punto donde el cortante V(x) = w*x + Vi = 0

# Verificar que el punto está dentro del elemento
if 0 <= x_max <= Le:
    M_max = (w / 2) * x_max**2 + Vi * x_max - Mi

# Coordenadas del punto de momento máximo en el diagrama
xm = xi + ex * x_max + escala_momento * M_max * nx
ym = yi + ey * x_max + escala_momento * M_max * ny

# Marcar el punto y etiquetar
plt.scatter(xm, ym, s=30, zorder=5)
plt.text(xm, ym + 0.3, f"{M_max:.2f}", fontsize=9,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

# Etiquetas con valores de momento en extremos
# Nota: El signo negativo en Mi es por convención gráfica
plt.text(X_diag[0], Y_diag[0], f"{-Mi:.2f}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
plt.text(X_diag[-1], Y_diag[-1], f"{Mf:.2f}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()
```

MÉTODO MATRICIAL DE RIGIDEZ PARA PÓRTICOS - EJERCICIO 3

=== DATOS DE ENTRADA ===

Módulo de elasticidad: $2.0 \times 10^8 \text{ kN/m}^2$

Áreas: $[0.025 \ 0.025 \ 0.014 \ 0.001 \ 0.001] \text{ m}^2$

Longitudes: $[4.5 \ 4.5 \ 9. \ 5.031 \ 5.031] \text{ m}$

Inercias: $[6.360 \times 10^{-4} \ 6.360 \times 10^{-4} \ 5.492 \times 10^{-4} \ 2.000 \times 10^{-7} \ 2.000 \times 10^{-7}] \text{ m}^4$

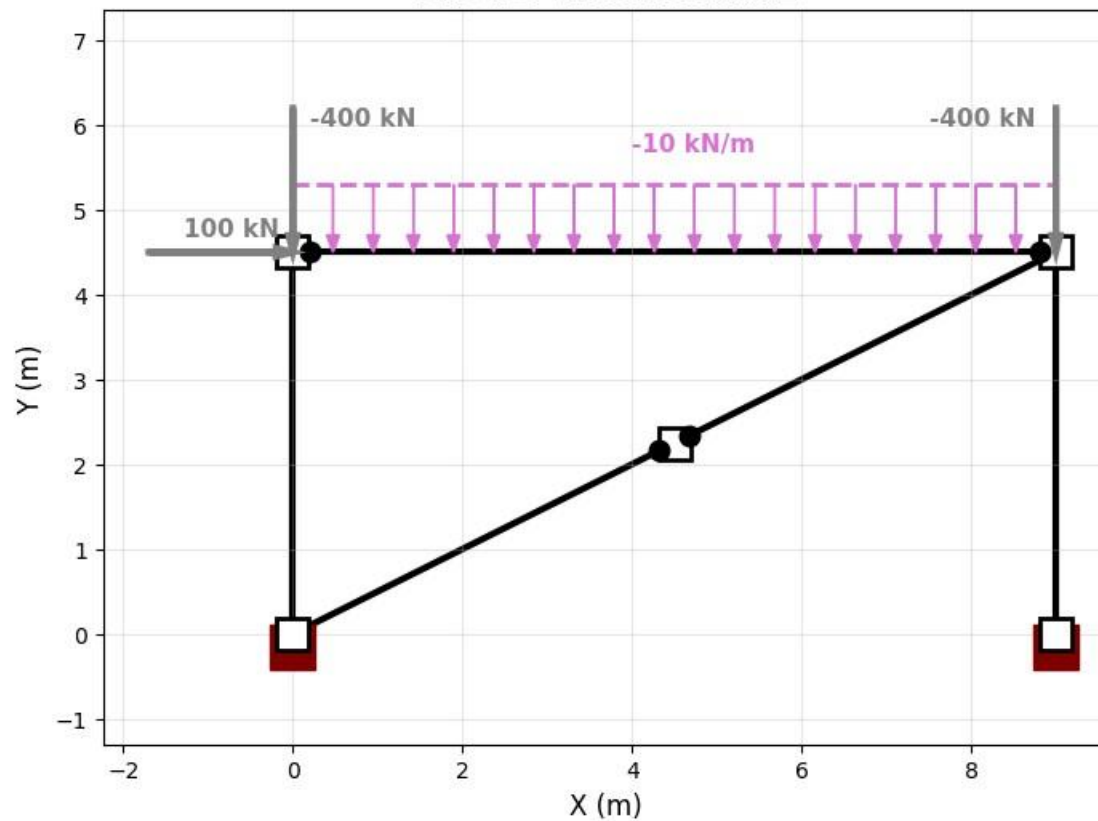
Ángulos: $[90. \ 90. \ 0. \ 26.57 \ 26.57]^\circ$

Número de elementos: 5

Número de nodos: 5

Grados de libertad: 15

SISTEMA ORIGINAL



=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL DEL SISTEMA ===

Matriz global del sistema:

```
[ [ 3.583200e+04  9.540000e+03 -3.769100e+04 -1.675100e+04 -0.000000e+00  -
3.768900e+04  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00 -1.908100e+04 -9.540000e+03  0.000000e+00]
[ 9.540000e+03  1.106993e+06  4.000000e+00 -0.000000e+00 -1.102222e+06
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00 -9.540000e+03 -4.771000e+03  0.000000e+00]
[ -3.769100e+04  4.000000e+00  1.130880e+05  3.768900e+04 -0.000000e+00
  5.653300e+04  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  2.000000e+00 -4.000000e+00  0.000000e+00]
[ -1.675100e+04 -0.000000e+00  3.768900e+04  3.345280e+05  0.000000e+00
  3.768900e+04 -3.177780e+05  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]
[ -0.000000e+00 -1.102222e+06 -0.000000e+00  0.000000e+00  1.102222e+06  -
0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]
[ -3.768900e+04  0.000000e+00  5.653300e+04  3.768900e+04 -0.000000e+00
  1.130670e+05  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]
[  0.000000e+00  0.000000e+00  0.000000e+00 -3.177780e+05  0.000000e+00
  0.000000e+00  3.536100e+05  9.540000e+03  3.769100e+04 -1.675100e+04
 -0.000000e+00  3.768900e+04 -1.908100e+04 -9.540000e+03  0.000000e+00]
[  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  9.540000e+03  1.106993e+06 -4.000000e+00 -0.000000e+00
 -1.102222e+06 -0.000000e+00 -9.540000e+03 -4.771000e+03  0.000000e+00]
[  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  3.769100e+04 -4.000000e+00  1.130880e+05 -3.768900e+04
  0.000000e+00  5.653300e+04 -2.000000e+00  4.000000e+00  0.000000e+00]
[  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00 -1.675100e+04 -0.000000e+00 -3.768900e+04  1.675100e+04
  0.000000e+00 -3.768900e+04  0.000000e+00  0.000000e+00  0.000000e+00]
[  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00 -0.000000e+00 -1.102222e+06  0.000000e+00  0.000000e+00
  1.102222e+06  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]
[  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  3.768900e+04 -0.000000e+00  5.653300e+04 -3.768900e+04
  0.000000e+00  1.130670e+05  0.000000e+00  0.000000e+00  0.000000e+00]
[ -1.908100e+04 -9.540000e+03  2.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00 -1.908100e+04 -9.540000e+03 -2.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  3.816300e+04  1.908000e+04  0.000000e+00]
[ -9.540000e+03 -4.771000e+03 -4.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00 -9.540000e+03 -4.771000e+03  4.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  1.908000e+04  9.542000e+03  0.000000e+00]
[  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]]
```

=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===

[0. 0. 0. 0. 45. 0. 0. 45. 0. 0. 0. 0. 0. 0.]

=== RESTRICCIONES ===

Estado de cada grado de libertad:

Nodo 1: Dx=Restringido, Dy=Restringido, θ_z =Restringido

Nodo 2: Dx=Libre, Dy=Libre, θ_z =Libre

Nodo 3: Dx=Libre, Dy=Libre, θ_z =Libre

Nodo 4: Dx=Restringido, Dy=Restringido, θ_z =Restringido

Nodo 5: Dx=Libre, Dy=Libre, θ_z =Libre

=== VECTOR DE FUERZAS EXTERNAS ===

Fuerzas externas: [0 0 0 100 -400 0 0 -400 0 0 0 0 0 0 0] kN

=== REDUCCIÓN DEL SISTEMA ===

Grados de libertad activos: 9

Índices: [4 5 6 7 8 9 13 14 15]

Matriz global reducida:

```
[ [ 3.345280e+05  0.000000e+00  3.768900e+04 -3.177780e+05  0.000000e+00
    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]
  [ 0.000000e+00  1.102222e+06 -0.000000e+00  0.000000e+00  0.000000e+00
    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]
  [ 3.768900e+04 -0.000000e+00  1.130670e+05  0.000000e+00  0.000000e+00
    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]
  [-3.177780e+05  0.000000e+00  0.000000e+00  3.536100e+05  9.540000e+03
    3.769100e+04 -1.908100e+04 -9.540000e+03  0.000000e+00]
  [ 0.000000e+00  0.000000e+00  0.000000e+00  9.540000e+03  1.106993e+06
    4.000000e+00 -9.540000e+03 -4.771000e+03  0.000000e+00]
  [ 0.000000e+00  0.000000e+00  0.000000e+00  3.769100e+04 -4.000000e+00
    1.130880e+05 -2.000000e+00  4.000000e+00  0.000000e+00]
  [ 0.000000e+00  0.000000e+00  0.000000e+00 -1.908100e+04 -9.540000e+03
    2.000000e+00  3.816300e+04  1.908000e+04  0.000000e+00]
  [ 0.000000e+00  0.000000e+00  0.000000e+00 -9.540000e+03 -4.771000e+03
    4.000000e+00  1.908000e+04  9.542000e+03  0.000000e+00]
  [ 0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00
    0.000000e+00  0.000000e+00  0.000000e+00  0.000000e+00]]
```

Vector de fuerzas reducido: [100 -400 0 0 -400 0 0 0 0] kN

Vector FEM reducido: [0. 45. 0. 0. 45. 0. 0. 0. 0.] kN

=== SOLUCIÓN DEL SISTEMA ===

Matriz pseudoinversa

=== REACCIONES ===

Nodo 1: Rx = -76.38 kN, Ry = 419.11 kN, Mz = 110.73 kN-m

Nodo 4: Rx = -23.62 kN, Ry = 470.89 kN, Mz = 106.28 kN-m

=== DESPLAZAMIENTOS ===

Nodo 1: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $\theta_z = 0.000e+00$ rad

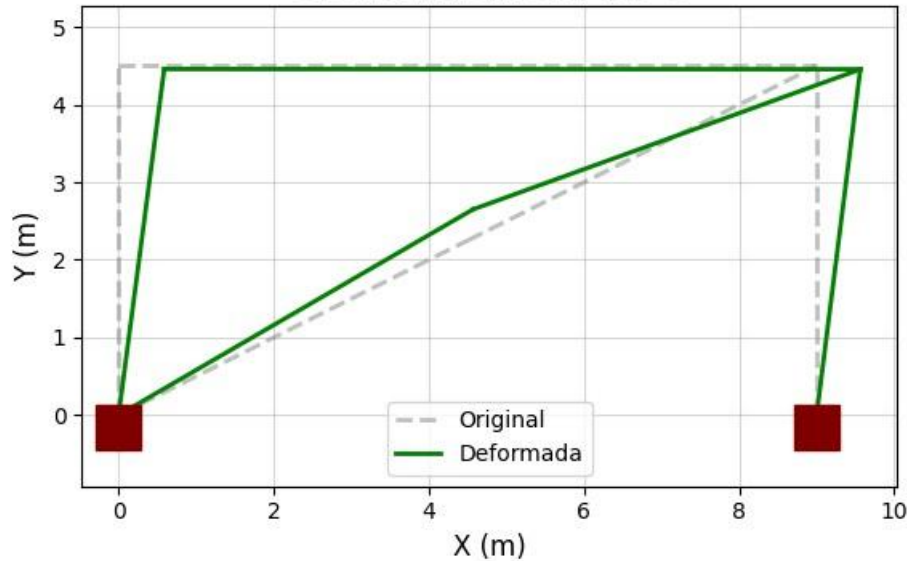
Nodo 2: $U_x = 5.877e-03$ m, $U_y = -4.037e-04$ m, $\theta_z = -1.959e-03$ rad

Nodo 3: $U_x = 5.640e-03$ m, $U_y = -4.272e-04$ m, $\theta_z = -1.880e-03$ rad

Nodo 4: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $\theta_z = 0.000e+00$ rad

Nodo 5: $U_x = 7.051e-04$ m, $U_y = 4.016e-03$ m, $\theta_z = -3.152e-19$ rad

SISTEMA DEFORMADO



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 445.000$ kN, $V = 24.611$ kN, $M = 110.749$ kN-m

Nodo 2: $N = -445.000$ kN, $V = -24.611$ kN, $M = 0.000$ kN-m

Elemento 2:

Nodo 4: $N = 470.887$ kN, $V = 23.622$ kN, $M = 106.285$ kN-m

Nodo 3: $N = -470.887$ kN, $V = -23.622$ kN, $M = 0.014$ kN-m

Elemento 3:

Nodo 2: $N = 75.389$ kN, $V = 45.000$ kN, $M = 0.000$ kN-m

Nodo 3: $N = -75.389$ kN, $V = 45.000$ kN, $M = 0.000$ kN-m

Elemento 4:

Nodo 1: $N = -57.879$ kN, $V = -0.003$ kN, $M = -0.014$ kN-m

Nodo 5: $N = 57.879$ kN, $V = 0.003$ kN, $M = 0.000$ kN-m

Elemento 5:

Nodo 5: $N = -57.879$ kN, $V = -0.003$ kN, $M = 0.000$ kN-m

Nodo 3: $N = 57.879$ kN, $V = 0.003$ kN, $M = -0.014$ kN-m

DIAGRAMA AXIAL

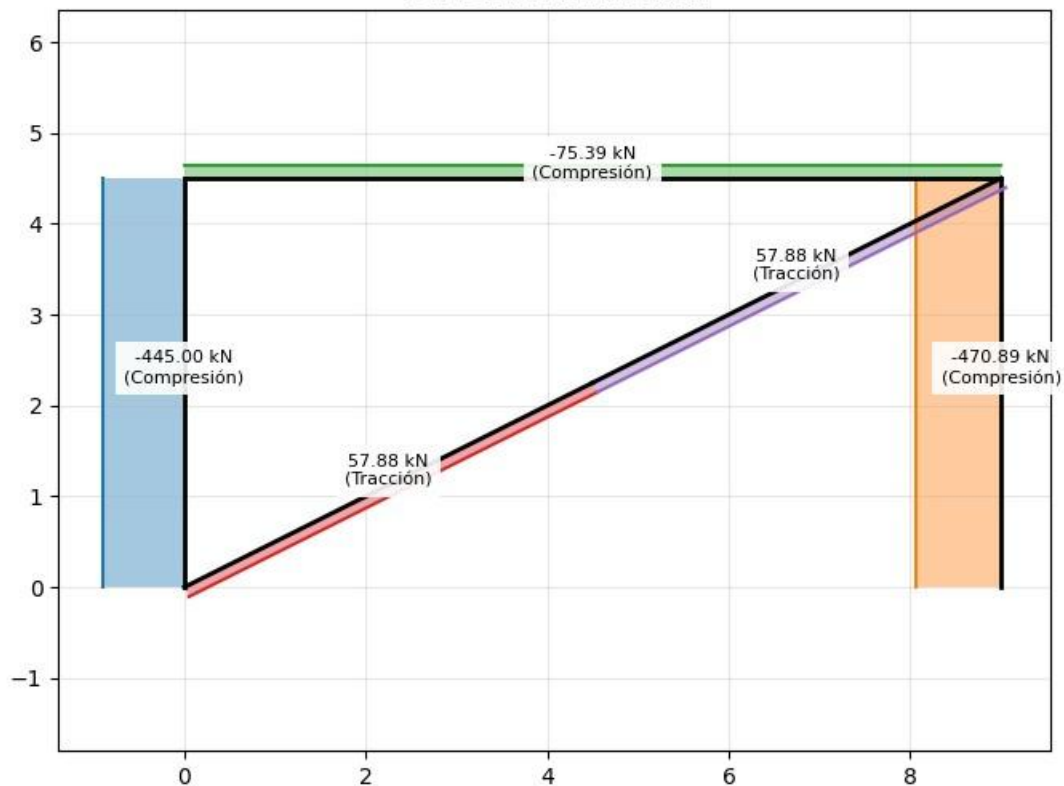


DIAGRAMA CORTANTE

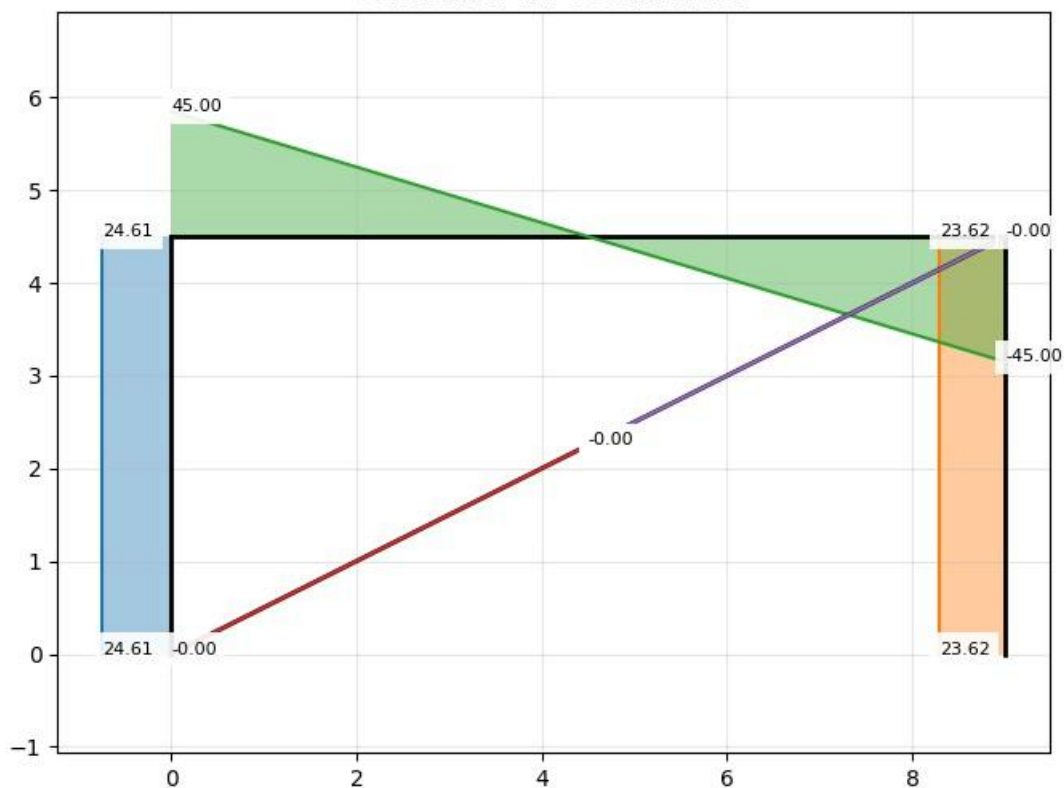
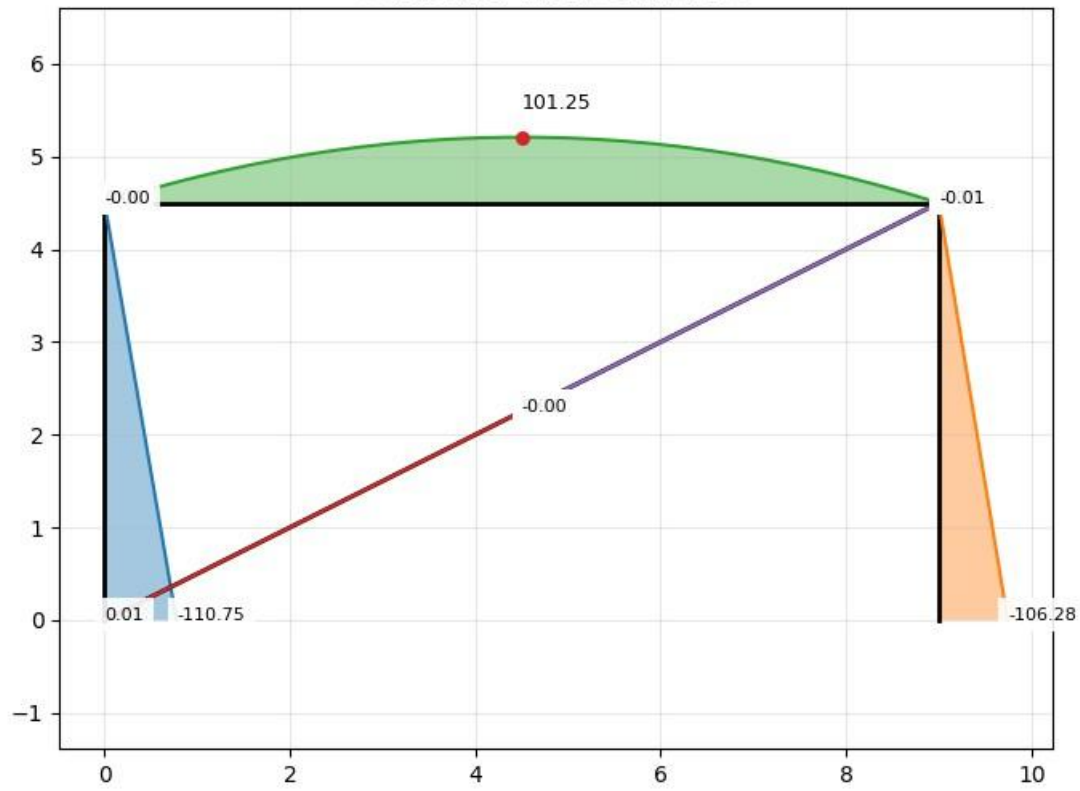


DIAGRAMA MOMENTO



MODELO OPENSEES

PÓRTICOS: EJERCICIO 3

```
# =====
# IMPORTACIÓN DE LIBRERÍAS
# =====
import openseespy.opensees as ops # Biblioteca principal de OpenSees para análisis estructural
import opsviz as opsv             # Librería para visualización de resultados y diagramas
import numpy as np                # Para operaciones numéricas y arreglos
import math                       # Para funciones matemáticas básicas (atan, sqrt)
import matplotlib.pyplot as plt   # Para visualización de resultados personalizada

print("MODELO EN OPENSEES PARA EL EJERCICIO 3 - PÓRTICOS CON RÓTULAS")

# =====
# INICIALIZACIÓN DEL MODELO
# =====
ops.wipe() # Limpiar memoria de análisis previos (reiniciar el modelo)

# Modelo 2D con 3 grados de libertad por nodo (dx, dy, θz)
# '-ndm': número de dimensiones espaciales (2D)
# '-ndf': número de grados de libertad por nodo (3:desplazamientos X, Y y rotación Z)
ops.model('basic', '-ndm', 2, '-ndf', 3)

# =====
# DATOS DE ENTRADA
# =====
E = 2e8 # Módulo de elasticidad (kN/m²)

# --- Columnas (sección transversal) ---
Ac = 0.0248 # Área de columna (m²)
Ic = 6.360e-4 # Momento de inercia de columna (m⁴)

# --- Viga (sección transversal) ---
Av = 0.0143 # Área de viga (m²)
Iv = 5.492e-4 # Momento de inercia de viga (m⁴)

# --- Diagonal (sección transversal) ---
Ad = 6e-4 # Área (m²)
Id = 1.8e-7 # Momento de inercia (m⁴)

# =====
# DEFINICIÓN DE NODOS (PRINCIPALES Y DUPLICADOS)
# =====
# Nodos principales (estructura base) - coordenadas en metros
ops.node(1, 0, 0) # Nodo 1
ops.node(2, 0, 4.5) # Nodo 2
```

```
ops.node(3, 9, 4.5)    # Nodo 3
ops.node(4, 9, 0)      # Nodo 4
ops.node(5, 4.5, 2.25) # Nodo 5

# Nodos DUPLICADOS para crear rótulas (misma coordenada que nodos principales)
# Estrategia de modelado de rótulas:
# - Nodo original: conectado a columna (conexión empotrada)
# - Nodo duplicado: conectado a viga/riostra (conexión articulada)
# - Elemento zeroLength entre ambos: libera el momento (rigidez rotacional ~0)
ops.node(20, 0, 4.5)   # Duplicado de nodo 2 (para conectar viga)
ops.node(30, 9, 4.5)   # Duplicado de nodo 3 (para conectar viga)
ops.node(50, 4.5, 2.25) # Duplicado de nodo 5 (para conectar riostra derecha)

# =====
# COMPATIBILIDAD DE DESPLAZAMIENTOS (equalDOF)
# =====
# Los nodos duplicados DEBEN tener los mismos desplazamientos (Ux, Uy) que sus nodos
# originales, pero rotación independiente (para simular rótula)
# Parámetros: (nodo_principal, nodo_duplicado, gdl1, gdl2, ...)

ops.equalDOF(2, 20, 1, 2) # Nodo 2 y 20: mismos Ux, Uy (θz Libre)
ops.equalDOF(3, 30, 1, 2) # Nodo 3 y 30: mismos Ux, Uy (θz Libre)
ops.equalDOF(5, 50, 1, 2) # Nodo 5 y 50: mismos Ux, Uy (θz Libre)

# =====
# MATERIAL PARA RÓTULAS (RIGIDEZ ROTACIONAL CERO)
# =====
# Material elástico con rigidez extremadamente baja para liberar momento 1e-16 es
# prácticamente cero pero evita singularidades numéricas en la matriz. Este material se
# usará en los elementos zeroLength solo para la rotación
ops.uniaxialMaterial('Elastic', 1, 1e-16) # Tag 1, E = 1e-16 kN·m/rad

# =====
# CONDICIONES DE APOYO
# =====
# Empotramiento: restringir dx, dy, θz (1 = restringido, 0 = Libre)
ops.fix(1, 1, 1, 1) # Nodo 1: Empotrado en base izquierda
ops.fix(4, 1, 1, 1) # Nodo 4: Empotrado en base derecha

# =====
# TRANSFORMACIÓN GEOMÉTRICA
# =====
transfTag = 1
# 'Linear': transformación lineal (asume pequeñas deformaciones y desplazamientos)
ops.geomTransf('Linear', transfTag)
```

```
# =====
# ELEMENTOS (elasticBeamColumn)
# =====
# Lista para almacenar información de elementos (para post-procesamiento y
# visualización)
Elementos = []

# --- ELEMENTO 1: Columna izquierda ---
ops.element('elasticBeamColumn', 1, 1, 2, Ac, E, Ic, transfTag)
Elementos.append({"ID": 1, "Nodo_i": 1, "Nodo_f": 2})

# --- ELEMENTO 2: Columna derecha ---
ops.element('elasticBeamColumn', 2, 4, 3, Ac, E, Ic, transfTag)
Elementos.append({"ID": 2, "Nodo_i": 4, "Nodo_f": 3})

# --- ELEMENTO 3: Viga superior ---
# Conecta nodo DUPLICADO 20 con nodo DUPLICADO 30. AL usar nodos duplicados, la viga
# tiene extremos articulados (rótulas en ambos lados)
ops.element('elasticBeamColumn', 3, 20, 30, Av, E, Iv, transfTag)
Elementos.append({"ID": 3, "Nodo_i": 20, "Nodo_f": 30})

# --- ELEMENTO 4: Riostra izquierda ---
# Conecta nodo 1 (base izquierda, empotrada) con nodo 5 (centro, articulado vía
# zeroLength)
ops.element('elasticBeamColumn', 4, 1, 5, Ad, E, Id, transfTag)
Elementos.append({"ID": 4, "Nodo_i": 1, "Nodo_f": 5})

# --- ELEMENTO 5: Riostra derecha ---
# Conecta nodo DUPLICADO 50 (centro, articulado) con nodo 3 (columna derecha,
# empotrada)
ops.element('elasticBeamColumn', 5, 50, 3, Ad, E, Id, transfTag)
Elementos.append({"ID": 5, "Nodo_i": 50, "Nodo_f": 3})

# =====
# ELEMENTOS ZEROLENGTH (RÓTULAS DE MOMENTO CERO)
# =====
# Estos elementos conectan nodos principales con sus duplicados
# Solo actúan en dirección 3 (rotación), con rigidez ~0 (material tag 1)
# Permiten que el nodo duplicado rote libremente respecto al principal
# Parámetros: (tipo, tag, nodo_i, nodo_j, '-mat', material_tag, '-dir',
# grado_de_libertad)

# Rótula en nodo 2 (conexión columna izquierda - viga)
ops.element('zeroLength', 6, 2, 20, '-mat', 1, '-dir', 3)

# Rótula en nodo 3 (conexión columna derecha - viga)
ops.element('zeroLength', 7, 3, 30, '-mat', 1, '-dir', 3)
```

```
# Rótula en nodo 5 (conexión riostras en el centro)
ops.element('zeroLength', 8, 5, 50, '-mat', 1, '-dir', 3)

# =====
# DEFINICIÓN DE CARGAS
# =====
# Configurar patrón de carga estática
ops.timeSeries('Linear', 1) # Serie temporal lineal (carga crece de 0 a 1 en un paso)
ops.pattern('Plain', 1, 1) # Patrón de carga estático asociado a serie temporal tag 1

# --- Carga distribuida en viga superior (elemento 3) ---
w1 = -10 # kN/m (negativo = hacia abajo en dirección local Y)

# '-beamUniform': carga uniforme en el elemento (carga_local_y, carga_local_x)
ops.eleLoad('-ele', 3, '-type', '-beamUniform', w1, 0.0)

# --- Cargas concentradas en nodos ---
# ops.Load(etiqueta_nodo, fuerza_x, fuerza_y, momento_z)
ops.load(2, 100, -400, 0) # Nodo 2: 100 kN (+X), 400 kN (-Y)
ops.load(3, 0, -400, 0) # Nodo 3: 400 kN (-Y)

# =====
# GRÁFICA SISTEMA ORIGINAL
# =====
plt.figure(figsize=(7, 5.6))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos estructurales (líneas negras)
for elem in Elementos:
    Ni = elem["Nodo_i"]
    Nf = elem["Nodo_f"]

    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    plt.plot([xi, xf], [yi, yf], 'k-', lw=2, zorder=1)

# Dibujar nodos PRINCIPALES (cuadrados blancos con borde negro)
nodos_principales = [1, 2, 3, 4, 5]

for nodo in nodos_principales:
    x, y = ops.nodeCoord(nodo)
    plt.plot(x, y, marker='s', markersize=12, markerfacecolor='white',
             markeredgewidth=2, markeredgewidth=2, zorder=3)
```

```
# Dibujar RÓTULAS (círculos negros en nodos principales)
# Indican visualmente que hay un elemento zeroLength conectado en ese nodo
rotulas = [(2, 20), (3, 30), (5, 50)]

for n1, n2 in rotulas:
    x, y = ops.nodeCoord(n1)
    plt.plot(x, y, marker='o', markersize=7, markerfacecolor='black',
             markeredgecolor='black', zorder=4)

# Dibujar apoyos empotrados (símbolos de empotramiento - cuadrados marrones)
plt.plot(0, -0.2, 's', color='maroon', markersize=18, zorder=2)
plt.plot(9, -0.2, 's', color='maroon', markersize=18, zorder=2)

# --- Dibujo de cargas ---
# Carga distribuida en viga (elemento 3 - viga superior)
xi, yi = ops.nodeCoord(20) # Nodo duplicado 20 (inicio de viga)
xf, yf = ops.nodeCoord(30) # Nodo duplicado 30 (fin de viga)
dx = xf - xi
L = dx # Longitud de la viga (horizontal)

# Dibujar flechas de carga distribuida (hacia abajo)
x_vals = np.linspace(xi, xf, 20)
y_vals = np.linspace(yi, yf, 20)

for x, y in zip(x_vals, y_vals):
    plt.arrow(x, y+0.8, 0, -0.6, head_width=0.15, head_length=0.2,
             fc='skyblue', ec='skyblue', zorder=4)

plt.plot([xi, xf], [yi+0.8, yf+0.8], '--', color='skyblue', linewidth=2, zorder=3)
plt.text(3.5, 5.5, '-10 kN/m', color='skyblue', fontsize=11, fontweight='bold',
        zorder=5)

# Cargas puntuales (flechas naranjas)
# Carga horizontal en nodo 2 (100 kN hacia la derecha)
plt.arrow(-2, 4.5, 1.5, 0, head_width=0.1, head_length=0.2,
        fc='orange', ec='orange', linewidth=3, zorder=4)
plt.text(-1.3, 4.7, '100 kN', color='orange', fontweight='bold', fontsize=11,
        zorder=5)

# Carga vertical en nodo 2 (400 kN hacia abajo)
plt.arrow(0, 6.5, 0, -1.5, head_width=0.1, head_length=0.2,
        fc='orange', ec='orange', linewidth=3, zorder=4)
plt.text(0.2, 6, '-400 kN', color='orange', fontweight='bold', fontsize=11, zorder=5)

# Carga vertical en nodo 3 (400 kN hacia abajo)
plt.arrow(9, 6.5, 0, -1.5, head_width=0.1, head_length=0.2,
        fc='orange', ec='orange', linewidth=3, zorder=4)
plt.text(7.5, 6, '-400 kN', color='orange', fontweight='bold', fontsize=11, zorder=5)
```



```
# Configuración del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal') # Misma escala en X e Y para proporciones reales
plt.tight_layout()
plt.show()

# =====
# CONFIGURACIÓN DEL ANÁLISIS
# =====
ops.system('BandSPD') # Almacena la matriz de rigidez en formato de banda
simétrica definida positiva.
ops.numberer('RCM') # Algoritmo Reverse Cuthill-McKee: reordenamiento de nodos para
reducir el ancho de banda de la matriz de rigidez.
ops.constraints('Plain') # Imposición directa de restricciones (apoyos y equalDOF)
ops.integrator('LoadControl', 1.0) # Control por carga: aplica 100% de la carga en 1 paso
ops.algorithm('Linear') # Algoritmo de solución lineal (suficiente para análisis elástico)
ops.analysis('Static') # Tipo de análisis: estático
ops.analyze(1) # Ejecutar análisis con 1 paso de carga

# =====
# RESULTADOS
# =====
print("\n=== REACCIONES EN APOYOS ===")
ops.reactions() # Calcular reacciones en nodos restringidos

for i in [1, 4]:
    Rx = ops.nodeReaction(i, 1) # Reacción en dirección X (gdl 1)
    Ry = ops.nodeReaction(i, 2) # Reacción en dirección Y (gdl 2)
    Mz = ops.nodeReaction(i, 3) # Momento de reacción (gdl 3)

    print(f"Node {i}: Rx = {Rx:.3f} kN, Ry = {Ry:.3f} kN, Mz = {Mz:.3f} kN-m")

print("\n=== DESPLAZAMIENTOS NODALES ===")

for i in range(1, 6):
    disp = ops.nodeDisp(i) # Devuelve [dx, dy, dz] para el nodo i

    print(f"Node {i}: Dx = {disp[0]:.3e} m, Dy = {disp[1]:.3e} m, Dz = {disp[2]:.3e}
rad")

# =====
# GRÁFICA SISTEMA DEFORMADO
# =====
# Crear figura para la deformada
fig, ax = plt.subplots(figsize=(8, 6))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')
```

Dibujar la deformada usando opsv. El parámetro 'ax' le dice a plot_defo() que use nuestros ejes existentes. Esto dibuja la estructura deformada (factor de escala automático)
opsv.plot_defo(ax=ax)

```
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

```
# =====
# FUERZAS INTERNAS EN ELEMENTOS
# =====
print("\n=== FUERZAS INTERNAS EN ELEMENTOS ===")
F_int_elementos = [] # Lista para almacenar fuerzas de todos los elementos
```

```
for element_data in Elementos:
    eleTag = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_f"]

    # Obtener fuerzas internas en coordenadas locales del elemento
    # 'LocalForces' devuelve: [N_i, V_i, M_i, N_f, V_f, M_f]
    # Orden: axial i, cortante i, momento i, axial f, cortante f, momento f
    F_int = opsv.eleResponse(eleTag, 'localForces')
    F_int_elementos.append(F_int)

    # Mostrar resultados por elemento
    print(f"Elemento {eleTag}")
    print(f"  Nodo {Ni}: N = {F_int[0]:.3f} kN, V = {F_int[1]:.3f} kN, M = {F_int[2]:.3f} kN-m")
    print(f"  Nodo {Nf}: N = {F_int[3]:.3f} kN, V = {F_int[4]:.3f} kN, M = {F_int[5]:.3f} kN-m\n")
```

```
# =====
# DIAGRAMAS DE ESFUERZOS
# =====
```

```
# --- Diagrama de Fuerza Axial ---
fig_n = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA AXIAL (kN)', fontsize=14, fontweight='bold')
ax_n = plt.gca()
```

```
# sf_type='N': diagrama de fuerza axial
# sfac: factor de escala para visualización (ajustar según magnitudes)
# nep: número de puntos de evaluación por elemento
opsv.section_force_diagram_2d(sf_type='N', sfac=0.008, nep=20, ax=ax_n)
```

```
plt.grid(True, alpha=0.3)
```

```
plt.axis('equal')
plt.tight_layout()
plt.show()

# --- Diagrama de Fuerza Cortante ---
fig_v = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA CORTANTE (kN)', fontsize=14, fontweight='bold')
ax_v = plt.gca()

# sf_type='V': diagrama de fuerza cortante
opsv.section_force_diagram_2d(sf_type='V', sfac=0.01, nep=20, ax=ax_v)

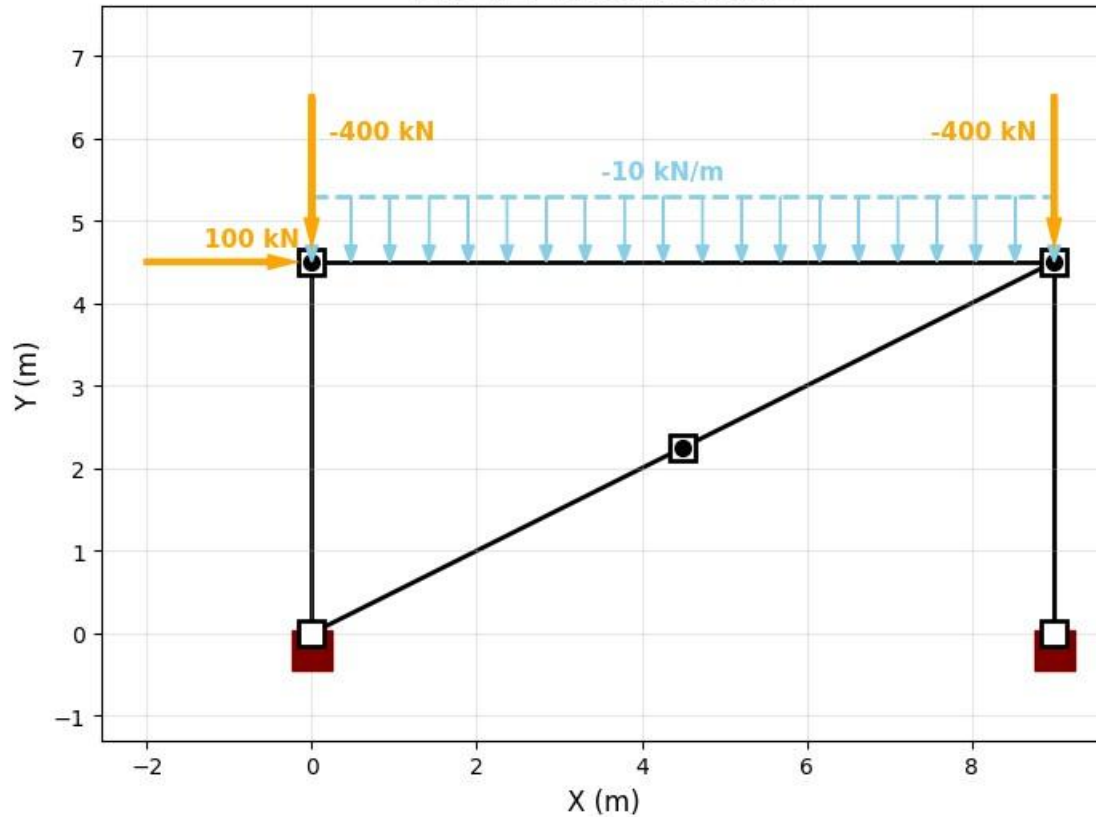
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

# --- Diagrama de Momento Flector ---
fig_m = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE MOMENTO FLECTOR (kN-m)', fontsize=14, fontweight='bold')
ax_m = plt.gca()
# sf_type='M': diagrama de momento flector
opsv.section_force_diagram_2d(sf_type='M', sfac=0.005, nep=20, ax=ax_m)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

MODELO EN OPENSEES PARA EL EJERCICIO 3 - PÓRTICOS CON RÓTULAS

SISTEMA ORIGINAL



=== REACCIONES EN APOYOS ===

Nodo 1: $R_x = -76.378 \text{ kN}$, $R_y = 419.113 \text{ kN}$, $R_z = 110.735 \text{ kN-m}$

Nodo 4: $R_x = -23.622 \text{ kN}$, $R_y = 470.887 \text{ kN}$, $R_z = 106.285 \text{ kN-m}$

=== DESPLAZAMIENTOS NODALES ===

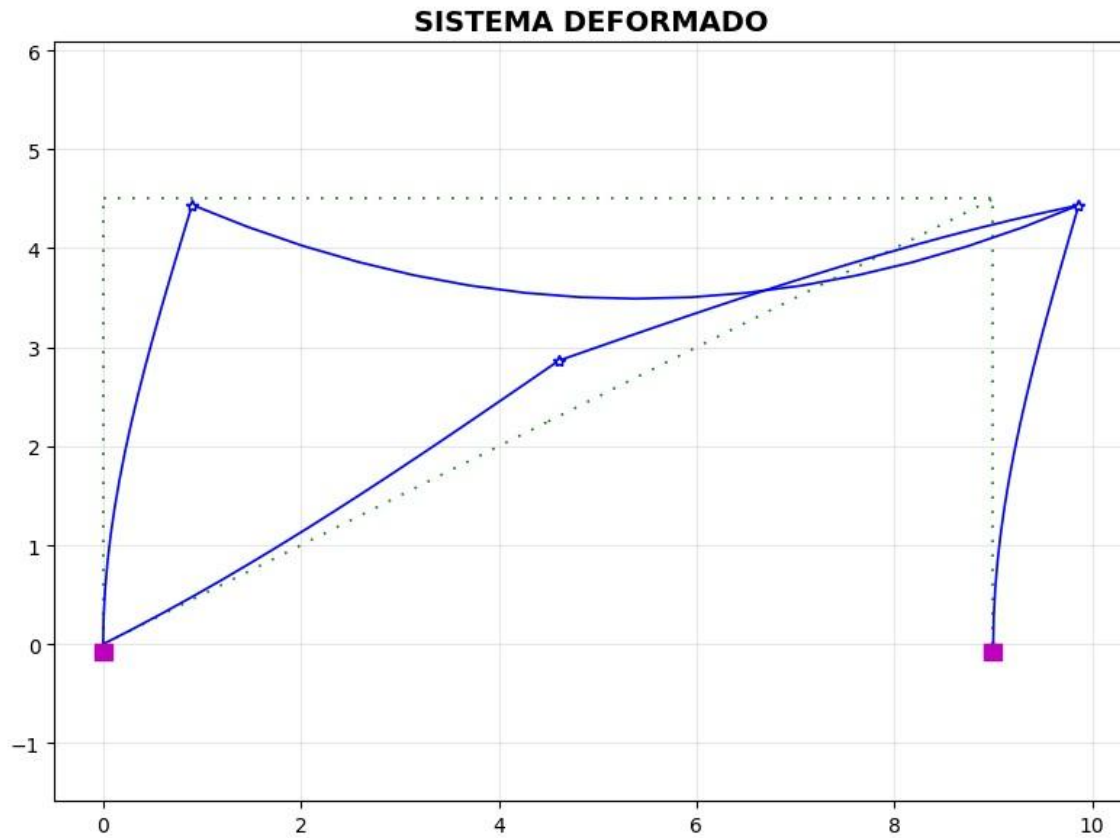
Nodo 1: $D_x = 0.000\text{e}+00 \text{ m}$, $D_y = 0.000\text{e}+00 \text{ m}$, $T_z = 0.000\text{e}+00 \text{ rad}$

Nodo 2: $D_x = 5.877\text{e}-03 \text{ m}$, $D_y = -4.037\text{e}-04 \text{ m}$, $T_z = -1.959\text{e}-03 \text{ rad}$

Nodo 3: $D_x = 5.640\text{e}-03 \text{ m}$, $D_y = -4.272\text{e}-04 \text{ m}$, $T_z = -1.880\text{e}-03 \text{ rad}$

Nodo 4: $D_x = 0.000\text{e}+00 \text{ m}$, $D_y = 0.000\text{e}+00 \text{ m}$, $T_z = 0.000\text{e}+00 \text{ rad}$

Nodo 5: $D_x = 7.051\text{e}-04 \text{ m}$, $D_y = 4.016\text{e}-03 \text{ m}$, $T_z = 9.769\text{e}-04 \text{ rad}$



=== FUERZAS INTERNAS EN ELEMENTOS ===

Elemento 1

Nodo 1: $N = 445.000 \text{ kN}$, $V = 24.611 \text{ kN}$, $M = 110.749 \text{ kN-m}$

Nodo 2: $N = -445.000 \text{ kN}$, $V = -24.611 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 2

Nodo 4: $N = 470.887 \text{ kN}$, $V = 23.622 \text{ kN}$, $M = 106.285 \text{ kN-m}$

Nodo 3: $N = -470.887 \text{ kN}$, $V = -23.622 \text{ kN}$, $M = 0.014 \text{ kN-m}$

Elemento 3

Nodo 20: $N = 75.389 \text{ kN}$, $V = 45.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 30: $N = -75.389 \text{ kN}$, $V = 45.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 4

Nodo 1: $N = -57.879 \text{ kN}$, $V = -0.003 \text{ kN}$, $M = -0.014 \text{ kN-m}$

Nodo 5: $N = 57.879 \text{ kN}$, $V = 0.003 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 5

Nodo 50: $N = -57.879 \text{ kN}$, $V = -0.003 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 3: $N = 57.879 \text{ kN}$, $V = 0.003 \text{ kN}$, $M = -0.014 \text{ kN-m}$

DIAGRAMA DE FUERZA AXIAL (kN)

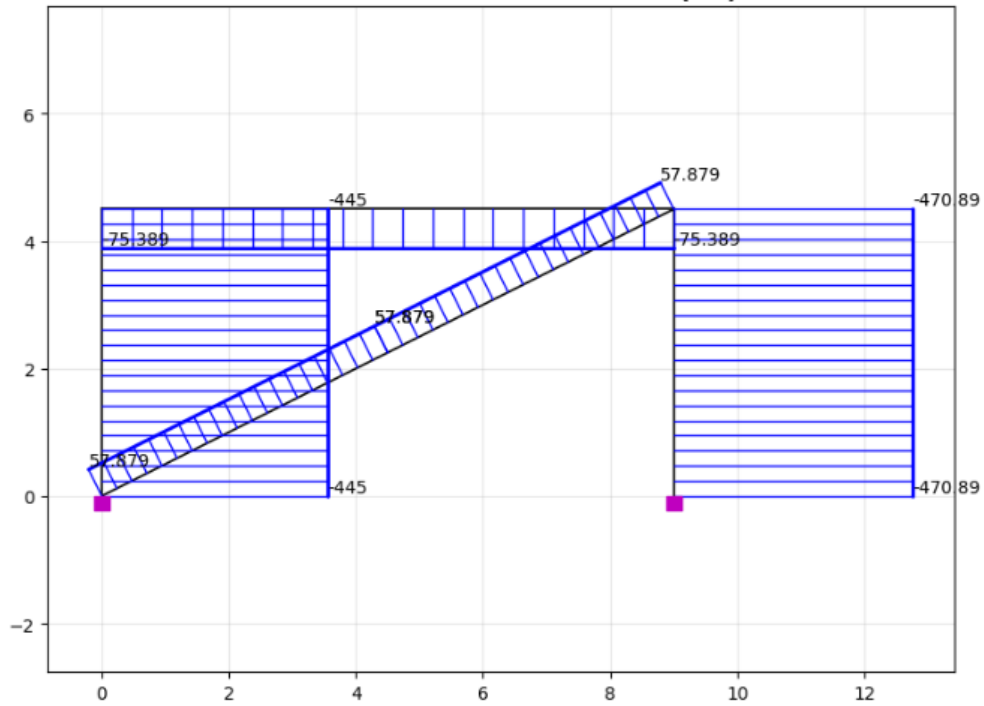
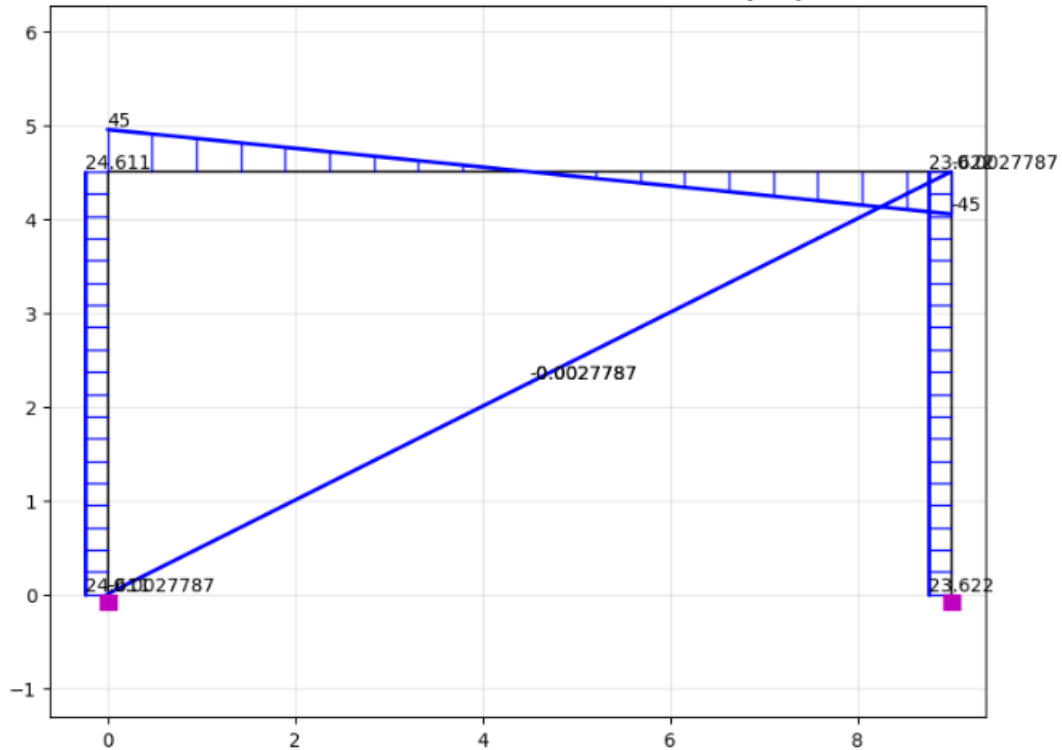
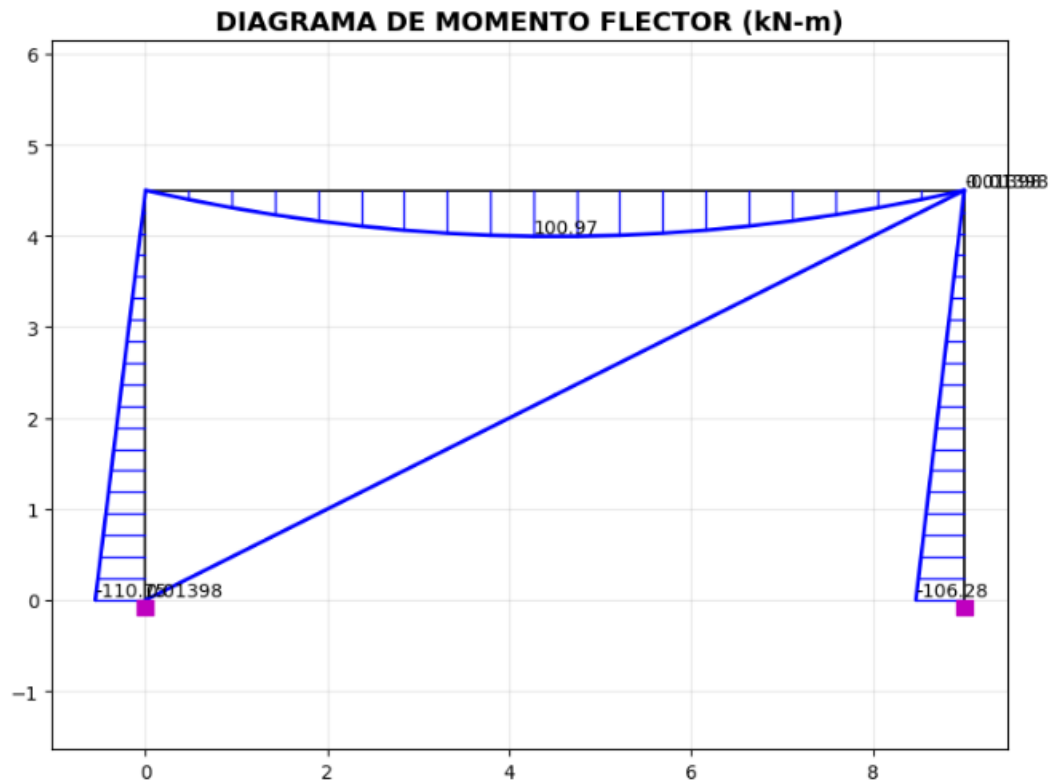


DIAGRAMA DE FUERZA CORTANTE (kN)





Sistemas Mixtos

MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON SISTEMAS MIXTOS: EJERCICIO 1

```
import numpy as np
import math
import matplotlib.pyplot as plt

print("MÉTODO MATRICIAL DE RIGIDEZ PARA ESTRUCTURAS MIXTAS: EJERCICIO 1")

# -----
# DATOS DE ENTRADA
# -----
print("\n=== DATOS DE ENTRADA ===")

# --- Módulos de elasticidad (kN/m²) ---
Ec = 24870062.324 # Hormigón
Es = 2e8          # Acero estructural
E = np.array([Ec, Ec, Es]) # Vector de módulos: [viga, columna, barra]

# --- Sección de VIGA - Elemento 1 ---
# Viga de hormigón (sección rectangular)
Av = 0.3 * 0.35 # Área (m²) = base × altura
Iv = (1/12) * 0.3 * (0.35**3) # Inercia (m⁴) = (b·h³)/12 para sección rectangular

# --- Sección de COLUMNA - Elemento 2 ---
# Columna de hormigón (sección cuadrada)
Acl = 0.3 * 0.3 # Área (m²)
Icl = (1/12) * 0.3 * (0.3**3) # Inercia (m⁴)

# --- Sección de BARRA - Elemento 3 ---
# Perfil tubular cuadrado de acero
Ab = (4*0.004) * (0.1-0.004) # Área neta (m²)
Ib = 1e-16 # Inercia despreciable (el tensor solo trabaja axialmente, se usa valor pequeño por estabilidad numérica)

# Vector de propiedades de los elementos
A = np.array([Av, Acl, Ab]) # Áreas transversales (m²)
I = np.array([Iv, Icl, Ib]) # Momentos de inercia (m⁴)

# --- Longitudes de elementos (m) ---
L = np.array([5.9, 4.0, math.sqrt((5.9**2) + (4.0**2))]) # [viga, columna, barra]

# --- Ángulos de inclinación (grados) ---
# Nota: El elemento 3 tiene ángulos diferentes en cada extremo por el apoyo inclinado
a3 = math.degrees(math.atan(4 / 5.9)) # Ángulo de inclinación del apoyo en el nodo 4
ai = np.array([0, 90, 90]) # Ángulos en nodos iniciales (°)
af = np.array([0, 90, a3]) # Ángulos en nodos finales (°)
```

```
m = 3      # Número de elementos
n = 4      # Número de nodos
GL = n * 3 # Grados de libertad totales (12: 4 nodos × 3 GL: Dx, Dy, θz)
```

Impresión de datos de entrada para verificación

```
print(f"Módulo de elasticidad: {E} kN/m²")
print(f"Área sección transversal: {A} m²")
print(f"Longitud de los elementos: {L} m")
print(f"Ángulo de inclinación inicial de los elementos: {ai} °")
print(f"Ángulo de inclinación final de los elementos: {af} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")
```

-----

COORDENADAS DE LOS NODOS

-----

```
# Formato: [x, y] en metros coordenadas_nodos = np.array([
    [0, 4],      # Nodo 1
    [5.9, 4],    # Nodo 2
    [5.9, 0],    # Nodo 3
    [0, 0]])     # Nodo 4
```

-----

CONECTIVIDAD DE ELEMENTOS

-----

Cada fila: [nodo_inicial, nodo_final]

Todos los elementos convergen al nodo 2 (nudo central)

```
Elementos = np.array([
    [1, 2],      # Elemento 1: viga
    [3, 2],      # Elemento 2: columna
    [4, 2]])     # Elemento 3: barra
```

-----

GRADOS DE LIBERTAD POR ELEMENTO

-----

Asignación de GL globales (numeración 1-based)

*# Cada nodo tiene 3 GL: (nodo-1)*3 + 1 = Dx, +2 = Dy, +3 = θz*

```
#           E1 E2 E3
Nx = np.array([1, 7, 10]) # GL Dx - nodo inicial
Ny = np.array([2, 8, 11]) # GL Dy - nodo inicial
Nz = np.array([3, 9, 12]) # GL θz - nodo inicial
Fx = np.array([4, 4, 4])  # GL Dx - nodo final
Fy = np.array([5, 5, 5])  # GL Dy - nodo final
Fz = np.array([6, 6, 6])  # GL θz - nodo final
```

```
# -----
# DEFINICIÓN DE CARGAS POR ELEMENTO
# -----
# Formato: [tipo, dirección, color, w_inicial, w_final]
Cargas = [
    ['Uniforme', 'Local_y', 'orange', -2.5, -7.5], # Elem 1: carga trapezoidal
    ['Uniforme', 'Local_y', 'slateblue', 5, 5],     # Elem 2: carga uniforme
    [] ]                                             # Elem 3: sin carga distribuida

# -----
# PREPROCESAMIENTO GEOMÉTRICO DE TODOS LOS ELEMENTOS
# -----
# Calcula propiedades geométricas fundamentales para cada elemento
# Almacena en lista 'geom' para uso posterior en gráficas y visualización
geom = []
for e in range(m):
    ni, nf = Elementos[e][0], Elementos[e][1] # Nodos inicial y final
    xi, yi = coordenadas_nodos[ni-1] # Coordenadas nodo inicial (ajuste índice 0-based)
    xf, yf = coordenadas_nodos[nf-1] # Coordenadas nodo final

    dx = xf - xi # Diferencia en X
    dy = yf - yi # Diferencia en Y
    Le = math.sqrt((dx**2) + (dy**2)) # Longitud del elemento

    # Vector director unitario (apunta del nodo inicial al final)
    ex = dx / Le
    ey = dy / Le

    # Vector normal unitario (perpendicular, rotado 90° antihorario)
    # Útil para visualización de cargas y diagramas perpendiculares al elemento
    nx = -ey
    ny = ex
    geom.append([ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny])

# -----
# GRÁFICA SISTEMA ORIGINAL
# -----
plt.figure(figsize=(7, 5.6))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos estructurales (líneas negras gruesas)
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=3, zorder=1)

# Dibujar nodos (puntos naranjas)
for i, (x, y) in enumerate(coordenadas_nodos):
    plt.plot(x, y, 'o', color='orange', markersize=10, zorder=2)
```

```
# Dibujar apoyos con simbología específica
# Nodo 1: Apoyo MÓVIL - círculo (restringe solo desplazamiento vertical)
plt.plot(0, 4-0.15, 'o', color='maroon', markersize=20, zorder=2)

# Nodo 3: Apoyo FIJO - triángulo (restringe desplazamientos horizontal y vertical)
plt.plot(5.9, -0.15, '^', color='maroon', markersize=20, zorder=2)

# Nodo 4: Apoyo FIJO INCLINADO - símbolo rotado (representación gráfica)
plt.scatter(-0.1, -0.1, marker=(3, 0, 60), s=700, color='maroon', zorder=2)

# --- Cargas distribuidas (visualización gráfica) ---
escala_visual = 0.2 # Factor de escala para visualización de flechas (solo efecto
gráfico, no afecta análisis)

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Saltar elementos sin carga
    if not Cargas[i]:
        continue
    else:
        tipo, direccion, color, wi, wf = Cargas[i]

        # Puntos a lo largo del elemento para dibujar flechas
        t_vals = np.linspace(0, 1, 20) # 20 puntos equiespaciados
        x_superior = []
        y_superior = []

        for t in t_vals:
            # Punto a lo largo del elemento (interpolación lineal)
            x = xi + t * dx
            y = yi + t * dy

            # Magnitud de carga en este punto (interpolación lineal)
            # w = wi + (wf - wi)*t es la fórmula correcta
            # El signo negativo se aplica para que las flechas apunten en dirección
            # correcta (visual)
            w_interpolada = wi + (wf - wi) * t
            w_visual = -w_interpolada # Invertir para visualización (las flechas
            apuntan hacia el elemento)

            # Base de la flecha (desplazada perpendicularmente al elemento)
            x_base = x + escala_visual * w_visual * nx
            y_base = y + escala_visual * w_visual * ny

            # Dibujar flecha individual
            plt.arrow(x_base, y_base, x-x_base, y-y_base, head_width=0.15,
                    head_length=0.2, fc=color, ec=color, length_includes_head=True)
```

```

x_superior.append(x_base)
y_superior.append(y_base)

# Dibujar línea que conecta las puntas de las flechas (contorno de la carga)
plt.plot(x_superior, y_superior, '--', color=color, linewidth=2, zorder=3)

# Etiquetas de cargas distribuidas (texto explicativo)
plt.text(1.6, 5.4, '-2.5 kN/m a -7.5 kN/m', color='orange', fontsize=11,
fontweight='bold')
plt.text(7, 2, '+5 kN/m', color='slateblue', fontsize=11, fontweight='bold')

# Carga concentrada horizontal en nodo 2 (-2 kN hacia la izquierda)
plt.arrow(7.7, 4, -1.5, 0, head_width=0.1, head_length=0.2, fc='grey', ec='grey',
linewidth=3)
plt.text(6.5, 4.2, '-2 kN', color='grey', fontweight='bold', fontsize=11,)

# Configuración del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.tight_layout()
plt.show()

# -----
# ENSAMBLE DE LA MATRIZ DE RIGIDEZ GLOBAL
# -----
print("\n=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL ===")
kG = np.zeros((GL, GL)) # Inicializar matriz de rigidez global (12x12) con ceros

# Listas para almacenar matrices de cada elemento (para post-procesamiento)
kL_elementos = [] # Matrices de rigidez local (6x6) en coordenadas Locales
T_elementos = [] # Matrices de transformación (6x6) de coordenadas globales a Locales

for i in range(m):
    # Ángulos de transformación - pueden ser DISTINTOS EN CADA EXTREMO
    theta_i = math.radians(ai[i]) # Ángulo en nodo INICIAL (convertir a radianes)
    theta_f = math.radians(af[i]) # Ángulo en nodo FINAL (convertir a radianes)

    c_i = math.cos(theta_i) # Coseno del ángulo en nodo inicial
    s_i = math.sin(theta_i) # Seno del ángulo en nodo inicial
    c_f = math.cos(theta_f) # Coseno del ángulo en nodo final
    s_f = math.sin(theta_f) # Seno del ángulo en nodo final

    # Propiedades de rigidez del elemento
    AE = A[i] * E[i] # Rigidez axial (EA) en kN
    EI = E[i] * I[i] # Rigidez flexional (EI) en kN·m²
    L2 = L[i] ** 2 # Longitud al cuadrado (m²)
    L3 = L[i] ** 3 # Longitud al cubo (m³)

```

```
# Matriz de rigidez LOCAL para elemento viga-columna biempotrado (6x6)
kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
      [0, (12*EI)/L3, (6*EI)/L2, 0, -(12*EI)/L3, (6*EI)/L2],
      [0, (6*EI)/L2, (4*EI)/L[i], 0, -(6*EI)/L2, (2*EI)/L[i]],
      [-AE/L[i], 0, 0, AE/L[i], 0, 0],
      [0, -(12*EI)/L3, -(6*EI)/L2, 0, (12*EI)/L3, -(6*EI)/L2],
      [0, (6*EI)/L2, (2*EI)/L[i], 0, -(6*EI)/L2, (4*EI)/L[i]]]
kL_elementos.append(kL)

# Matriz de transformación de coordenadas con ÁNGULOS DIFERENTES por extremo
# Los primeros 3 GL corresponden al nodo inicial, Los últimos 3 al nodo final
T = [[c_i, s_i, 0, 0, 0, 0],
      [-s_i, c_i, 0, 0, 0, 0],
      [0, 0, 1, 0, 0, 0],
      [0, 0, 0, c_f, s_f, 0],
      [0, 0, 0, -s_f, c_f, 0],
      [0, 0, 0, 0, 0, 1]]
T_elementos.append(T)

# Transformar matriz LOCAL a GLOBAL:  $K_{global} = T^T \cdot K_{local} \cdot T$ 
#  $T^T$  es la matriz de transformación inversa (de local a global)
T_T = np.transpose(T) # Transformación inversa (local a global)
kg_e = np.matmul(np.matmul(T_T, kL), T) # Matriz de rigidez del elemento en
                                         # coordenadas globales (6x6)

# Ensamblar en matriz global - Ajuste por indexación Python (0-based)
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1] # Índices 0-based

# Sumar contribución del elemento a la matriz global
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj] # Ensamblaje directo de rigidez

# Código comentado para depuración (útil para verificar cada elemento)
# print(f"ELEMENTO {i+1}")
# print(f"Módulo de elasticidad: {E[i]:.2f} kN/m², Longitud: {L[i]} m")
# print(f"Área: {A[i]} m², Inercia: {I[i]:.3e} m⁴")
# print(f"Matriz global del elemento: \n {np.round(kg_e, 0)}\n")
# print(f"Matriz global del sistema después del elemento {i+1}: \n {np.round(kG, 0)}")

print(f"Matriz global del sistema: \n {np.round(kG, 0)}")

# -----
# VECTOR DE FUERZAS DE EMPOTRAMIENTO PERFECTO (FEM)
# -----
print("\n=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===")
FEM_L_elementos = [] # FEM en coordenadas Locales por elemento
FEM_G = np.zeros(GL) # FEM en coordenadas globales (vector de 12 componentes)
```

```
for i in range(m):
    # Verificar si el elemento tiene carga distribuida
    if not Cargas[i]:
        FEM_L = [0, 0, 0, 0, 0, 0] # Sin carga → FEM = 0

    elif Cargas[i][0] == 'Uniforme' and Cargas[i][1] == 'Local_y':
        # Carga en dirección Y Local (perpendicular al elemento)
        # Convertir a magnitudes positivas para las fórmulas
        wi = -Cargas[i][3] # w inicial positiva hacia abajo (valor absoluto)
        wf = -Cargas[i][4] # w final positiva hacia abajo (valor absoluto)

        # Fórmulas analíticas para viga biempotrada con carga trapezoidal
        # Estas fórmulas se obtienen integrando las funciones de forma de la viga
        Ryi = ((7*wi*L[i])/20)+((3*wf*L[i])/20) # Reacción vertical en nodo inicial (kN)
        Mzi = ((wi*(L[i]**2))/20)+((wf*(L[i]**2))/30) # Momento en nodo inicial (kN-m)
        Ryf = ((3*wi*L[i])/20)+((7*wf*L[i])/20) # Reacción vertical en nodo final (kN)
        Mzf = -(((wi*(L[i]**2))/30)+((wf*(L[i]**2))/20)) # Momento en nodo final (kN-m)

        FEM_L = [0, Ryi, Mzi, 0, Ryf, Mzf] # Vector FEM Local

    else:
        FEM_L = [0, 0, 0, 0, 0, 0] # Otros tipos de carga no implementados

    FEM_L_elementos.append(FEM_L) # Almacenar para post-procesamiento

    # Transformar FEM Local a global: FEM_global = T^T · FEM_Local
    # T^T transforma de coordenadas locales a globales
    FEM_Ge = np.matmul(np.transpose(T_elementos[i]), FEM_L)

    # Ensamblar en vector global
    GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1] # Índices 0-based

    for jj, gl_j in enumerate(GL_elem):
        FEM_G[gl_j] += FEM_Ge[jj] # Sumar contribución del elemento

print(f"{np.round(FEM_G, 2)} kN") # Mostrar FEM global redondeado a 2 decimales

# -----
# CONDICIONES DE CONTORNO (RESTRICCIONES)
# -----
# Vector de restricciones: 1 = restringido (desplazamiento conocido = 0)
#                          0 = libre (desplazamiento desconocido)
#
#                      GL: 1  2  3  4  5  6  7  8  9  10 11 12
restricciones = np.array([0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0])

# Nodo 1: GL1(Dx)=libre, GL2(Dy)=fijo, GL3(θz)=libre → apoyo móvil
# Nodo 2: GL4(Dx)=libre, GL5(Dy)=libre, GL6(θz)=libre → nodo libre
# Nodo 3: GL7(Dx)=fijo, GL8(Dy)=fijo, GL9(θz)=libre → apoyo fijo
# Nodo 4: GL10(Dx)=fijo, GL11(Dy)=fijo, GL12(θz)=libre → apoyo fijo
```



```
# Imprimir estado de cada grado de libertad
print("\n=== RESTRICCIONES ===")
for i in range(n):
    rtx = "[Restringido]" if restricciones[i*3] == 1 else "[Libre]"
    rty = "[Restringido]" if restricciones[i*3+1] == 1 else "[Libre]"
    rtz = "[Restringido]" if restricciones[i*3+2] == 1 else "[Libre]"

    print(f"Node {i+1}: Rx={restricciones[i*3]} {rtx}, Ry={restricciones[i*3+1]}
{rty}, Rz={restricciones[i*3+2]} {rtz}")

# -----
# VECTOR DE FUERZAS EXTERNAS
# -----
# Fuerzas nodales aplicadas directamente (cargas puntuales en nodos)
#      GL: 1  2  3  4  5  6  7  8  9  10 11 12
F = np.array([0, 0, 0, -2, 0, 0, 0, 0, 0, 0, 0, 0]) # -2 kN en GL4 (Dx del nodo 2)
print(f"\n=== VECTOR DE FUERZAS EXTERNAS ===\n{F} kN")

# -----
# REDUCCIÓN DEL SISTEMA
# -----
print("\n=== REDUCCIÓN DEL SISTEMA ===")
# Identificar grados de libertad activos (no restringidos)
GL_activos = np.where(restricciones == 0)[0] # Índices donde restricciones = 0
n_GL_activos = len(GL_activos)
print(f"Grados de libertad activos: {n_GL_activos}")
print(f"Índices: {GL_activos+1}") # +1 para mostrar numeración 1-based

# Extraer submatrices correspondientes a GL activos (partición de la matriz global)
kG_reducida = kG[np.ix_(GL_activos, GL_activos)] # Matriz de rigidez reducida
F_reducido = F[GL_activos] # Vector de fuerzas reducido
FEM_G_reducido = FEM_G[GL_activos] # Vector FEM reducido
print(f"\nMatriz global reducida:\n {np.round(kG_reducida, 0)}")
print(f"Vector de fuerzas reducido: {F_reducido} kN")
print(f"Vector FEM reducido: {np.round(FEM_G_reducido, 2)} kN")

# -----
# SOLUCIÓN DEL SISTEMA
# -----
print("\n\n===== SOLUCIÓN DEL SISTEMA =====")
# Calcular la inversa de la matriz reducida
kG_r_inv = np.linalg.inv(kG_reducida) # Inversa de la matriz de rigidez reducida

# Calcular desplazamientos desconocidos:  $U = K^{-1} \cdot (F - FEM)$ 
U_desconocidos = np.matmul(kG_r_inv, F_reducido - FEM_G_reducido)

# Reconstruir vector completo de desplazamientos (incluyendo ceros en GL restringidos)
U_totales = np.zeros(GL) # Inicializar con ceros
U_totales[GL_activos] = U_desconocidos # Asignar desplazamientos calculados a GL activos
```



```
# Calcular reacciones en apoyos:  $R = K \cdot U + FEM$ 
# En los GL restringidos, esta ecuación da las fuerzas de reacción
Reacciones = np.matmul(kG, U_totales) + FEM_G

print("\n=== REACCIONES ===")
# Mostrar reacciones solo en nodos con restricciones
for i in range(n):
    # Verificar si algún GL del nodo está restringido
    if np.any(restricciones[i*3:i*3+3] == 1):
        Rx = Reacciones[i*3]      # Reacción horizontal (kN)
        Ry = Reacciones[i*3+1]    # Reacción vertical (kN)
        Mz = Reacciones[i*3+2]    # Momento de reacción (kN-m)

        print(f"Nodo {i+1}: Rx = {Rx:.2f} kN, Ry = {Ry:.2f} kN, Mz = {Mz:.2f} kN-m")

print("\n=== DESPLAZAMIENTOS ===")
# Desplazamientos nodales (notación científica para valores pequeños)
for i in range(n):
    Ux = U_totales[i*3]          # Desplazamiento horizontal (m)
    Uy = U_totales[i*3+1]        # Desplazamiento vertical (m)
    Tz = U_totales[i*3+2]        # Rotación (radianes)

    print(f"Nodo {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m,  $\theta_z = {Tz:.3e}$  rad")

# -----
# GRÁFICA SISTEMA DEFORMADO
# -----
FS = 1000    # Factor de escala para visualización (amplifica desplazamientos para ver deformación)
plt.figure(figsize=(6, 5))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar sistema original (líneas punteadas grises) como referencia
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '--', color='grey',
             linewidth=2, alpha=0.5, label='Original' if i == 0 else "")

# Dibujar sistema deformado (líneas verdes) con desplazamientos amplificados
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Desplazamientos de los nodos (del vector solución)
    desp_i = np.array([U_totales[(ni-1)*3], U_totales[(ni-1)*3+1], U_totales[(ni-1)*3+2]])
    desp_f = np.array([U_totales[(nf-1)*3], U_totales[(nf-1)*3+1], U_totales[(nf-1)*3+2]])
```

```
# Coordenadas deformadas (original + desplazamiento × factor de escala)
xi_def = xi + desp_i[0] * FS
yi_def = yi + desp_i[1] * FS
xf_def = xf + desp_f[0] * FS
yf_def = yf + desp_f[1] * FS

# Dibujar elemento deformado
plt.plot([xi_def, xf_def], [yi_def, yf_def], '-',
         color='g', linewidth=2, label='Deformada' if i == 0 else "")

# Dibujar apoyos en posición original (referencia)
plt.plot(0, 4-0.15, 'o', color='maroon', markersize=20, zorder=2)
plt.plot(5.9, -0.15, '^', color='maroon', markersize=20, zorder=2)
plt.scatter(-0.1, -0.1, marker=(3, 0, 60), s=700, color='maroon', zorder=2)

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid(True, alpha=0.5)
plt.axis('equal')
plt.tight_layout()
plt.show()

print("\n=== FUERZAS INTERNAS ===")
# Cálculo de fuerzas internas en los extremos de cada elemento
F_int_elementos = [] # Lista para almacenar las fuerzas internas de cada elemento

for i in range(m):
    # Extraer desplazamientos globales del elemento (del vector U_totales)
    Ue = np.array([
        U_totales[Nx[i]-1], # Dx nodo inicial
        U_totales[Ny[i]-1], # Dy nodo inicial
        U_totales[Nz[i]-1], # Dz nodo inicial
        U_totales[Fx[i]-1], # Dx nodo final
        U_totales[Fy[i]-1], # Dy nodo final
        U_totales[Fz[i]-1]]) # Dz nodo final

    # Transformar desplazamientos a coordenadas locales: U_local = T · U_global
    Ue_L = np.matmul(T_elementos[i], Ue)

    # Calcular fuerzas internas en coordenadas locales: F = K_Local · U_Local + FEM_Local
    Fe_L = np.matmul(kL_elementos[i], Ue_L) + FEM_L_elementos[i]
    F_int_elementos.append(Fe_L)

    # Presentar resultados en sistema local del elemento
    print(f"Elemento {i+1}:")
    print(f"  Nodo {Elementos[i][0]}: N = {Fe_L[0]:.3f} kN, V = {Fe_L[1]:.3f} kN, M = {Fe_L[2]:.3f} kN-m")
    print(f"  Nodo {Elementos[i][1]}: N = {Fe_L[3]:.3f} kN, V = {Fe_L[4]:.3f} kN, M = {Fe_L[5]:.3f} kN-m\n")
```

```
# -----
# DIAGRAMA AXIAL
# -----
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA AXIAL (kN)", fontsize=14, fontweight="bold")
escala_axial = 0.08 # Factor de escala para visualización (controla tamaño del diagrama)

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=3) # Línea del elemento (negro)

    # Extraer fuerzas axiales en extremos (del cálculo de fuerzas internas)
    Ni = F_int_elementos[i][0] # Fuerza axial en nodo inicial (kN)
    Nf = F_int_elementos[i][3] # Fuerza axial en nodo final (kN)

    # Puntos para dibujar el prisma del diagrama axial
    # El diagrama se dibuja perpendicular al elemento (dirección del vector normal)
    P1 = np.array([xi, yi]) # Punto base inicial (sobre el elemento)
    P2 = P1 + escala_axial * Ni * np.array([nx, ny]) # Desplazamiento perpendicular
    P3 = P2 + Le * np.array([ex, ey]) # Avanzar a lo largo del elemento
    P4 = P3 + escala_axial * Nf * np.array([nx, ny]) # Desplazamiento perpendicular

    # Dibujar relleno y contorno del diagrama
    # fill() dibuja un polígono relleno, plot() dibuja el contorno superior
    plt.fill([P1[0], P2[0], P3[0], P4[0]], [P1[1], P2[1], P3[1], P4[1]], alpha=0.3)
    plt.plot([P2[0], P3[0]], [P2[1], P3[1]])

    # Texto en punto medio indicando magnitud y tipo de esfuerzo
    xm = xi + ex * Le / 2 # Coordenada X del punto medio
    ym = yi + ey * Le / 2 # Coordenada Y del punto medio
    tipo = "Nula" if abs(Nf)<1e-6 else "Tracción" if Nf>0 else "Compresión" if Nf<0
    else "" # Determinar tipo de esfuerzo
    plt.text(xm, ym, f"{Nf:.2f} kN\n({tipo})", fontsize=8, ha='center',
             bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()

# -----
# DIAGRAMA CORTANTE
# -----
escala_cortante = 0.05 # Factor de escala para visualización
plt.figure(figsize=(8, 6))

plt.title("DIAGRAMA CORTANTE (kN)", fontsize=14, fontweight="bold")
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2) # Línea del elemento
```

```
# Extraer fuerzas cortantes en extremos (del cálculo de fuerzas internas)
Vi = F_int_elementos[i][1] # Cortante en nodo inicial (kN)
Vf = F_int_elementos[i][4] # Cortante en nodo final (kN)
x = np.linspace(0, Le, 30) # Puntos de evaluación a lo largo del elemento

# Obtener información de carga si existe (para diagrama parabólico)
if len(Cargas[i])>0:
    tipo, direccion, color, w1, w2 = Cargas[i]
else:
    w1 = 0
    w2 = 0

# Cálculo de la fuerza cortante variable V(x)
# Para carga transversal linealmente variable:  $V(x) = \int w_y(x) dx + V_i$ 
# La integral resulta en:  $V(x) = ((w2 - w1)/Le)*(x^2/2) + w1*x + Vi$ 
V = (((w2 - w1)/Le)*((x**2)/2)) + (w1*x) + Vi

# Coordenadas del diagrama (desplazamiento perpendicular al elemento)
# X_diag, Y_diag son los puntos del diagrama (desplazados según V(x))
X_diag = xi + ex*x + escala_cortante * V * nx
Y_diag = yi + ey*x + escala_cortante * V * ny

# Línea base del elemento (sin desplazamiento)
X_base = xi + ex*x
Y_base = yi + ey*x

# Dibujar diagrama con relleno: plot() dibuja la línea del diagrama,
# fill() rellena el área entre base y diagrama
plt.plot(X_diag, Y_diag, linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:, -1]]),
        np.concatenate([Y_base, Y_diag[:, -1]]), alpha=0.4)

# Etiquetas con los valores de cortante en los extremos
# Se muestra -Vf en el extremo j por convención gráfica
plt.text(X_diag[0], Y_diag[0], f"{{Vi:.2f}}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
plt.text(X_diag[-1], Y_diag[-1], f"{{-Vf:.2f}}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()

# -----
# DIAGRAMA MOMENTO
# -----
escala_momento = 0.05 # Factor de escala para visualización
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA MOMENTO (kN-m)", fontsize=14, fontweight="bold")
```

```
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2) # Línea del elemento

    # Extraer fuerzas internas necesarias
    Vi = F_int_elementos[i][1] # Cortante en nodo inicial (kN)
    Vf = F_int_elementos[i][4] # Cortante en nodo final (kN)
    Mi = F_int_elementos[i][2] # Momento en nodo inicial (kN-m)
    Mf = F_int_elementos[i][5] # Momento en nodo final (kN-m)
    x = np.linspace(0, Le, 30) # Puntos de evaluación a lo largo del elemento

    # Obtener información de carga si existe (para diagrama parabólico)
    if len(Cargas[i])>0:
        tipo, direccion, color, w1, w2 = Cargas[i]
    else:
        w1 = 0
        w2 = 0

    # Integración de V(x) (que ya incluye w1, w2) para obtener M(x)
    #  $M(x) = \int V(x) dx + \text{constante}$ . Siendo  $V(x) = ((w2 - w1)/Le)*(x^2/2) + w1*x + Vi$ 
    # Integrando:  $M(x) = ((w2 - w1)/Le)*(x^3/6) + w1*(x^2/2) + Vi*x + C$ 
    # La constante de integración C se determina con la condición de borde:  $M(0) = -Mi$ 
    M = (((w2 - w1)/Le)*((x**3)/6)) + (w1*((x**2)/2)) + Vi*x - Mi

    # Coordenadas del diagrama (desplazamiento perpendicular al elemento)
    X_diag = xi + ex*x + escala_momento * M * nx
    Y_diag = yi + ey*x + escala_momento * M * ny

    # Línea base del elemento
    X_base = xi + ex*x
    Y_base = yi + ey*x

    # Dibujar diagrama con relleno
    plt.plot(X_diag, Y_diag, linewidth=1.5)
    plt.fill(np.concatenate([X_base, X_diag[::-1]]),
            np.concatenate([Y_base, Y_diag[::-1]]), alpha=0.4)

    # Etiquetas con los valores de momento en los extremos
    # Se muestra -Mi en el extremo i para mantener la convención de que el momento se
    # dibuja. El signo positivo en Mf se muestra directamente
    plt.text(X_diag[0], Y_diag[0], f"{{-Mi:.2f}}", fontsize=8,
            bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
    plt.text(X_diag[-1], Y_diag[-1], f"{{Mf:.2f}}", fontsize=8,
            bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()
```

MÉTODO MATRICIAL DE RIGIDEZ PARA ESTRUCTURAS MIXTAS: EJERCICIO 1

=== DATOS DE ENTRADA ===

Módulo de elasticidad: [2.48700623e+07 2.48700623e+07 2.00000000e+08] kN/m²

Área sección transversal: [0.105 0.09 0.001536] m²

Longitud de los elementos: [5.9 4. 7.12811335] m

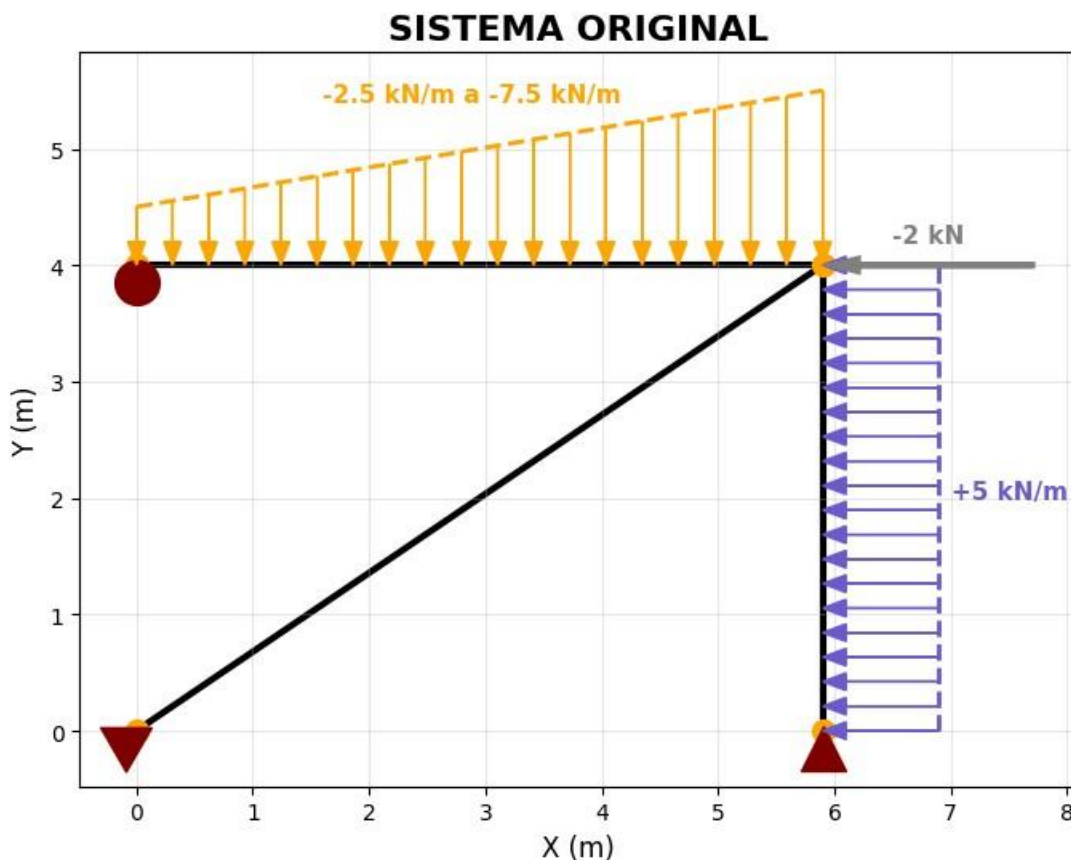
Ángulo de inclinación inicial de los elementos: [0 90 90] °

Ángulo de inclinación final de los elementos: [0. 90. 34.13594008] °

Número de elementos: 3

Número de nodos: 4

Número de grados de libertad: 12



=== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL ===

Matriz global del sistema:

[442603.	0.	0.	-442603.	0.	0.	0.	0.	0.
0.	0.	0.]							
[0.	1558.	4595.	0.	-1558.	4595.	0.	0.	0.
0.	0.	0.]							
[0.	4595.	18073.	0.	-4595.	9036.	0.	0.	0.
0.	0.	0.]							
[-442603.	0.	0.	475276.	20017.	6295.	-3148.	-0.	6295.
-0.	-35672.	0.]							



```
[ 0. -1558. -4595. 20017. 574705. -4595. -0. -559576. -0.
0. -24184. -0.]
[ 0. 4595. 9036. 6295. -4595. 34860. -6295. 0. 8394.
-0. 0. 0.]
[ 0. 0. 0. -3148. -0. -6295. 3148. 0.
-6295. 0. 0. 0.]
[ 0. 0. 0. -0. -559576. 0. 0. 559576. 0.
0. 0. 0.]
[ 0. 0. 0. 6295. -0. 8394. -6295. 0.
16787. 0. 0. 0.]
[ 0. 0. 0. -0. 0. -0. 0. 0. 0.
0. 0. -0.]
[ 0. 0. 0. -35672. -24184. 0. 0. 0. 0.
0. 43097. 0.]
[ 0. 0. 0. 0. -0. 0. 0. 0.
0. -0. 0. 0.]
```

=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===

```
[ 0. 11.8 13.05 10. 17.7 -9.29 10. -0. -6.67 0. 0. 0. ] kN
```

=== RESTRICCIONES ===

Nodo 1: Rx=0 [Libre], Ry=1 [Restringido], Rz=0 [Libre]

Nodo 2: Rx=0 [Libre], Ry=0 [Libre], Rz=0 [Libre]

Nodo 3: Rx=1 [Restringido], Ry=1 [Restringido], Rz=0 [Libre]

Nodo 4: Rx=1 [Restringido], Ry=1 [Restringido], Rz=0 [Libre]

=== VECTOR DE FUERZAS EXTERNAS ===

```
[ 0 0 0 -2 0 0 0 0 0 0 0 0 ] kN
```

=== REDUCCIÓN DEL SISTEMA ===

Grados de libertad activos: 7

Índices: [1 3 4 5 6 9 12]

Matriz global reducida:

```
[ 442603. 0. -442603. 0. 0. 0. 0.
[ 0. 18073. 0. -4595. 9036. 0. 0.
[-442603. 0. 475276. 20017. 6295. 6295. 0.
[ 0. -4595. 20017. 574705. -4595. -0. -0.
[ 0. 9036. 6295. -4595. 34860. 8394. 0.
[ 0. 0. 6295. -0. 8394. 16787. 0.
[ 0. 0. 0. -0. 0. 0. 0.]
```

Vector de fuerzas reducido: [0 0 -2 0 0 0 0] kN

Vector FEM reducido: [0. 13.05 10. 17.7 -9.29 -6.67 0.] kN

===== SOLUCIÓN DEL SISTEMA =====

=== REACCIONES ===

Nodo 1: $R_x = 0.00 \text{ kN}$, $R_y = 9.73 \text{ kN}$, $M_z = -0.00 \text{ kN-m}$

Nodo 3: $R_x = 6.22 \text{ kN}$, $R_y = 9.07 \text{ kN}$, $M_z = 0.00 \text{ kN-m}$

Nodo 4: $R_x = -0.00 \text{ kN}$, $R_y = 19.07 \text{ kN}$, $M_z = -0.00 \text{ kN-m}$

=== DESPLAZAMIENTOS ===

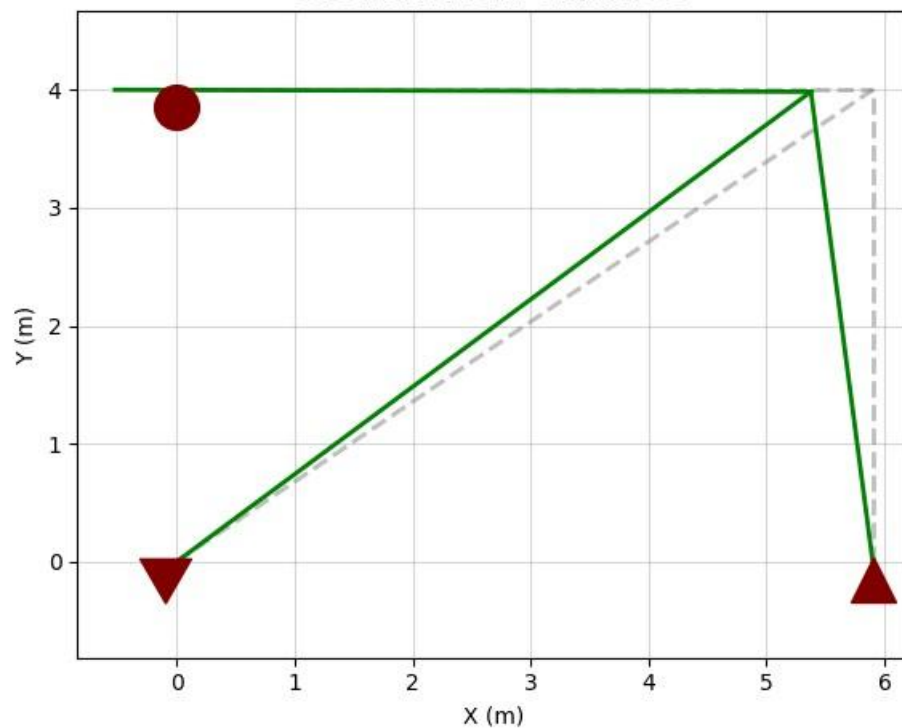
Nodo 1: $U_x = -5.236e-04 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = -9.959e-04 \text{ rad}$

Nodo 2: $U_x = -5.236e-04 \text{ m}$, $U_y = -1.621e-05 \text{ m}$, $\theta_z = 5.390e-04 \text{ rad}$

Nodo 3: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = 3.240e-04 \text{ rad}$

Nodo 4: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = -2.105e-04 \text{ rad}$

SISTEMA DEFORMADO



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 0.000 \text{ kN}$, $V = 9.726 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 2: $N = 0.000 \text{ kN}$, $V = 19.774 \text{ kN}$, $M = -15.138 \text{ kN-m}$

Elemento 2:

Nodo 3: $N = 9.073 \text{ kN}$, $V = -6.215 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 2: $N = -9.073 \text{ kN}$, $V = -13.785 \text{ kN}$, $M = 15.138 \text{ kN-m}$

Elemento 3:

Nodo 4: $N = 19.070 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 2: $N = -19.070 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

DIAGRAMA AXIAL (kN)

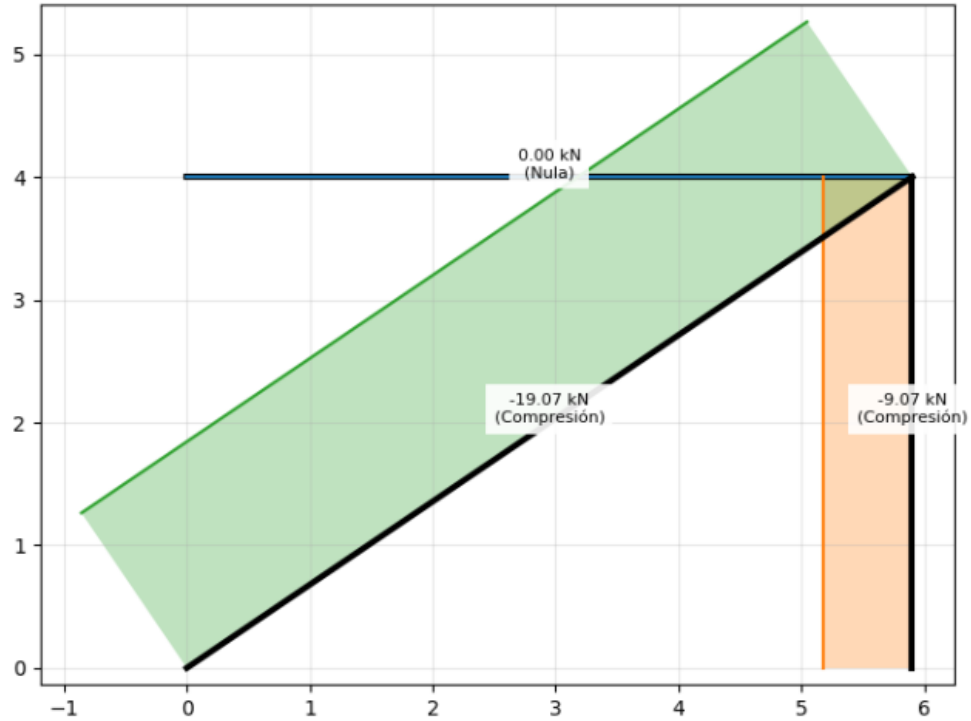


DIAGRAMA CORTANTE (kN)

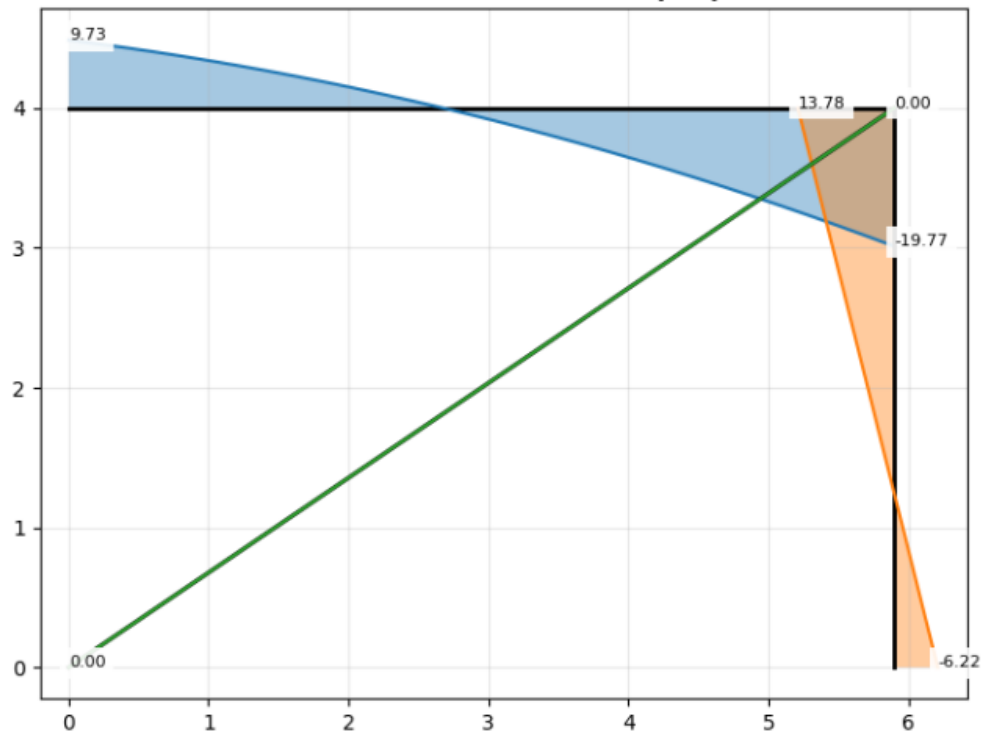
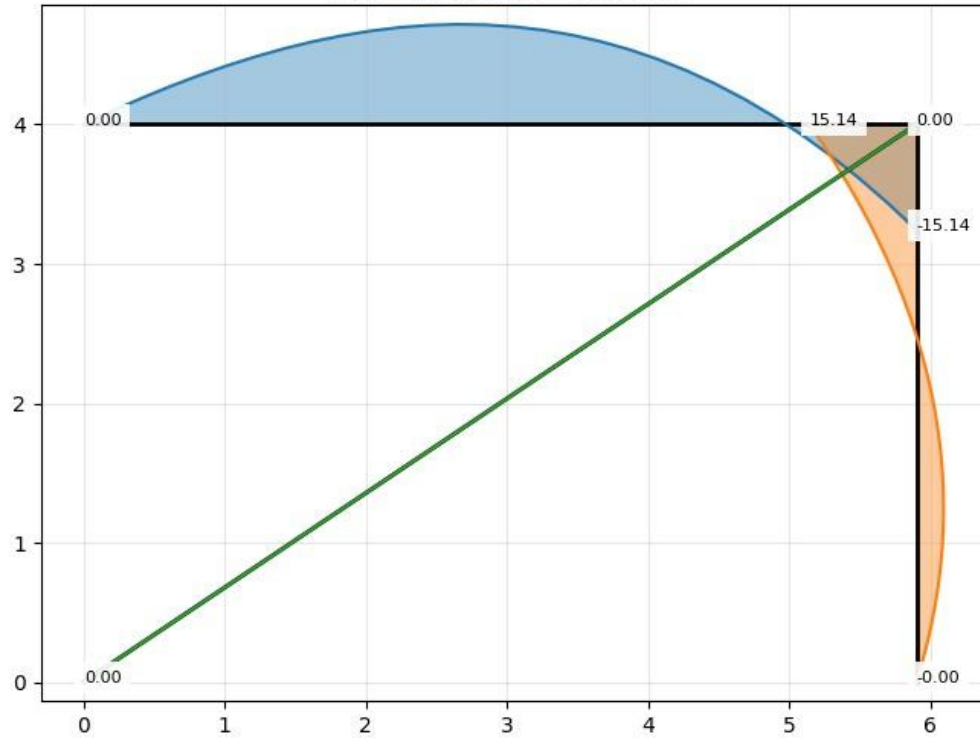


DIAGRAMA MOMENTO



MODELO OPENSEES

SISTEMAS MIXTOS: EJERCICIO 1

```
# -----
# IMPORTACIÓN DE LIBRERÍAS
# -----
import openseespy.opensees as ops #Biblioteca principal de OpenSees para análisis estructural
import opsviz as opsv # Extensión para visualización de resultados
import numpy as np # Para operaciones numéricas y arreglos
import math # Para funciones matemáticas básicas (atan, sqrt, etc.)
import matplotlib.pyplot as plt # Para visualización personalizada de resultados

print("MODELO EN OPENSEES PARA EL EJERCICIO 1 - SISTEMAS MIXTOS")

# -----
# INICIALIZACIÓN DEL MODELO
# -----
ops.wipe() # Limpiar memoria de análisis previos (eliminar modelos anteriores)

# Modelo 2D con 3 grados de libertad por nodo (dx, dy, θz)
# -ndm: número de dimensiones espaciales (2: plano XY)
# -ndf: número de grados de libertad por nodo (3: desplazamientos X, Y y rotación Z)
ops.model("Basic", "-ndm", 2, "-ndf", 3)

# -----
# DEFINICIÓN DE NODOS PRINCIPALES
# -----
ops.node(1, 0.0, 4.0) # Nodo 1
ops.node(2, 5.9, 4.0) # Nodo 2
ops.node(3, 5.9, 0.0) # Nodo 3
ops.node(4, 0.0, 0.0) # Nodo 4

# -----
# CONDICIONES DE APOYO
# -----
# Nodo 1: apoyo MÓVIL - solo restringe desplazamiento vertical (Dy)
ops.fix(1, 0, 1, 0) # [Dx: Libre, Dy: fijo, θz: Libre]

# Nodo 3: apoyo FIJO - restringe desplazamientos horizontal y vertical (Dx, Dy)
ops.fix(3, 1, 1, 0) # [Dx: fijo, Dy: fijo, θz: Libre]

# Nodo 4: apoyo FIJO INCLINADO - restringe desplazamientos horizontal y vertical (Dx, Dy) NOTA: Aunque se menciona "inclinado", la restricción es en dirección de los ejes globales
ops.fix(4, 1, 1, 0) # [Dx: fijo, Dy: fijo, θz: Libre]
```

```
# -----
# DEFINICIÓN DE MATERIALES
# -----
Ec = 24870062.32 # Módulo de elasticidad del hormigón (kN/m²) - vigas y columnas
Es = 2e8 # Módulo de elasticidad del acero (kN/m²) - barra

# -----
# PROPIEDADES DE SECCIONES
# -----
# --- Sección de VIGA (hormigón) ---
Av = 0.105 # Área de sección transversal (m²)
Iv = 0.001071875 # Momento de inercia (m⁴)

# --- Sección de COLUMNA (hormigón) ---
Ac = 0.09 # Área de sección transversal (m²)
Ic = 0.000675 # Momento de inercia (m⁴)

# --- Sección de BARRA (perfil tubular de acero) ---
Ab = 0.001536 # Área de sección transversal (m²)
Ib = 1e-16 # Inercia despreciable (valor pequeño pero no nulo por
            # estabilidad numérica)
# Nota: La barra se modela como elemento viga-columna con inercia muy pequeña,
# aunque conceptualmente trabaja solo a tracción/compresión axial

# -----
# TRANSFORMACIÓN GEOMÉTRICA
# -----
# Transformación lineal para elementos viga-columna. Define la orientación y
# comportamiento geométrico de los elementos. Parámetros: (tipo, tag)
ops.geomTransf("Linear", 1) # Transformación única para todos los elementos

# -----
# DEFINICIÓN DE SECCIONES ELÁSTICAS
# -----
# Se definen secciones elásticas para cada tipo de elemento
# Parámetros: ('Elastic', tag, E, A, I)
ops.section('Elastic', 1, Ec, Av, Iv) # Sección de viga (hormigón) - Tag=1
ops.section('Elastic', 2, Ec, Ac, Ic) # Sección de columna (hormigón) - Tag=2
ops.section('Elastic', 3, Es, Ab, Ib) # Sección de barra (acero) - Tag=3

# -----
# ESQUEMA DE INTEGRACIÓN
# -----
# Integración de Lobatto a lo largo del elemento para elementos viga-columna
# Parámetros: ('Lobatto', tag_integración, tag_sección, puntos_integración)
ops.beamIntegration('Lobatto', 1, 1, 10) # Para viga: 10 puntos (alta precisión)
ops.beamIntegration('Lobatto', 2, 2, 10) # Para columna: 10 puntos
ops.beamIntegration('Lobatto', 3, 3, 2) # Para barra: solo 2 puntos (suficiente
para sección con inercia despreciable)
```

```
# -----
# ELEMENTOS
# -----
Elementos = [] # Lista para almacenar información de elementos (usada en post-
               # procesamiento y gráficas)

# --- Elemento 1: Viga (Discretizada para carga trapezoidal) ---
nDiv = 100     # Número de subdivisiones para discretizar la viga (mayor precisión
               # para carga variable)
nodeStart = 100 # Tag inicial para nodos intermedios (evita conflicto con nodos
               # principales 1-4)
elemStart = 100 # Tag inicial para subelementos

# Coordenadas de la viga (nodos 1 y 2)
xi, yi = ops.nodeCoord(1)
xj, yj = ops.nodeCoord(2)
L = math.sqrt((xj-xi)**2 + (yj-yi)**2) # Longitud de la viga = 5.9 m (horizontal)

# Crear nodos intermedios para la discretización
beamNodes = [1] # Comienza con nodo 1 existente (extremo izquierdo)

for i in range(1, nDiv):
    # Interpolación lineal de coordenadas entre nodo 1 y nodo 2
    x = xi + (xj - xi) * i / nDiv
    y = yi + (yj - yi) * i / nDiv
    tag = nodeStart + i
    ops.node(tag, x, y) # Crear nodo intermedio
    beamNodes.append(tag)

beamNodes.append(2) # Termina con nodo 2 existente

# Crear subelementos entre nodos consecutivos
beamElems = []

for i in range(nDiv):
    eTag = elemStart + i
    # Elemento viga-columna con desplazamientos (dispBeamColumn)
    # Parámetros: (tipo, tag, nodo_i, nodo_j, transfTag, beamIntTag)
    ops.element('dispBeamColumn', eTag, beamNodes[i], beamNodes[i+1], 1, 1)
    beamElems.append(eTag)

# Registrar elemento conceptual en la lista principal (para visualización)
Elementos.append({"ID": 1, "Nodo_i": 1, "Nodo_j": 2})

# --- Elemento 2: Columna ---
# Elemento viga-columna entre nodo 3 y nodo 2
ops.element('dispBeamColumn', 2, 3, 2, 1, 2)
Elementos.append({"ID": 2, "Nodo_i": 3, "Nodo_j": 2})
```

```
# --- Elemento 3: Barra diagonal ---
# Elemento viga-columna entre nodo 4 y nodo 2. Aunque es una barra, se modela como
# viga-columna con inercia muy pequeña
ops.element('dispBeamColumn', 3, 4, 2, 1, 3)
Elementos.append({"ID": 3, "Nodo_i": 4, "Nodo_j": 2})

# -----
# DEFINICIÓN DE CARGAS
# -----
ops.timeSeries("Linear", 1) # Serie temporal lineal (carga proporcional al factor de
# tiempo)
ops.pattern("Plain", 1, 1) # Patrón de carga estático (tag=1, timeSeries=1)

# --- Elemento 1: Carga trapezoidal ---
w_i = -2.5 # Carga inicial en nodo 1 (kN/m) - negativo = hacia abajo
w_f = -7.5 # Carga final en nodo 2 (kN/m) - negativo = hacia abajo

# Aplicar carga uniforme por tramo (aproximación de carga trapezoidal mediante tramos
# uniformes)
for i in range(nDiv):
    # Posición local del inicio y fin del subelemento (0 a L)
    x1 = L * i / nDiv
    x2 = L * (i + 1) / nDiv

    # Interpolación lineal de la carga a lo largo de la viga
    w1 = w_i + (w_f - w_i) * x1 / L
    w2 = w_i + (w_f - w_i) * x2 / L

    # Carga promedio (uniforme equivalente) para el subelemento
    w_prom = (w1 + w2) / 2

    # Aplicar carga uniforme al subelemento (en coordenadas locales del elemento)
    # NOTA: "-beamUniform" aplica carga perpendicular en el eje local del elemento
    ops.eleLoad("-ele", beamElems[i], "-type", "-beamUniform", w_prom)

# --- Elemento 2: Carga uniformemente distribuida en columna ---
# Aplica carga uniforme de +5 kN/m en dirección perpendicular al elemento
# El signo positivo indica dirección según el sistema local (en este caso, hacia la
# derecha).
#Parámetros: (elemento, tipo_carga, magnitud_perpendicular, magnitud_axial)
ops.eleLoad("-ele", 2, "-type", "-beamUniform", 5.0, 0.0)

# --- Nodo 2: Carga puntual ---
# Parámetros: (nodo, Fx, Fy, Mz)
ops.load(2, -2.0, 0.0, 0.0) # -2 kN en dirección X (hacia la izquierda)
```

```
# -----
# GRÁFICA SISTEMA ORIGINAL
# -----
# Definición de cargas para visualización personalizada (NO afecta el análisis)
# Formato: [tipo, dirección, color, valor_inicial, valor_final]
Cargas = [
    ['Uniforme', 'Local_y', 'teal', -2.5, -7.5], # Viga: trapezoidal
    ['Uniforme', 'Local_y', 'blue', 5, 5],       # Columna: uniforme
    []                                             # Barra: sin carga distribuida

plt.figure(figsize=(7, 5.6))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos (líneas negras continuas)
for element_data in Elementos:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos del elemento
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Dibujar el elemento
    plt.plot([xi, xf], [yi, yf], 'k-', lw=2, zorder=1)

# Dibujar nodos principales (puntos morados)
for i in range(1, 5):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'o', color='purple', markersize=8, zorder=2)

# Dibujar apoyos con simbología específica
plt.plot(0, 4-0.15, 'o', color='maroon', markersize=20, zorder=2) # Apoyo móvil nodo 1
plt.plot(5.9, -0.15, '^', color='maroon', markersize=20, zorder=2) # Apoyo fijo nodo 3
plt.scatter(-0.1, -0.1, marker=(3, 0, 60), s=700, color='maroon', zorder=2) # Apoyo fijo nodo 4

# --- Dibujo cargas distribuidas ---
escala = 0.2 # Factor de escala para longitud de flechas (visual)
for i, element_data in enumerate(Elementos):
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    if not Cargas[i]:
        continue # Continuar si el elemento no tiene cargas definidas en la lista

    tipo, direccion, color, w1, w2, *extra = Cargas[i]

    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)
```

```

dx = xf - xi
dy = yf - yi
L_elem = (dx**2 + dy**2)**0.5

# Vectores unitarios del sistema local del elemento
ex = dx / L_elem # Vector dirección axial
ey = dy / L_elem # Vector dirección transversal (perpendicular)

# Vector normal (perpendicular al elemento) para dirección de la carga
nx = -ey
ny = ex

t_vals = np.linspace(0, 1, 20) # 20 puntos a lo largo del elemento
x_superior = []
y_superior = []

for t in t_vals:
    x = xi + t * dx
    y = yi + t * dy

    # Magnitud de carga (interpolación lineal)
    w = -(w1 + (w2 - w1) * t) # Signo negativo por convención de visualización
    # Base de la flecha (desplazada perpendicularmente)
    x_base = x + escala * w * nx
    y_base = y + escala * w * ny

    # Dibujar flecha desde la base hacia el elemento
    plt.arrow(x_base, y_base, x - x_base, y - y_base, head_width=0.15,
              head_length=0.2, fc=color, ec=color, length_includes_head=True)

    x_superior.append(x_base)
    y_superior.append(y_base)

# Línea de carga (conecta las puntas de las flechas)
plt.plot(x_superior, y_superior, '--', color=color, linewidth=2, zorder=3)

# Etiquetas de cargas (texto explicativo)
plt.text(1.6, 5.4, '-2.5 kN/m a -7.5 kN/m', color='teal', fontsize=11,
         fontweight='bold', zorder=5)
plt.text(7, 2, '+5 kN/m', color='blue', fontsize=11, fontweight='bold', zorder=5)

# Carga puntual en nodo 2 (flecha naranja)
plt.arrow(7.7, 4, -1.5, 0, head_width=0.1, head_length=0.2, fc='coral', ec='coral',
         linewidth=3, zorder=4)
plt.text(6.5, 4.2, '-2 kN', color='coral', fontweight='bold', fontsize=11, zorder=5)

# Configuración del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)

```



```
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.tight_layout()
plt.show()

# -----
# CONFIGURACIÓN DEL ANÁLISIS
# -----
ops.system("BandSPD") # Solver para matrices banda simétrica definida positiva
ops.numberer("RCM") # Numeración de grados de libertad optimiza el ancho de banda
ops.constraints("Plain") # Imposición directa de restricciones (método de eliminación)
ops.integrator("LoadControl", 1.0) # Control por carga: aplica el 100% de la carga en 1 paso
ops.algorithm("Linear") # Algoritmo de solución lineal (suficiente para análisis
                        # elástico lineal)
ops.analysis("Static") # Tipo de análisis: estático
ops.analyze(1) # Ejecutar análisis (1 paso)

# -----
# RESULTADOS
# -----
print("\n===== RESULTADOS =====")

print("\n=== REACCIONES ===")
ops.reactions() # Calcular reacciones en los nodos con restricciones

for i in [1, 3, 4]:
    Rx = ops.nodeReaction(i, 1) # Reacción en X (kN)
    Ry = ops.nodeReaction(i, 2) # Reacción en Y (kN)
    Rz = ops.nodeReaction(i, 3) # Momento de reacción (kN-m)
    print(f"Node {i}: Rx = {Rx:.2f} kN, Ry = {Ry:.2f} kN, Mz = {Rz:.2f} kN-m")

print("\n=== DESPLAZAMIENTOS ===")
for i in range(1, 5):
    ux, uy, uz = ops.nodeDisp(i) # [dx, dy, dz] en metros y radianes
    print(f"Node {i}: Ux = {ux:.3e} m, Uy = {uy:.3e} m, Uz = {uz:.3e} rad")

# -----
# GRÁFICA SISTEMA DEFORMADO
# -----
fig, ax = plt.subplots(figsize=(8, 6)) # Crear figura para la deformada
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar la deformada usando opsviz. El parámetro 'ax' especifica los ejes donde
# dibujar
opsv.plot_defo(ax=ax)
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

```
print("\n=== FUERZAS INTERNAS ===")

# --- Elemento 1: Viga (resultados en los extremos) ---
F_ini = ops.eleResponse(beamElems[0], 'localForces') # Primer subelemento (nodo 1)
F_fin = ops.eleResponse(beamElems[-1], 'localForces') # Último subelemento (nodo 2)

# Formato LocalForces: [N_i, V_i, M_i, N_j, V_j, M_j] en sistema local del elemento
print("Elemento 1: Viga")
print(f"Node 1: N = {F_ini[0]:.3f} kN, V = {F_ini[1]:.3f} kN, M = {F_ini[2]:.3f} kN-m")
print(f"Node 2: N = {F_fin[3]:.3f} kN, V = {F_fin[4]:.3f} kN, M = {F_fin[5]:.3f} kN-m")

# --- Elemento 2: Columna ---
F_col = ops.eleResponse(2, 'localForces')
print("\nElemento 2: Columna")
print(f"Node 2: N = {F_col[0]:.3f} kN, V = {F_col[1]:.3f} kN, M = {F_col[2]:.3f} kN-m")
print(f"Node 3: N = {F_col[3]:.3f} kN, V = {F_col[4]:.3f} kN, M = {F_col[5]:.3f} kN-m")

# --- Elemento 3: Barra ---
F_truss = ops.eleResponse(3, 'localForces') # Para elemento viga-columna

print("\nElemento 3: Barra")
# Para un barra con inercia despreciable, la fuerza axial es el componente relevante
print(f"Fuerza axial = {F_truss[0]:.3f} kN {'(tracción)' if F_truss[0] < 0 else '(compresión)'}")
# NOTA: El signo de la fuerza axial: negativo = tracción (convención de OpenSees)

# -----
# DIAGRAMAS DE ESFUERZOS
# -----
# --- Diagrama de Fuerza Axial ---
fig_n = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA AXIAL (kN)', fontsize=14, fontweight='bold')
ax_n = plt.gca()

# sf_type: tipo de fuerza ('N' = axial, 'V' = cortante, 'M' = momento)
# sfac: factor de escala para visualización
# nep: número de puntos para dibujar el diagrama
opsv.section_force_diagram_2d(sf_type='N', sfac=0.08, nep=20, ax=ax_n)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

```
# --- Diagrama de Fuerza Cortante -
fig_v = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA CORTANTE (kN)', fontsize=14, fontweight='bold')
ax_v = plt.gca()
opsv.section_force_diagram_2d(sf_type='V', sfac=0.05, nep=20, ax=ax_v)

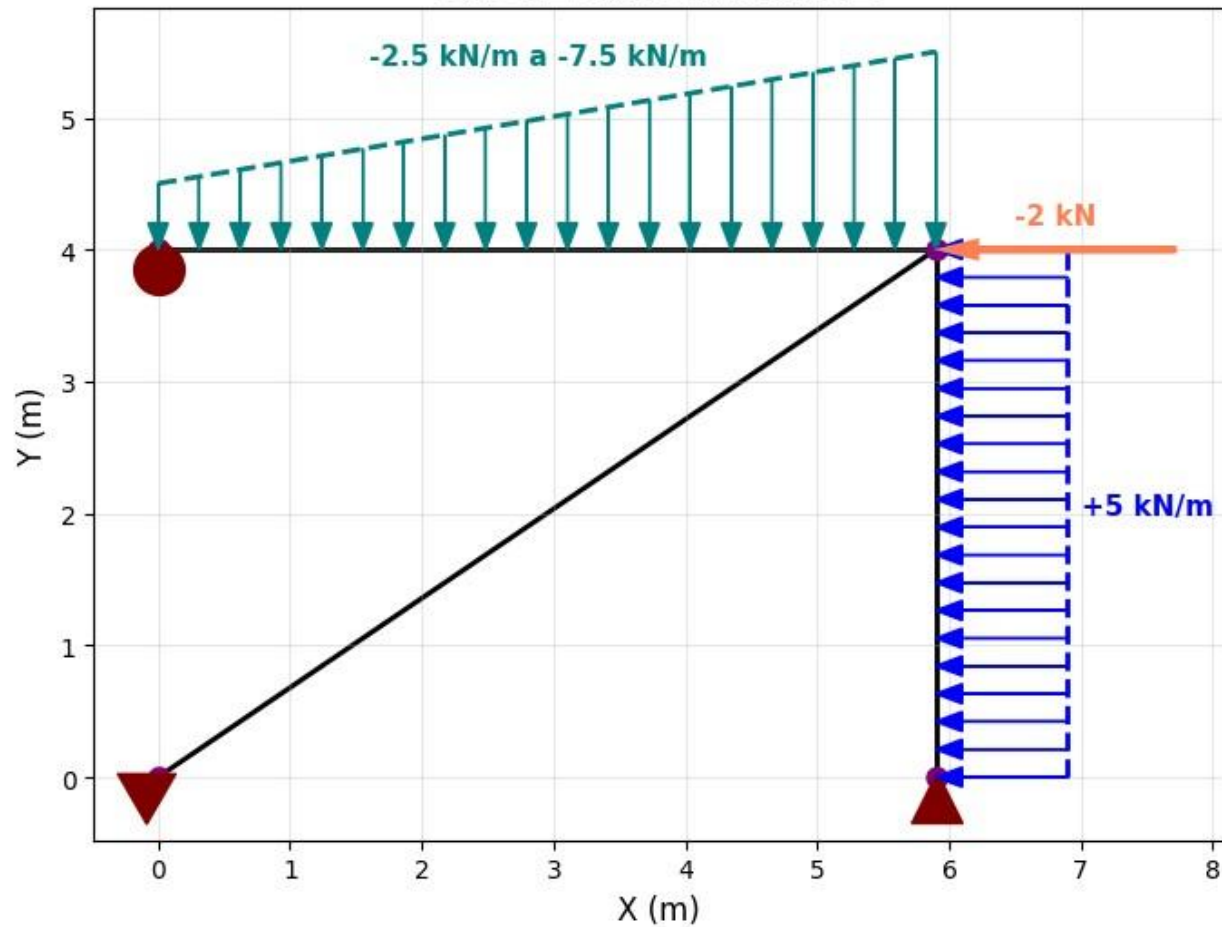
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

# --- Diagrama de Momento Flector -
fig_m = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE MOMENTO FLECTOR (kN-m)', fontsize=14, fontweight='bold')
ax_m = plt.gca()
opsv.section_force_diagram_2d(sf_type='M', sfac=0.05, nep=20, ax=ax_m)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

MODELO EN OPENSEES PARA EL EJERCICIO 1 - SISTEMAS MIXTOS

SISTEMA ORIGINAL



===== RESULTADOS =====

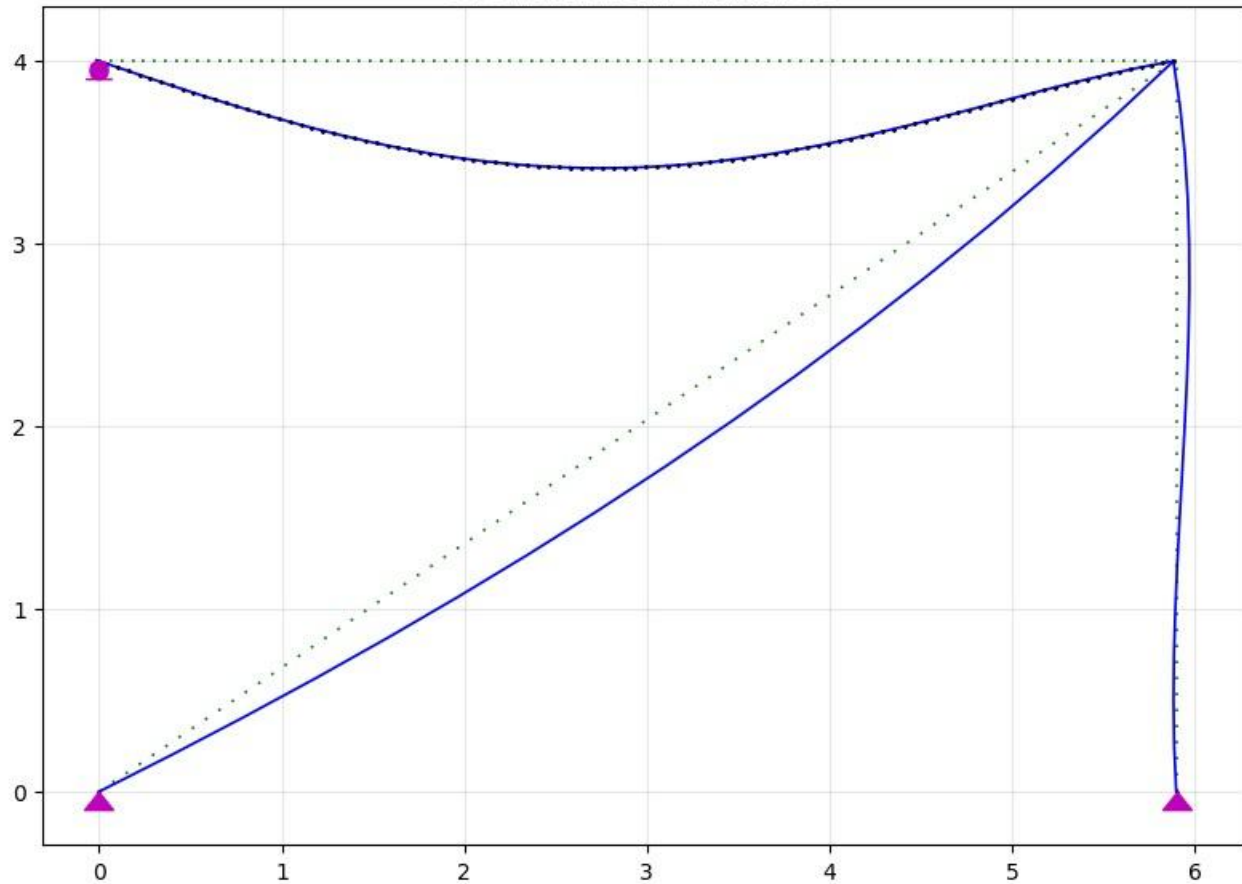
=== REACCIONES ===

Nodo 1: $R_x = 0.00 \text{ kN}$, $R_y = 9.73 \text{ kN}$, $M_z = -0.00 \text{ kN-m}$
 Nodo 3: $R_x = 6.22 \text{ kN}$, $R_y = 9.07 \text{ kN}$, $M_z = -0.00 \text{ kN-m}$
 Nodo 4: $R_x = 15.78 \text{ kN}$, $R_y = 10.70 \text{ kN}$, $M_z = 0.00 \text{ kN-m}$

=== DESPLAZAMIENTOS ===

Nodo 1: $U_x = -5.236e-04 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = -9.959e-04 \text{ rad}$
 Nodo 2: $U_x = -5.236e-04 \text{ m}$, $U_y = -1.621e-05 \text{ m}$, $\theta_z = 5.390e-04 \text{ rad}$
 Nodo 3: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = 3.240e-04 \text{ rad}$
 Nodo 4: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = -3.604e-04 \text{ rad}$

SISTEMA DEFORMADO



=== FUERZAS INTERNAS ===

Elemento 1: Viga

Nodo 1: $N = 0.000$ kN, $V = 9.726$ kN, $M = -0.000$ kN-m

Nodo 2: $N = -0.000$ kN, $V = 19.774$ kN, $M = -15.138$ kN-m

Elemento 2: Columna

Nodo 2: $N = 9.072$ kN, $V = -6.215$ kN, $M = -0.000$ kN-m

Nodo 3: $N = -9.072$ kN, $V = -13.785$ kN, $M = 15.138$ kN-m

Elemento 3: Barra

Fuerza axial = 19.070 kN (compresión)

DIAGRAMA DE FUERZA AXIAL (kN)

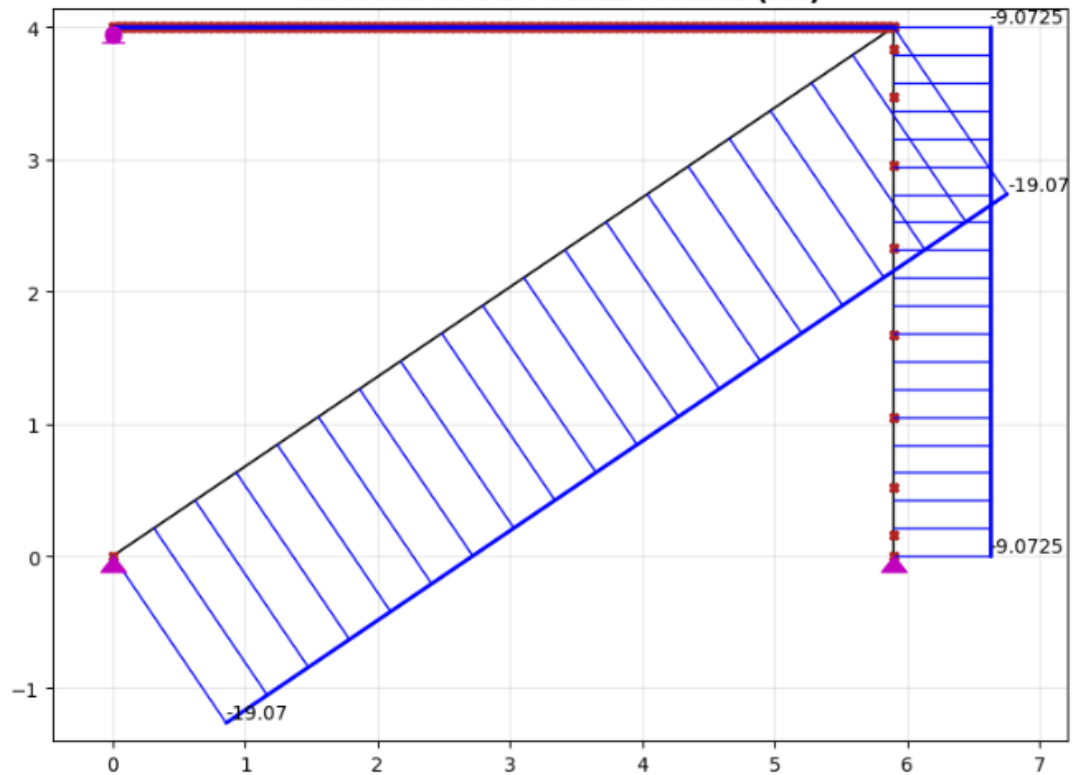
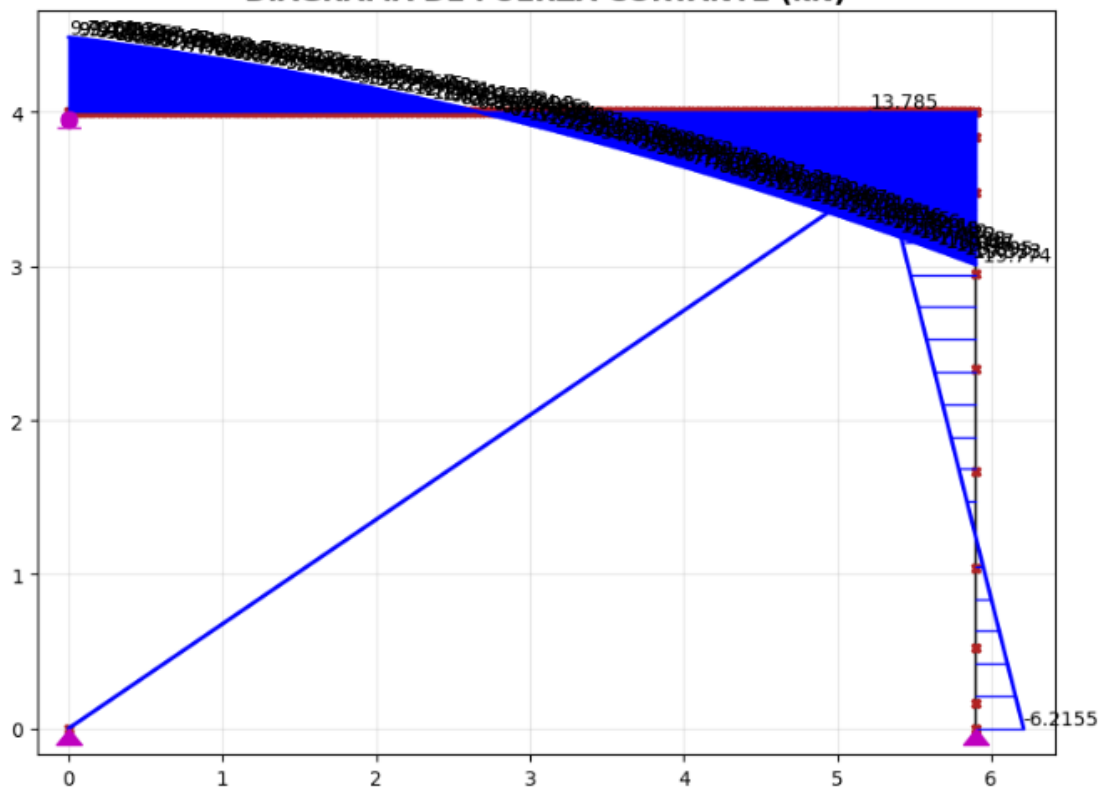
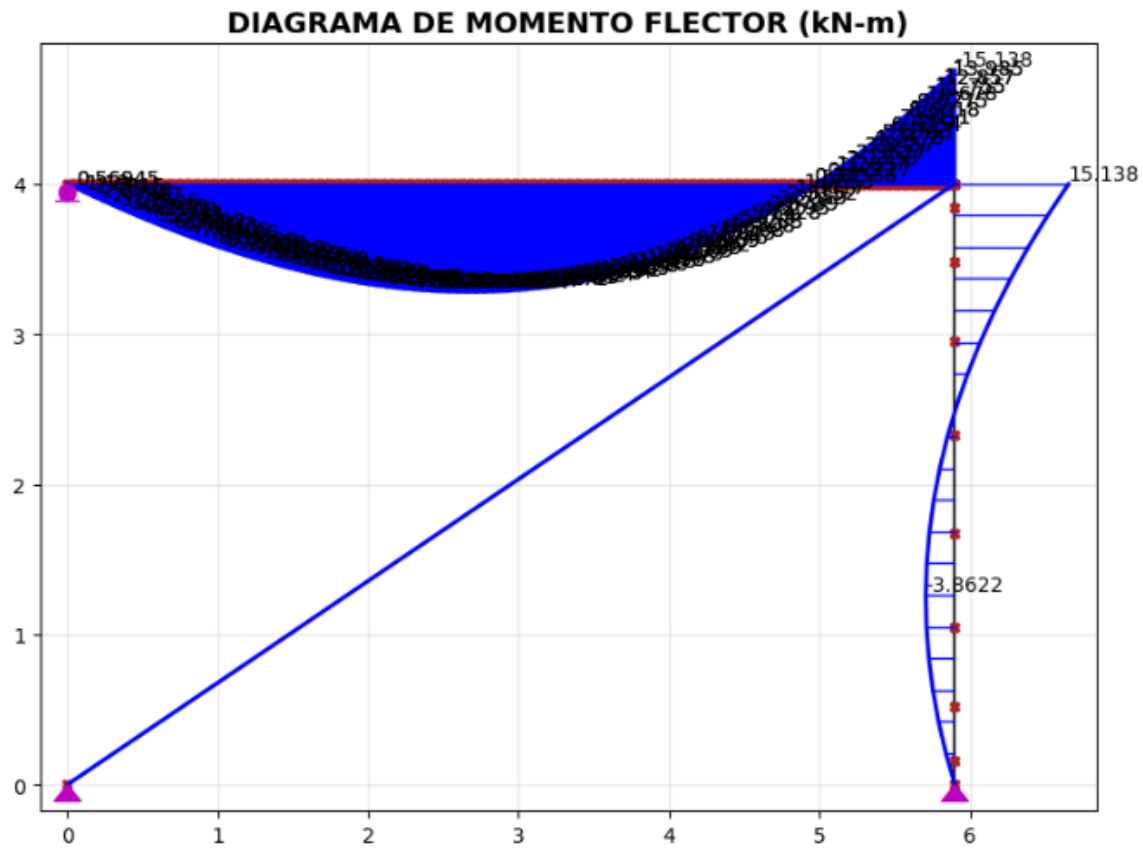


DIAGRAMA DE FUERZA CORTANTE (kN)





MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON SISTEMAS MIXTOS: EJERCICIO 2

```
import numpy as np
import math
import matplotlib.pyplot as plt

print("MÉTODO MATRICIAL DE RIGIDEZ PARA ESTRUCTURAS MIXTAS: EJERCICIO 2")

# -----
# DATOS DE ENTRADA
# -----
print("\n==== DATOS DE ENTRADA =====")
# --- Módulos de elasticidad (kN/m²) ---
Ec = 24870062.324 # Hormigón
Es = 2e8          # Acero estructural

# Vector de módulos: elementos 1-3 hormigón, elementos 4-10 acero
E = np.array([Ec, Ec, Ec, Es, Es, Es, Es, Es, Es, Es])

# --- Sección de COLUMNA ---
Acl = 0.3 * 0.3 # Área (m²) = base × altura
Icl = (1/12) * 0.3 * (0.3**3) # Inercia (m⁴) = (b·h³)/12 para sección rectangular

# --- Sección de VIGA ---
Av = 0.3 * 0.35 # Área (m²)
Iv = (1/12) * 0.3 * (0.35**3) # Inercia (m⁴)

# --- Sección de BARRAS (perfil tubular de acero) ---
Ab = (0.1**2) - ((0.1 - (2 * 0.004))**2) # Área neta (m²)
Ib = 1e-16 # Inercia despreciable (las barras solo trabajan a tracción/compresión)

# Vector de propiedades por elemento (10 elementos)
A = np.array([Acl, Av, Acl, Ab, Ab, Ab, Ab, Ab, Ab, Ab])
I = np.array([Icl, Iv, Icl, Ib, Ib, Ib, Ib, Ib, Ib, Ib])

# --- Longitudes de elementos (m) ---
Ld = math.sqrt(1**2 + 1**2) # Longitud diagonal m
L = np.array([3, 3, 3, 2, 2, 2, Ld, Ld, Ld, Ld])

# --- Ángulos de inclinación (°) ---
a = np.array([90, 0, 90, 0, 0, 0, -45, 45, -45, 45])

m = 10 # Número de elementos
n = 8 # Número de nodos
GL = n * 3 # Grados de libertad totales (24: 8 nodos × 3 GL: Dx, Dy, θz)
```


Impresión de datos de entrada (formato científico para valores pequeños)

```
np.set_printoptions(formatter={'float_kind': '{:.2e}'.format})
print(f"Módulo de elasticidad: {E} kN/m²")
print(f"Área sección transversal: {A} m²")
print(f"Momento de inercia: {I} m²")
np.set_printoptions(formatter=None) # Restaurar formato normal
print(f"Longitud de los elementos: {np.round(L, 2)} m")
print(f"Ángulo de inclinación de los elementos: {a} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")
```

-----
COORDENADAS DE LOS NODOS
-----

Matriz de coordenadas nodales en el plano X-Y (unidades: metros)
Organización: filas = nodos (1 a 8), columnas = coordenadas (x, y)

```
coordenadas_nodos = np.array([
    [0, 0],      # Nodo 1
    [0, 3],      # Nodo 2
    [3, 3],      # Nodo 3
    [0, 6],      # Nodo 4
    [2, 6],      # Nodo 5
    [4, 6],      # Nodo 6
    [1, 5],      # Nodo 7
    [3, 5]])     # Nodo 8
```

-----
CONECTIVIDAD DE LOS ELEMENTOS
-----

Cada fila: [nodo_inicial, nodo_final]

```
Elementos = [
    [1, 2],      # Elemento 1
    [2, 3],      # Elemento 2
    [2, 4],      # Elemento 3
    [4, 5],      # Elemento 4
    [5, 6],      # Elemento 5
    [7, 8],      # Elemento 6
    [4, 7],      # Elemento 7
    [7, 5],      # Elemento 8
    [5, 8],      # Elemento 9
    [8, 6]]      # Elemento 10
```

-----
GRADOS DE LIBERTAD - NUMERACIÓN AUTOMÁTICA
-----

```
print("\n==== GRADOS DE LIBERTAD =====")
Nx, Ny, Nz, Fx, Fy, Fz = [], [], [], [], [], []
```

```

for i in range(m):
    # Fórmula:  $GL = (nodo-1)*3 + 1, 2, 3$  para  $Dx, Dy, \partial z$ 
    Ni = Elementos[i][0] # Nodo inicial del elemento i
    Nf = Elementos[i][1] # Nodo final del elemento i

    Nx.append(Ni*3 - 2) # GL desplazamiento X - nodo inicial
    Ny.append(Ni*3 - 1) # GL desplazamiento Y - nodo inicial
    Nz.append(Ni*3)      # GL rotación Z - nodo inicial
    Fx.append(Nf*3 - 2) # GL desplazamiento X - nodo final
    Fy.append(Nf*3 - 1) # GL desplazamiento Y - nodo final
    Fz.append(Nf*3)      # GL rotación Z - nodo final

nombres = ['Nx:', 'Ny:', 'Nz:', 'Fx:', 'Fy:', 'Fz:']
datos = [Nx, Ny, Nz, Fx, Fy, Fz]
print(f"{'':6}", end="") # Espacio para los nombres
for elemento in range(len(datos[0])): # m = número de elementos
    print(f"E{elemento+1:<3d}", end="")
print()

# Imprimir cada fila de datos (tabla de grados de libertad)
for nombre, lista in zip(nombres, datos):
    print(f"{nombre:4}", end="") # Nombre de la fila alineado
    for valor in lista:
        print(f"{valor:4d}", end="")
    print()

# -----
# DEFINICIÓN DE CARGAS POR ELEMENTO
# -----
# Formato: [tipo, dirección, color, parámetro1, parámetro2]
# - Para 'Uniforme': [tipo, dirección, color, w_inicial, w_final]
# - Para 'Puntual': [tipo, dirección, color, magnitud, distancia_desde_inicio]
# - Para 'MomentoPuntual': [tipo, dirección, color, magnitud, distancia_desde_inicio]
# Nota: El parámetro 'extra' (posición 4) tiene diferente significado según el tipo
Cargas = [
    ['Uniforme', 'Local_y', 'olivedrab', -8, -8], # Elem 1: carga distribuida
    ['Puntual', 'Local_y', 'peru', -30, 1.5],      # Elem 2: carga puntual
    ['MomentoPuntual', 'Local_z', 'teal', 20, 1.5], # Elem 3: momento puntual
    [], [], [], [], [], [], [], []]              # Elem 4-10: sin carga

# -----
# PREPROCESAMIENTO GEOMÉTRICO DE TODOS LOS ELEMENTOS
# -----
# Calcula propiedades geométricas fundamentales para cada elemento:
geom = [] # Almacena en lista 'geom' para uso posterior en gráficas

for e in range(m):
    ni, nf = Elementos[e][0], Elementos[e][1]

```

```

xi, yi = coordenadas_nodos[ni-1] # Coordenadas nodo inicial (índice 0-based)
xf, yf = coordenadas_nodos[nf-1] # Coordenadas nodo final

dx = xf - xi # Diferencia en X
dy = yf - yi # Diferencia en Y
Le = math.sqrt((dx**2) + (dy**2)) # Longitud del elemento

# Vector director unitario (apunta del nodo inicial al final)
ex = dx / Le
ey = dy / Le

# Vector normal unitario (perpendicular, rotado 90° antihorario)
# Útil para visualización de diagramas perpendiculares al elemento
nx = -ey
ny = ex
geom.append([ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny])

# -----
# GRÁFICA SISTEMA ORIGINAL
# -----
plt.figure(figsize=(8, 6))
plt.title('SISTEMA ORIGINAL', fontsize=14, fontweight='bold')

# Dibujar elementos
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k-', linewidth=2, zorder=1)

# Dibujar nodos
for i, (x, y) in enumerate(coordenadas_nodos):
    plt.plot(x, y, 'o', color='orange', markersize=8, zorder=2)

# Dibujar apoyos con simbología específica
# Nodo 1: Apoyo EMPOTRADO (cuadrado) - restringe Dx, Dy, ̸z
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=2)

# Nodo 3: Apoyo FIJO (triángulo) - restringe Dx, Dy
plt.plot(3, 3-0.15, '^', color='maroon', markersize=20, zorder=2)

# Nodo 4: Apoyo FIJO INCLINADO (triángulo rotado) - restringe en dirección del apoyo
plt.scatter(-0.15, 6, marker=(3, 0, 270), s=700, color='maroon', zorder=2)
# Nodo 6: Apoyo FIJO (triángulo) - restringe Dx, Dy
plt.plot(4, 6-0.15, '^', color='maroon', markersize=20, zorder=2)

# --- Dibujo de cargas (solo visualización) ---
# Carga distribuida en columna izquierda (Elemento 1)
xi, yi = coordenadas_nodos[0] # Nodo 1
xf, yf = coordenadas_nodos[1] # Nodo 2

```

```
x_vals = np.linspace(xi, xf, 15)
y_vals = np.linspace(yi, yf, 15)

# Flechas horizontales representan carga distribuida horizontal
for x, y in zip(x_vals, y_vals):
    plt.arrow(x-0.8, y, 0.6, 0, head_width=0.15, head_length=0.2, fc='olivedrab',
              ec='olivedrab', zorder=4)

plt.plot([xi-0.8, xf-0.8], [yi, yf], '--', color='olivedrab', linewidth=2, zorder=3)
plt.text(-2, 1.5, '8 kN/m', color='olivedrab', fontsize=11, fontweight='bold',
         zorder=5)

# Carga puntual sobre el elemento 2 (viga inferior)
plt.arrow(1.5, 4.1, 0, -0.8, head_width=0.1, head_length=0.2, fc='peru', ec='peru',
         linewidth=3, zorder=4)
plt.text(1.6, 3.5, '-30 kN', color='peru', fontweight='bold', fontsize=11, zorder=5)

# Carga puntual en nodo 5 (nodo central superior)
plt.arrow(2, 7.1, 0, -0.8, head_width=0.1, head_length=0.2, fc='peru', ec='peru',
         linewidth=3, zorder=4)
plt.text(2.1, 6.5, '-30 kN', color='peru', fontweight='bold', fontsize=11, zorder=5)

# Momento puntual en elemento 3 (columna izquierda superior)
plt.text(-0.3, 4.2, "5", color="teal", fontsize=30, fontweight='bold', zorder=5)
plt.text(-1.5, 4.5, "20 kN·m", color="teal", fontsize=11, fontweight='bold',
         zorder=5)

# Configuración del gráfico
plt.xlabel('X [m]', fontsize=12)
plt.ylabel('Y [m]', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.tight_layout()
plt.show()

# -----
# ENSAMBLE DE LA MATRIZ DE RIGIDEZ GLOBAL
# -----
print("\n==== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL =====")
kG = np.zeros((GL, GL)) # Inicializar matriz de rigidez global (24x24) con ceros

# Listas para almacenar matrices de cada elemento (para post-procesamiento)
kL_elementos = [] # Matrices de rigidez Local (6x6) en coordenadas Locales
T_elementos = [] # Matrices de transformación (6x6)

for i in range(m):
    # Ángulo de transformación (conversión a radianes)
    theta = math.radians(a[i])
```

```

c = math.cos(theta)
s = math.sin(theta)

# Propiedades de rigidez del elemento
AE = A[i] * E[i] # Rigidez axial (EA) en kN
EI = E[i] * I[i] # Rigidez flexional (EI) en kN·m²
L2 = L[i] ** 2    # Longitud al cuadrado (m²)
L3 = L[i] ** 3    # Longitud al cubo (m³)

# Matriz de rigidez LOCAL para elemento viga-columna biempotrado (6×6)
# Para elementos con I≠0 (barras), los términos flexionales serán cercanos a cero
kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
      [0, (12*EI)/L3, (6*EI)/L2, 0, -(12*EI)/L3, (6*EI)/L2],
      [0, (6*EI)/L2, (4*EI)/L[i], 0, -(6*EI)/L2, (2*EI)/L[i]],
      [-AE/L[i], 0, 0, AE/L[i], 0, 0],
      [0, -(12*EI)/L3, -(6*EI)/L2, 0, (12*EI)/L3, -(6*EI)/L2],
      [0, (6*EI)/L2, (2*EI)/L[i], 0, -(6*EI)/L2, (4*EI)/L[i]]]
kL_elementos.append(kL)

# Matriz de transformación de coordenadas
T = [[c, s, 0, 0, 0, 0],
     [-s, c, 0, 0, 0, 0],
     [0, 0, 1, 0, 0, 0],
     [0, 0, 0, c, s, 0],
     [0, 0, 0, -s, c, 0],
     [0, 0, 0, 0, 0, 1]]
T_elementos.append(T)

# Transformar matriz LOCAL a GLOBAL: K_global = T^T · K_Local · T
# T^T es la matriz de transformación inversa (de Local a global)
T_T = np.transpose(T) # Transformación inversa (Local a global)
kg_e = np.matmul(np.matmul(T_T, kL), T) # Matriz de rigidez del elemento en
coordenadas globales (6×6)

# Ensamblar en matriz global - Ajuste por indexación Python (0-based)
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]

# Sumar contribución del elemento a la matriz global
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj] # Ensamblaje directo de rigidez

# Código comentado para depuración (útil para verificar cada elemento)
#print(f"ELEMENTO {i+1}")
#print(f"Longitud: {L[i]} m, Ángulo: {a[i]:.0f}°, Módulo de elasticidad: {E[i]:.2f} kN/m²")
#print(f"Área: {A[i]} m², Inercia: {I[i]:.3e} m⁴")
#print(f"Matriz global del elemento: \n {np.round(kg_e, 0)}\n")
#print(f"Matriz global del sistema después del elemento {i+1}: \n {np.round(kG, 0)}")

#print(f"Matriz global del sistema: \n {np.round(kG, 0)}")

```

```
# -----
# VECTOR DE FUERZAS DE EMPOTRAMIENTO PERFECTO (FEM)
# -----

print("\n=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===")
FEM_L_elementos = [] # FEM en coordenadas Locales por elemento
FEM_G = np.zeros(GL) # FEM en coordenadas globales (vector de 24 componentes)

for i in range(m):
    if Cargas[i]:

        # --- CASO 1: Carga uniforme/trapezoidal/triangular ---
        if Cargas[i][0] == 'Uniforme' and Cargas[i][1] == 'Local_y':
            # Convertir a magnitudes positivas (por idealización de ecuaciones)
            wi = -Cargas[i][3] # w inicial positiva hacia abajo
            wf = -Cargas[i][4] # w final positiva hacia abajo

            # Fórmulas analíticas para viga biempotrada con carga trapezoidal
            Ryi = ((7*wi*L[i])/20)+((3*wf*L[i])/20) # Reacción vertical en nodo inicial
            Mzi = ((wi*(L[i]**2))/20)+((wf*(L[i]**2))/30) # Momento en nodo inicial
            Ryf = ((3*wi*L[i])/20)+((7*wf*L[i])/20) # Reacción vertical en nodo final
            Mzf = -(((wi*(L[i]**2))/30) + ((wf*(L[i]**2))/20)) # Momento en nodo final

            FEM_L = [0, Ryi, Mzi, 0, Ryf, Mzf] # Vector FEM Local
            FEM_L_elementos.append(FEM_L)

        # --- CASO 2: Carga puntual a una distancia a del nodo inicial ---
        elif Cargas[i][0] == 'Puntual' and Cargas[i][1] == 'Local_y':
            P = -Cargas[i][3] # Magnitud de la carga (positiva hacia abajo)
            a = Cargas[i][4] # Distancia desde nodo inicial (m)
            b = L[i] - a # Distancia desde nodo final (m)

            # Fórmulas para viga biempotrada con carga puntual
            Ryi = ((P*(b**2))/(L[i]**2)) * (3-((2*b)/L[i])) # Reacción vertical en i
            Mzi = (P*a*(b**2))/(L[i]**2) # Momento en i
            Ryf = ((P*(a**2))/(L[i]**2)) * (3-((2*a)/L[i])) # Reacción vertical en f
            Mzf = -((P*(a**2)*b)/(L[i]**2)) # Momento en f (negativo)

            FEM_L = [0, Ryi, Mzi, 0, Ryf, Mzf]
            FEM_L_elementos.append(FEM_L)

        # --- CASO 3: Momento puntual a una distancia a del nodo inicial ---
        elif Cargas[i][0] == 'MomentoPuntual' and Cargas[i][1] == 'Local_z':
            M_mag = Cargas[i][3] # Magnitud del momento (kN-m)
            a = Cargas[i][4] # Distancia desde nodo inicial (m)
            b = L[i] - a # Distancia desde nodo final (m)

            # Fórmulas para viga biempotrada con momento puntual
            Ryi = (6*M_mag*a*b) / (L[i]**3) # Reacción vertical en i
            Mzi = (M_mag*b*((2*a)-b)) / (L[i]**2) # Momento en i
```

```

Ryf = -(6*M_mag*a*b) / (L[i]**3)      # Reacción vertical en f
Mzf = (M_mag*a*((2*b)-a)) / (L[i]**2) # Momento en f

FEM_L = [0, Ryi, Mzi, 0, Ryf, Mzf]
FEM_L_elementos.append(FEM_L)

else:
    # Elementos sin carga
    FEM_L = [0, 0, 0, 0, 0, 0]
    FEM_L_elementos.append(FEM_L)

# Transformar FEM Local a global: FEM_global = T^T * FEM_local
# T^T transforma de coordenadas locales a globales
FEM_Ge = np.matmul(np.transpose(T_elementos[i]), FEM_L)

# Ensamblar en vector global
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]
for jj, gl_j in enumerate(GL_elem):
    FEM_G[gl_j] += FEM_Ge[jj] # Sumar contribución del elemento

print(np.round(FEM_G, 2)) # Mostrar FEM global redondeado a 2 decimales

# -----
# CONDICIONES DE CONTORNO (RESTRICCIONES)
# -----
print("\n==== RESTRICCIONES =====")
# Vector de restricciones: 1 = restringido (desplazamiento conocido = 0)
#                               0 = libre (desplazamiento desconocido)
#                               GL: 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
restricciones = np.array([1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0,
                           21 22 23 24
                           0, 0, 0, 0])

# Estado de cada grado de libertad
for i in range(n):
    rtx = "[Restringido]" if restricciones[i*3] == 1 else "[Libre]"
    rty = "[Restringido]" if restricciones[i*3+1] == 1 else "[Libre]"
    rtz = "[Restringido]" if restricciones[i*3+2] == 1 else "[Libre]"
    print(f"Node {i+1}: Rx={restricciones[i*3]} {rtx}, Ry={restricciones[i*3+1]} {rty}, Rz={restricciones[i*3+2]} {rtz}")

# -----
# VECTOR DE FUERZAS EXTERNAS
# -----
print("\n==== VECTOR DE FUERZAS EXTERNAS =====")
# Fuerzas nodales aplicadas directamente (cargas puntuales en nodos)
#                               GL: 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
F=np.array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -30, 0, 0, 0, 0, 0, 0, 0, 0, 0])
print(f"Vector de fuerzas externas: {F} kN")

```



```
# -----
# REDUCCIÓN DEL SISTEMA
# -----
print("\n=== REDUCCIÓN DEL SISTEMA ===")
# Identificar grados de libertad activos (no restringidos)
GL_activos = np.where(restricciones == 0)[0] # Índices donde restricciones = 0
n_GL_activos = len(GL_activos)
print(f"Grados de libertad activos: {n_GL_activos}")
print(f"Índices: {GL_activos+1}") # +1 para mostrar numeración 1-based

# Extraer submatrices correspondientes a GL activos (partición de la matriz global)
kG_reducida = kG[np.ix_(GL_activos, GL_activos)] # Matriz de rigidez reducida
F_reducido = F[GL_activos] # Vector de fuerzas reducido
FEM_G_reducido = FEM_G[GL_activos] # Vector FEM reducido
print(f"\nMatriz global reducida:\n {np.round(kG_reducida, 0)}")
print(f"\nVector de fuerzas reducido: {F_reducido} kN")
print(f"\nVector FEM reducido: {np.round(FEM_G_reducido, 2)} kN")

# -----
# SOLUCIÓN DEL SISTEMA
# -----
print("\n\n===== SOLUCIÓN DEL SISTEMA =====")
# Calcular inversa de la matriz de rigidez global reducida
kG_r_inv = np.linalg.inv(kG_reducida)

# Calcular desplazamientos desconocidos:  $U = K^{-1} \cdot (F - FEM)$ 
# La ecuación de equilibrio:  $K \cdot U = F - FEM$  (Los FEM actúan como fuerzas equivalentes)
U_desconocidos = np.matmul(kG_r_inv, F_reducido - FEM_G_reducido)

# Reconstruir vector completo de desplazamientos (incluyendo ceros en GL restringidos)
U_totales = np.zeros(GL) # Inicializar con ceros
U_totales[GL_activos] = U_desconocidos # Asignar desplazamientos calculados a GL activos

# Calcular reacciones en apoyos:  $R = K \cdot U + FEM$ 
Reacciones = np.matmul(kG, U_totales) + FEM_G

# -----
# RESULTADOS
# -----
print("\n=== REACCIONES ===")
# Mostrar reacciones solo en nodos con restricciones
for i in range(n):
    Rx = Reacciones[i*3] # Reacción horizontal (kN)
    Ry = Reacciones[i*3+1] # Reacción vertical (kN)
    Mz = Reacciones[i*3+2] # Momento de reacción (kN-m)

    if np.any(restricciones[i*3:i*3+3] == 1):
        print(f"Node {i+1}: Rx = {Rx:.3f} kN, Ry = {Ry:.3f} kN, Mz = {Mz:.3f} kN-m")
```



```
print("\n=== DESPLAZAMIENTOS NODALES ===")
# Desplazamientos nodales (notación científica para valores pequeños)
for i in range(n):
    Ux = U_totales[i*3]      # Desplazamiento horizontal (m)
    Uy = U_totales[i*3+1]    # Desplazamiento vertical (m)
    Tz = U_totales[i*3+2]    # Rotación (radianes)
    print(f"Node {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m,  θz = {Tz:.3e} rad")

# -----
# GRÁFICA SISTEMA DEFORMADO
# -----
FS = 1000 # Factor de escala para visualización (amplifica desplazamientos para ver
deformación)
plt.figure(figsize=(6, 5))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar sistema original (líneas punteadas grises) como referencia
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '--', color='grey', linewidth=2, alpha=0.5)

# Dibujar sistema deformado (líneas verdes) con desplazamientos amplificados
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    # Desplazamientos de los nodos (del vector solución)
    desp_i = np.array([U_totales[(ni-1)*3], U_totales[(ni-1)*3+1], U_totales[(ni-
1)*3+2]])
    desp_f = np.array([U_totales[(nf-1)*3], U_totales[(nf-1)*3+1], U_totales[(nf-
1)*3+2]])

    # Coordenadas deformadas (original + desplazamiento × factor de escala)
    xi_def = xi + desp_i[0] * FS
    yi_def = yi + desp_i[1] * FS
    xf_def = xf + desp_f[0] * FS
    yf_def = yf + desp_f[1] * FS

    # Dibujar elemento deformado
    plt.plot([xi_def, xf_def], [yi_def, yf_def], '-', color='g', linewidth=2)

# Dibujar apoyos en posición original (referencia)
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=2) # Apoyo empotrado
plt.plot(3, 3-0.15, '^', color='maroon', markersize=20, zorder=2) # Apoyo fijo
plt.scatter(-0.15, 6, marker=(3, 0, 270), s=700, color='maroon', zorder=2) # Apoyo fijo
plt.plot(4, 6-0.15, '^', color='maroon', markersize=20, zorder=2) # Apoyo fijo

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.grid(True, alpha=0.5)
```

```
plt.axis('equal')
plt.tight_layout()
plt.show()

print("\n=== FUERZAS INTERNAS ===")
F_int_elementos = [] # Lista para almacenar las fuerzas internas de cada elemento

for i in range(m):
    # Extraer desplazamientos globales del elemento (del vector U_totales)
    Ue = np.array([
        U_totales[Nx[i]-1], # Dx nodo inicial
        U_totales[Ny[i]-1], # Dy nodo inicial
        U_totales[Nz[i]-1], # Dz nodo inicial
        U_totales[Fx[i]-1], # Dx nodo final
        U_totales[Fy[i]-1], # Dy nodo final
        U_totales[Fz[i]-1]] # Dz nodo final
    )
    # Transformar desplazamientos a coordenadas locales: U_local = T · U_global
    Ue_L = np.matmul(T_elementos[i], Ue)

    # Calcular fuerzas internas en coordenadas locales: F = K_local · U_local + FEM_local
    Fe_L = np.matmul(kL_elementos[i], Ue_L) + FEM_L_elementos[i]
    F_int_elementos.append(Fe_L)

    # Presentar resultados
    print(f"Elemento {i+1}:")
    print(f"  Nodo {Elementos[i][0]}: N = {Fe_L[0]:.3f} kN, V = {Fe_L[1]:.3f} kN, M = {Fe_L[2]:.3f} kN-m")
    print(f"  Nodo {Elementos[i][1]}: N = {Fe_L[3]:.3f} kN, V = {Fe_L[4]:.3f} kN, M = {Fe_L[5]:.3f} kN-m\n")

# -----
# DIAGRAMA AXIAL
# -----
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA AXIAL", fontsize=14, fontweight="bold")
escala_axial = 0.008 # Factor de escala para visualización

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=3) # Línea del elemento (negro)

    # Extraer fuerzas axiales en extremos (del cálculo de fuerzas internas)
    Ni = F_int_elementos[i][0] # Fuerza axial en nodo inicial (kN)
    Nf = F_int_elementos[i][3] # Fuerza axial en nodo final (kN)

    # Puntos para dibujar el prisma del diagrama axial
    # El diagrama se dibuja perpendicular al elemento (dirección del vector normal)
    P1 = np.array([xi, yi]) # Punto base inicial (sobre el elemento)
    P2 = P1 + escala_axial * Ni * np.array([nx, ny]) # Despl. perpendicular según Ni
```

```
P3 = P2 + Le * np.array([ex, ey]) # Avanzar a lo largo del elemento
P4 = P3 + escala_axial * Nf * np.array([nx, ny]) # Despl. perpendicular según Nf

# Dibujar relleno y contorno del diagrama
# fill() dibuja un polígono relleno, plot() dibuja el contorno superior
plt.fill([P1[0], P2[0], P3[0], P4[0]], [P1[1], P2[1], P3[1], P4[1]], alpha=0.3)
plt.plot([P2[0], P3[0]], [P2[1], P3[1]])

# Texto en punto medio indicando magnitud y tipo de esfuerzo
xm = xi + ex*Le/2 # Coordenada X del punto medio
ym = yi + ey*Le/2 # Coordenada Y del punto medio
tipo = "Nula" if abs(Nf)<1e-6 else "Tracción" if Nf>0 else "Compresión" if Nf<0
      else "" # Determinar tipo de esfuerzo
plt.text(xm, ym, f"{Nf:.2f} kN\n({tipo})", fontsize=8, ha='center',
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()

# -----
# DIAGRAMA CORTANTE
# -----
escala_cortante = 0.05
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA CORTANTE", fontsize=14, fontweight="bold")

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Extraer Vi de fuerzas internas
    Vi = F_int_elementos[i][1] # Cortante en nodo inicial (kN)
    Vf = F_int_elementos[i][4] # Cortante en nodo final (kN)

    x = np.linspace(0, Le, 100) # Puntos de evaluación

    # Verificar si el elemento tiene carga
    if len(Cargas[i]) > 0:
        tipo, direccion, color, w1, extra = Cargas[i]

        if tipo == 'Uniforme' and direccion == 'Local_y':
            wi = w1
            wf = extra
            # Carga trapezoidal:  $V(x) = Vi + wi \cdot x + ((wf-wi)/(2L)) \cdot x^2$ 
            V = Vi + wi*x + ((wf - wi)/(2*Le)) * x**2

        elif tipo == 'Puntual' and direccion == 'Local_y':
            P = w1 # Magnitud de la carga puntual (kN)
            a = extra # Distancia desde nodo inicial (m)
```

```
# DEFINICIÓN POR TRAMOS CON np.piecewise
# Condiciones: [x <= a, x > a]
# Funciones: [constante = Vi, constante = Vi + P]
V = np.piecewise(x,
                 [x <= a, x > a],      # Condiciones
                 [Vi, Vi + P])        # Valores constantes

# Agregar punto exacto en 'a' para mejor visualización
# Esto asegura que se vea la discontinuidad vertical
x = np.concatenate([x, [a, a + 1e-6]]) # Punto antes y después
V = np.concatenate([V, [Vi, Vi + P]])

# Reordenar para mantener consistencia
idx = np.argsort(x)
x = x[idx]
V = V[idx]

else:
    # Tipo de carga no implementado
    V = Vi * np.ones_like(x) # Constante (array del tamaño de x)

else:
    # Sin carga
    V = Vi * np.ones_like(x) # Constante

# Coordenadas del diagrama (desplazamiento perpendicular)
X_diag = xi + ex*x + escala_cortante * V * nx
Y_diag = yi + ey*x + escala_cortante * V * ny
X_base = xi + ex*x
Y_base = yi + ey*x

# Dibujar diagrama con relleno
plt.plot(X_diag, Y_diag, linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:-1]]),
         np.concatenate([Y_base, Y_diag[:-1]]), alpha=0.4)

# Etiquetas
plt.text(X_diag[0], Y_diag[0], f"{Vi:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.text(X_diag[-1], Y_diag[-1], f"{-Vf:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()
```

```
# -----
# DIAGRAMA MOMENTO
# -----
escala_momento = 0.05
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA MOMENTO", fontsize=14, fontweight="bold")

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2)

    # Extraer valores necesarios
    Vi = F_int_elementos[i][1] # Cortante inicial (kN)
    Mi = F_int_elementos[i][2] # Momento inicial (kN-m)
    Mf = F_int_elementos[i][5] # Momento final (kN-m)
    x = np.linspace(0, Le, 100)

    if len(Cargas[i]) > 0:
        tipo, direccion, color, w1, extra = Cargas[i]

        if tipo == 'Uniforme' and direccion == 'Local_y':
            wi = w1
            wf = extra

            # Momento para carga trapezoidal
            #  $M(x) = -M_i + V_i \cdot x + (w_i \cdot x^2)/2 + ((w_f - w_i)/(6L)) \cdot x^3$ 
            M = -Mi + Vi*x + (wi * x**2)/2 + ((wf - wi)/(6*Le)) * x**3

        elif tipo == 'Puntual' and direccion == 'Local_y':
            P = w1
            a = extra

            # MOMENTO CON CARGA PUNTUAL
            # PRIMERO: calcular con el x original
            M_original = np.piecewise(x,
                                      [x <= a, x > a],
                                      [lambda x: -Mi + Vi*x,
                                       lambda x: -Mi + Vi*x + P*(x - a)])

            # AGREGAR PUNTOS EXTRA alrededor de 'a' (para visualización)
            x_extra = np.array([a - 1e-6, a, a + 1e-6])
            M_extra = np.array([
                -Mi + Vi*(a - 1e-6), # Valor justo antes
                -Mi + Vi*a,          # Valor en a
                -Mi + Vi*a + P*(a - a)]) # = -Mi + Vi*a (igual, es continuo)

            # Concatenar y ordenar
            x = np.concatenate([x, x_extra])
            M = np.concatenate([M_original, M_extra])
            idx = np.argsort(x)
```

```

x = x[idx]
M = M[idx]

elif tipo == 'MomentoPuntual' and direccion == 'Local_z':
    M_puntual = w1
    a = extra

    # MOMENTO CON MOMENTO PUNTUAL
    M_original = np.piecewise(x,
                              [x < a, x >= a],
                              [lambda x: -Mi + Vi*x,
                               lambda x: -Mi + Vi*x - M_puntual])

    # Puntos extra para visualizar discontinuidad
    x_extra = np.array([a - 1e-6, a, a + 1e-6])
    M_extra = np.array([
        -Mi + Vi*(a - 1e-6),
        -Mi + Vi*a,
        -Mi + Vi*a - M_puntual])

    x = np.concatenate([x, x_extra])
    M = np.concatenate([M_original, M_extra])
    idx = np.argsort(x)
    x = x[idx]
    M = M[idx]

else:
    M = -Mi + Vi*x
else:
    M = -Mi + Vi*x

# Coordenadas del diagrama
X_diag = xi + ex*x + escala_momento * M * nx
Y_diag = yi + ey*x + escala_momento * M * ny
X_base = xi + ex*x
Y_base = yi + ey*x

# Dibujar
plt.plot(X_diag, Y_diag, linewidth=1.5)
plt.fill(np.concatenate([X_base, X_diag[:, -1]]),
         np.concatenate([Y_base, Y_diag[:, -1]]), alpha=0.4)

# Etiqueta en extremo inicial
plt.text(X_diag[0], Y_diag[0], f"{-Mi:.2f}", fontsize=8,
        bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

# Etiquetas en puntos especiales (cargas puntuales y momentos puntuales)
if len(Cargas[i]) > 0:
    tipo, direccion, color, w1, extra = Cargas[i]

```

```

if tipo in ['Puntual', 'MomentoPuntual']:
    a = extra

    # Buscar TODOS los puntos cercanos a 'a'
    mascara_cerca = np.abs(x - a) < 1e-4

    if np.any(mascara_cerca):

        for idx in np.where(mascara_cerca)[0]:
            plt.text(xi + ex*x[idx] + escala_momento * M[idx] * nx,
                    yi + ey*x[idx] + escala_momento * M[idx] * ny,
                    f"{M[idx]:.2f}", fontsize=7, bbox=dict(facecolor='white',
                    alpha=0.9, edgecolor='none'))

    # Etiqueta en extremo final
    plt.text(X_diag[-1], Y_diag[-1], f"{Mf:.2f}", fontsize=8,
            bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()

```

MÉTODO MATRICIAL DE RIGIDEZ PARA ESTRUCTURAS MIXTAS: EJERCICIO 2

===== DATOS DE ENTRADA =====

Módulo de elasticidad: [2.49e+07 2.49e+07 2.49e+07 2.00e+08 2.00e+08 2.00e+08
2.00e+08 2.00e+08 2.00e+08 2.00e+08] kN/m²

Área sección transversal: [9.00e-02 1.05e-01 9.00e-02 1.54e-03 1.54e-03 1.54e-03
1.54e-03 1.54e-03 1.54e-03 1.54e-03] m²

Momento de inercia: [6.75e-04 1.07e-03 6.75e-04 1.00e-16 1.00e-16 1.00e-16 1.00e-16
1.00e-16 1.00e-16 1.00e-16] m⁴

Longitud de los elementos: [3. 3. 3. 2. 2. 2. 1.41 1.41 1.41 1.41] m

Ángulo de inclinación de los elementos: [90 0 90 0 0 0 -45 45 -45 45] °

Número de elementos: 10

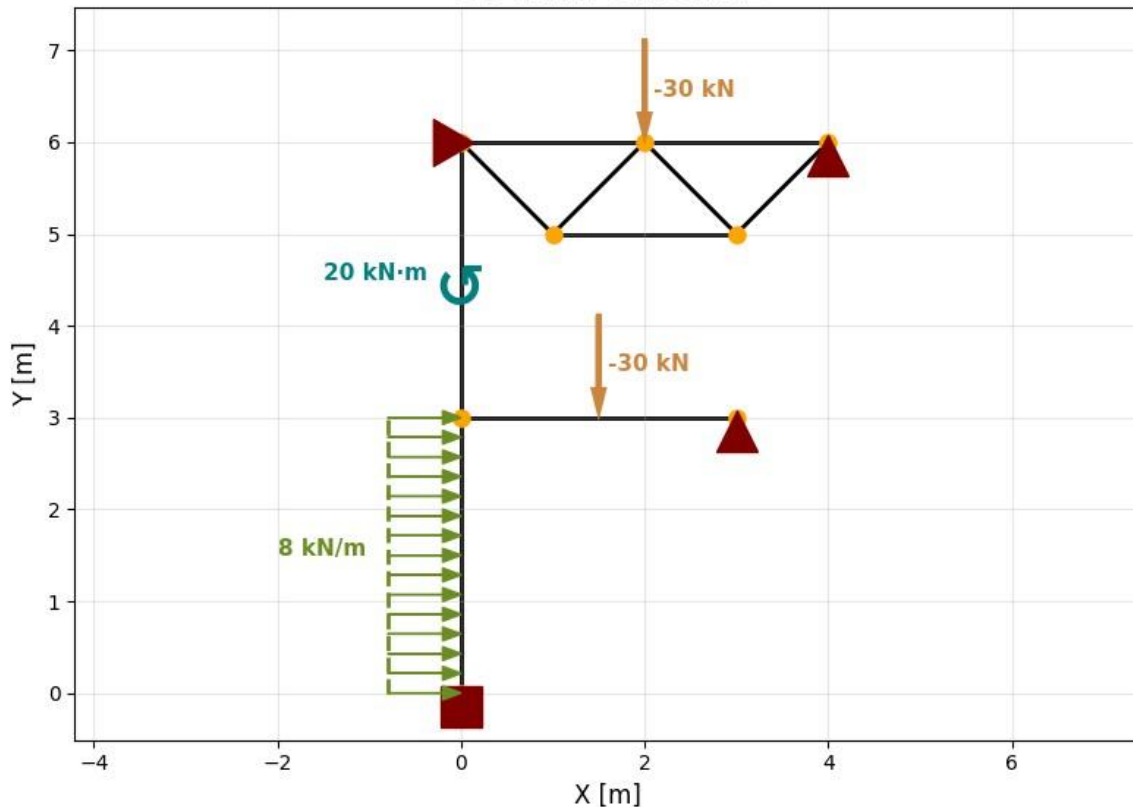
Número de nodos: 8

Número de grados de libertad: 24

===== GRADOS DE LIBERTAD =====

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10
Nx:	1	4	4	10	13	19	10	19	13	22
Ny:	2	5	5	11	14	20	11	20	14	23
Nz:	3	6	6	12	15	21	12	21	15	24
Fx:	4	7	10	13	16	22	19	13	22	16
Fy:	5	8	11	14	17	23	20	14	23	17
Fz:	6	9	12	15	18	24	21	15	24	18

SISTEMA ORIGINAL



===== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL =====

=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ===

```
[ -12.    0.    6.   -22.   15.   10.25  0.    15.   -11.25  10.   -0.    5.
  0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0.    0. ]
```

===== RESTRICCIONES =====

Nodo 1: Rx=1 [Restringido], Ry=1 [Restringido], Rz=1 [Restringido]

Nodo 2: Rx=0 [Libre], Ry=0 [Libre], Rz=0 [Libre]

Nodo 3: Rx=1 [Restringido], Ry=1 [Restringido], Rz=0 [Libre]

Nodo 4: Rx=1 [Restringido], Ry=1 [Restringido], Rz=0 [Libre]

Nodo 5: Rx=0 [Libre], Ry=0 [Libre], Rz=0 [Libre]

Nodo 6: Rx=1 [Restringido], Ry=1 [Restringido], Rz=0 [Libre]

Nodo 7: Rx=0 [Libre], Ry=0 [Libre], Rz=0 [Libre]

Nodo 8: Rx=0 [Libre], Ry=0 [Libre], Rz=0 [Libre]

===== VECTOR DE FUERZAS EXTERNAS =====

Vector de fuerzas externas: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 -30
0 0 0 0 0 0 0 0 0 0 0] kN

=== REDUCCIÓN DEL SISTEMA ===

Grados de libertad activos: 15

Índices: [4 5 6 9 12 13 14 15 18 19 20 21 22 23 24]

Matriz global reducida:

```
[ [ 885374.    0.    0.    0.  -11192.    0.    0.    0.
    0.    0.    0.    0.    0.    0.    0. ]
 [    0. 1504052. 17772. 17772.    0.    0.    0.    0.
    0.    0.    0.    0.    0.    0.    0. ]
 [    0. 17772. 80310. 17772. 11192.    0.    0.    0.
    0.    0.    0.    0.    0.    0.    0. ]
 [    0. 17772. 17772. 35543.    0.    0.    0.    0.
    0.    0.    0.    0.    0.    0.    0. ]
 [ -11192.    0. 11192.    0. 22383.    0.   -0.    0.
    0.   -0.   -0.    0.    0.    0.    0. ]
 [    0.    0.    0.    0.    0. 524423.    0.    0.
    0. -108612. -108612.    0. -108612. 108612.    0. ]
 [    0.    0.    0.    0.   -0.    0. 217223.    0.
    0. -108612. -108612.   -0. 108612. -108612.    0. ]
 [    0.    0.    0.    0.    0.    0.    0.    0.
    0.   -0.    0.    0.   -0.   -0.    0. ]
 [    0.    0.    0.    0.    0.    0.    0.    0.
    0.    0.    0.    0.   -0.    0.    0. ]
 [    0.    0.    0.    0.   -0. -108612. -108612.   -0.
    0. 370823.    0.   -0. -153600.    0.    0. ]
 [    0.    0.    0.    0.   -0. -108612. -108612.    0.
    0.    0. 217223.    0.    0.   -0.    0. ]
 [    0.    0.    0.    0.    0.    0.    0.   -0.
    0.   -0.    0.    0.    0.   -0.    0. ]
```

```
[ 0. 0. 0. 0. 0. 0. -108612. 108612. -0. -0.
-153600. 0. 0. 370823. 0. -0.]
[ 0. 0. 0. 0. 0. 0. 108612. -108612. -0.
0. 0. -0. -0. 0. 217223. -0.]
[ 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. -0. -0. 0.]]
Vector de fuerzas reducido: [ 0 0 0 0 0 0 -30 0 0 0 0 0 0 0 0] kN
Vector FEM reducido: [-22. 15. 10.25 -11.25 5. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0.] kN
```

===== SOLUCIÓN DEL SISTEMA =====

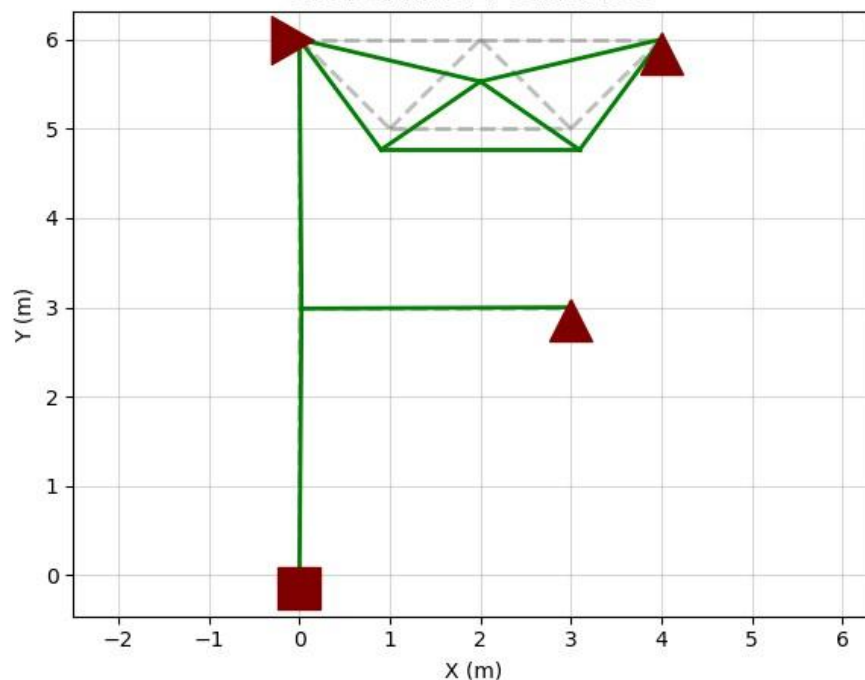
=== REACCIONES ===

Nodo 1: Rx = -9.898 kN, Ry = 9.390 kN, Mz = 3.985 kN-m
Nodo 3: Rx = -20.420 kN, Ry = 11.220 kN, Mz = 0.000 kN-m
Nodo 4: Rx = -8.682 kN, Ry = 24.390 kN, Mz = 0.000 kN-m
Nodo 6: Rx = 15.000 kN, Ry = 15.000 kN, Mz = -0.000 kN-m

=== DESPLAZAMIENTOS NODALES ===

Nodo 1: Ux = 0.000e+00 m, Uy = 0.000e+00 m, θz = 0.000e+00 rad
Nodo 2: Ux = 2.346e-05 m, Uy = -1.259e-05 m, θz = -2.035e-04 rad
Nodo 3: Ux = 0.000e+00 m, Uy = 0.000e+00 m, θz = 4.245e-04 rad
Nodo 4: Ux = 0.000e+00 m, Uy = 0.000e+00 m, θz = -1.099e-04 rad
Nodo 5: Ux = 2.390e-17 m, Uy = -4.715e-04 m, θz = -1.146e-05 rad
Nodo 6: Ux = 0.000e+00 m, Uy = 0.000e+00 m, θz = 2.469e-04 rad
Nodo 7: Ux = -9.766e-05 m, Uy = -2.358e-04 m, θz = -1.839e-04 rad
Nodo 8: Ux = 9.766e-05 m, Uy = -2.358e-04 m, θz = 1.653e-04 rad

SISTEMA DEFORMADO



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 9.390 \text{ kN}$, $V = 9.898 \text{ kN}$, $M = 3.985 \text{ kN-m}$

Nodo 2: $N = -9.390 \text{ kN}$, $V = 14.102 \text{ kN}$, $M = -10.292 \text{ kN-m}$

Elemento 2:

Nodo 2: $N = 20.420 \text{ kN}$, $V = 18.780 \text{ kN}$, $M = 11.339 \text{ kN-m}$

Nodo 3: $N = -20.420 \text{ kN}$, $V = 11.220 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 3:

Nodo 2: $N = -9.390 \text{ kN}$, $V = 6.318 \text{ kN}$, $M = -1.047 \text{ kN-m}$

Nodo 4: $N = 9.390 \text{ kN}$, $V = -6.318 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 4:

Nodo 4: $N = -0.000 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 5: $N = 0.000 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 5:

Nodo 5: $N = 0.000 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 6: $N = -0.000 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 6:

Nodo 7: $N = -30.000 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 8: $N = 30.000 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 7:

Nodo 4: $N = -21.213 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 7: $N = 21.213 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 8:

Nodo 7: $N = 21.213 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 5: $N = -21.213 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 9:

Nodo 5: $N = 21.213 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 8: $N = -21.213 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 10:

Nodo 8: $N = -21.213 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 6: $N = 21.213 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

DIAGRAMA AXIAL

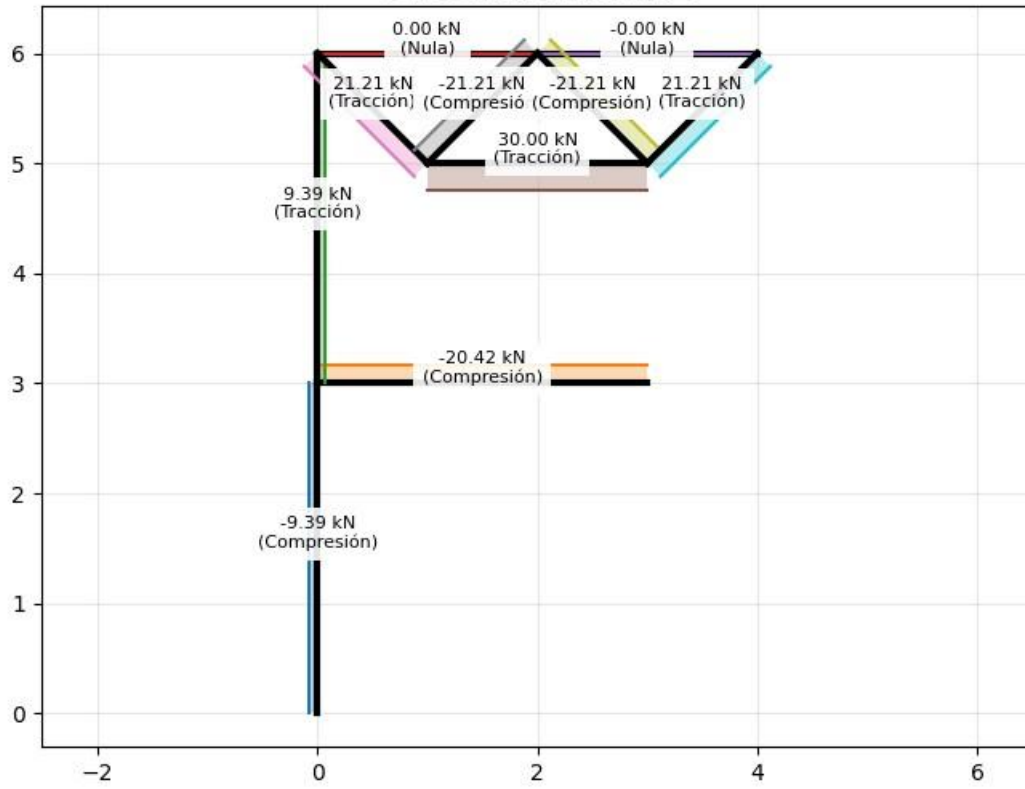


DIAGRAMA CORTANTE

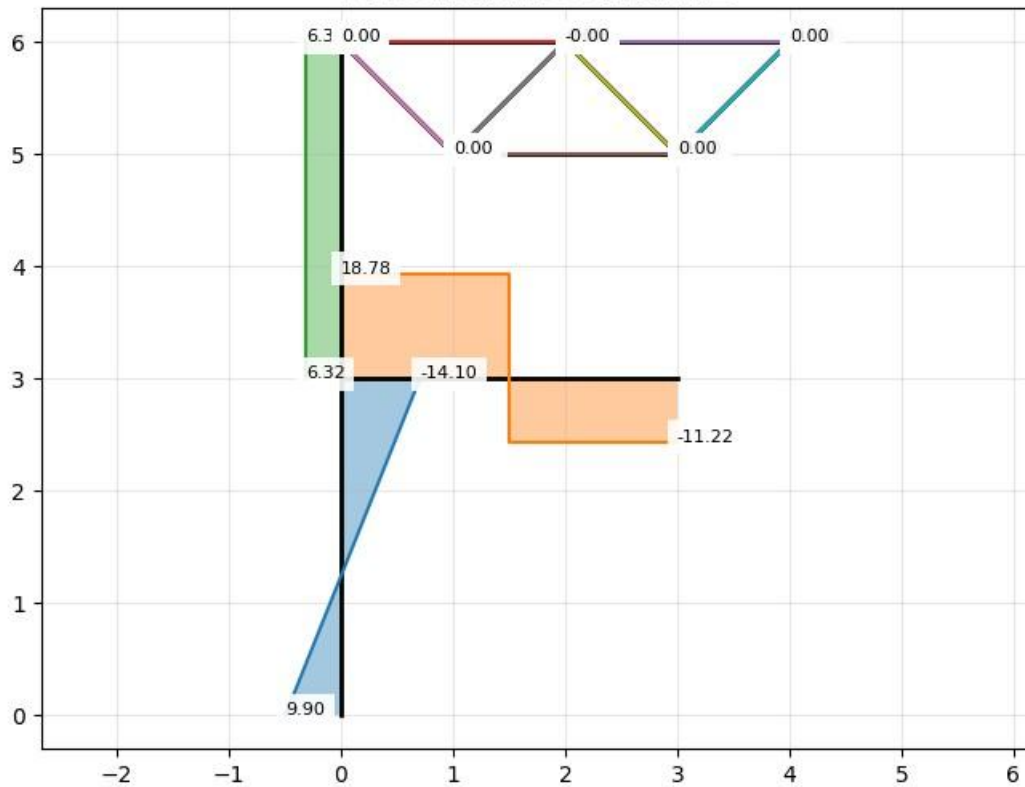
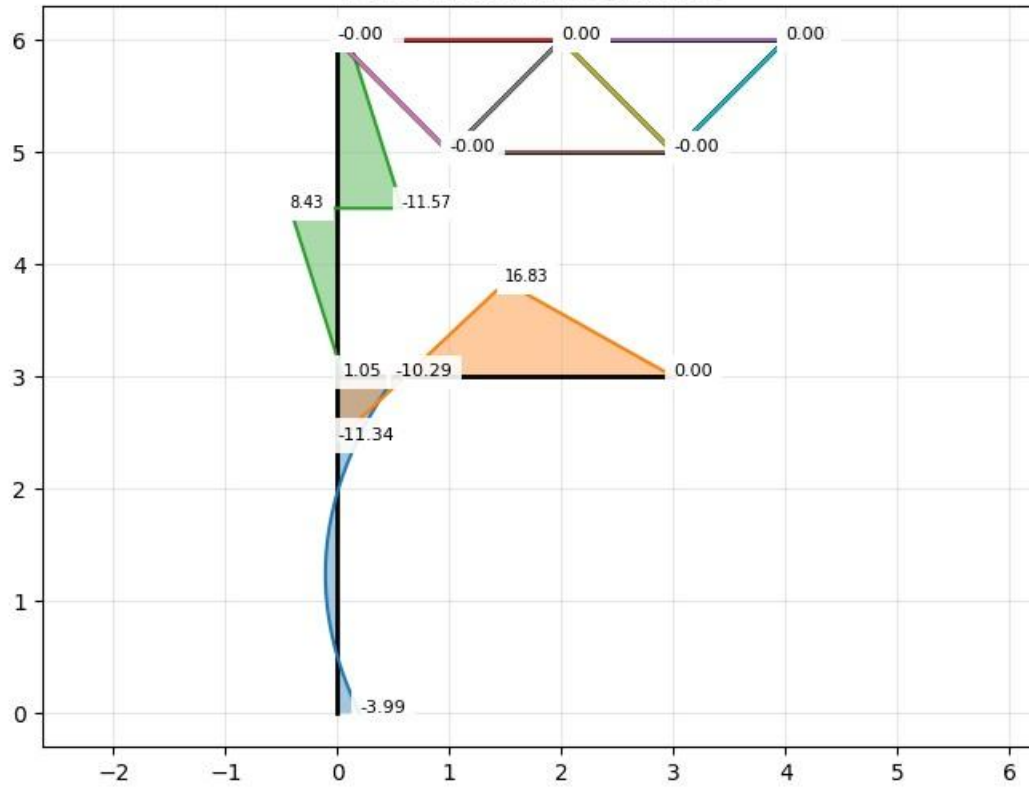


DIAGRAMA MOMENTO



MODELO OPENSEES

SISTEMAS MIXTOS: EJERCICIO 2

```
# -----
# IMPORTACIÓN DE LIBRERÍAS
# -----
import openseespy.opensees as ops # Biblioteca principal para análisis estructural
import opsvis as opsv            # Extensión para visualización de resultados
import numpy as np               # Para operaciones numéricas y arreglos
import math                      # Para funciones matemáticas básicas (atan, sqrt, etc.)
import matplotlib.pyplot as plt  # Para visualización personalizada de resultados

print("MODELO EN OPENSEES DEL EJERCICIO 2 - ESTRUCTURAS MIXTAS")

# -----
# INICIALIZACIÓN DEL MODELO
# -----
ops.wipe() # Limpiar memoria de análisis previos (eliminar modelos anteriores)

# Modelo 2D con 3 grados de libertad por nodo (dx, dy,  $\partial z$ )
# -ndm: número de dimensiones espaciales (2: plano XY)
# -ndf: número de grados de libertad por nodo (3: desplazamientos X, Y y rotación Z)
ops.model("Basic", "-ndm", 2, "-ndf", 3)

# -----
# DEFINICIÓN DE NODOS
# -----
# Creación de nodos, distancias en metros
ops.node(1, 0.0, 0.0) # Nodo 1
ops.node(2, 0.0, 3.0) # Nodo 2
ops.node(3, 3.0, 3.0) # Nodo 3
ops.node(4, 0.0, 6.0) # Nodo 4
ops.node(5, 2.0, 6.0) # Nodo 5
ops.node(6, 4.0, 6.0) # Nodo 6
ops.node(7, 1.0, 5.0) # Nodo 7
ops.node(8, 3.0, 5.0) # Nodo 8
ops.node(9, 0.0, 4.5) # Nodo 9

# -----
# CONDICIONES DE APOYO
# -----
ops.fix(1, 1, 1, 1) # Nodo 1: Empotramiento (Dx, Dy,  $\partial z$ ) - todos los GL fijos
ops.fix(3, 1, 1, 0) # Nodo 3: Apoyo fijo (Dx, Dy) - rotación Libre ( $\partial z=0$ )
ops.fix(4, 1, 1, 0) # Nodo 4: Apoyo fijo (Dx, Dy) - rotación Libre ( $\partial z=0$ )
ops.fix(6, 1, 1, 0) # Nodo 6: Apoyo fijo (Dx, Dy) - rotación Libre ( $\partial z=0$ )
```

```
# -----
# DEFINICIÓN DE MATERIALES
# -----
Ec = 24870062.32      # Módulo de elasticidad del hormigón (kN/m²)
Es = 2e8              # Módulo de elasticidad del acero (kN/m²)

# -----
# DEFINICIÓN DE SECCIONES
# -----
# --- Sección de VIGA ---
# Viga de hormigón de 0.3m x 0.35m (sección rectangular)
Av = 0.105            # Área (m²)
Iv = 0.001071875      # Inercia (m⁴)
ops.section('Elastic', 1, Ec, Av, Iv)

# --- Sección de COLUMNA ---
# Columna de hormigón de 0.3m x 0.3m (sección cuadrada)
Ac = 0.09             # Área (m²)
Ic = 0.000675         # Inercia (m⁴)
ops.section('Elastic', 2, Ec, Ac, Ic)

# --- Sección de BARRA ---
# Perfil tubular cuadrado de 100mm de lado y espesor 4mm
Ab = 1.536e-3         # Área neta (m²)
Ib = 1e-16            # Inercia despreciable (las barras solo trabajan axialmente)
ops.section('Elastic', 3, Es, Ab, Ib)

# -----
# TRANSFORMACIÓN GEOMÉTRICA
# -----
# Transformación lineal para elementos viga-columna
# Define la orientación y comportamiento geométrico de los elementos
ops.geomTransf("Linear", 1) # Transformación única para todos los elementos

# -----
# ESQUEMA DE INTEGRACIÓN
# -----
# Integración de Lobatto a lo largo del elemento para elementos viga-columna
# Parámetros: ('Lobatto', tag_integración, tag_sección, puntos_integración)
ops.beamIntegration('Lobatto', 1, 1, 10) # Para vigas (10 puntos - alta precisión)
ops.beamIntegration('Lobatto', 2, 2, 10) # Para columnas (10 puntos - alta
precisión)
ops.beamIntegration('Lobatto', 3, 3, 2)  # Para barras (2 puntos - suficiente para
comportamiento axial)
```

```
# -----  
# ELEMENTOS  
# -----  
Elementos = [] # Lista para post-procesamiento (almacena información de cada elemento)  
  
# Elemento 1: Columna (nodo 1 → 2)  
ops.element('dispBeamColumn', 1, 1, 2, 1, 2) # transfTag=1, beamIntTag=2  
Elementos.append({"ID": 1, "Nodo_i": 1, "Nodo_j": 2})  
  
# Elemento 2: Viga (nodo 2 → 3)  
ops.element('dispBeamColumn', 2, 2, 3, 1, 1) # transfTag=1, beamIntTag=1  
Elementos.append({"ID": 2, "Nodo_i": 2, "Nodo_j": 3})  
  
# Elemento 3: Columna (nodo 2 → 9)  
ops.element('dispBeamColumn', 3, 2, 9, 1, 2)  
Elementos.append({"ID": 3, "Nodo_i": 2, "Nodo_j": 9})  
  
# Elemento 4: Columna (nodo 9 → 4)  
ops.element('dispBeamColumn', 4, 9, 4, 1, 2)  
Elementos.append({"ID": 4, "Nodo_i": 9, "Nodo_j": 4})  
  
# Elemento 5: Barra (nodo 4 → 5)  
ops.element('dispBeamColumn', 5, 4, 5, 1, 3) # transfTag=1, beamIntTag=3  
Elementos.append({"ID": 5, "Nodo_i": 4, "Nodo_j": 5})  
  
# Elemento 6: Barra (nodo 5 → 6)  
ops.element('dispBeamColumn', 6, 5, 6, 1, 3)  
Elementos.append({"ID": 6, "Nodo_i": 5, "Nodo_j": 6})  
  
# Elemento 7: Barra (nodo 7 → 8)  
ops.element('dispBeamColumn', 7, 7, 8, 1, 3)  
Elementos.append({"ID": 7, "Nodo_i": 7, "Nodo_j": 8})  
  
# Elemento 8: Barra (nodo 4 → 7)  
ops.element('dispBeamColumn', 8, 4, 7, 1, 3)  
Elementos.append({"ID": 8, "Nodo_i": 4, "Nodo_j": 7})  
  
# Elemento 9: Barra (nodo 7 → 5)  
ops.element('dispBeamColumn', 9, 7, 5, 1, 3)  
Elementos.append({"ID": 9, "Nodo_i": 7, "Nodo_j": 5})  
  
# Elemento 10: Barra (nodo 5 → 8)  
ops.element('dispBeamColumn', 10, 5, 8, 1, 3)  
Elementos.append({"ID": 10, "Nodo_i": 5, "Nodo_j": 8})  
  
# Elemento 11: Barra (nodo 8 → 6)  
ops.element('dispBeamColumn', 11, 8, 6, 1, 3)  
Elementos.append({"ID": 11, "Nodo_i": 8, "Nodo_j": 6})
```



```
# -----
# DEFINICIÓN DE CARGAS
# -----
ops.timeSeries("Linear", 1) # Serie temporal lineal (carga proporcional al factor
de tiempo) ops.pattern("Plain", 1, 1) # Patrón de carga estático (tag=1, timeSeries=1)

# Elemento 1: Carga distribuida en columna izquierda
# Parámetros: (elemento, tipo_carga, magnitud_carga)
# -8 kN/m (negativo = hacia abajo en sistema local)
ops.eleLoad("-ele", 1, "-type", "-beamUniform", -8.0)

# Elemento 2: Carga puntual en viga inferior
# Parámetros: (elemento, tipo_carga, magnitud, posición_relativa)
# -30 kN en L/2 (0.5 = mitad del elemento)
ops.eleLoad("-ele", 2, "-type", "-beamPoint", -30.0, 0.5)

# Nodo 9: Momento puntual
# Parámetros: (nodo, Fx, Fy, Mz)
# Momento 20 kN·m en sentido antihorario (positivo según convención)
ops.load(9, 0.0, 0.0, 20.0)

# Nodo 5: Carga puntual vertical
# -30 kN en dirección Y (negativo = hacia abajo)
ops.load(5, 0.0, -30.0, 0.0)

# -----
# GRÁFICA SISTEMA ORIGINAL
# -----
# Crear gráfica para visualizar la geometría y cargas
plt.figure(figsize=(8, 6))
plt.title('SISTEMA ORIGINAL', fontsize=14, fontweight='bold')

# Dibujar elementos estructurales (líneas negras)
for element_data in Elementos:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=2)

# Dibujar nodos principales (puntos rosados)
for i in range(1, 9):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'o', color='hotpink', markersize=7, zorder=2)
```

```
# Dibujar apoyos con simbología específica
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=2) # Nodo 1
plt.plot(3, 3-0.15, '^', color='maroon', markersize=20, zorder=2) # Nodo 3
plt.scatter(-0.15, 6, marker=(3, 0, 270), s=700, color='maroon', zorder=2) # Nodo 4
plt.plot(4, 6-0.15, '^', color='maroon', markersize=20, zorder=2) # Nodo 6

# --- Dibujo de cargas (solo visualización gráfica) ---
# Carga distribuida en columna (elemento 1)
xi, yi = ops.nodeCoord(1) xf, yf = ops.nodeCoord(2)

x_vals = np.linspace(xi, xf, 15)
y_vals = np.linspace(yi, yf, 15)

# Flechas horizontales representan carga distribuida
for x, y in zip(x_vals, y_vals):
    plt.arrow(x-0.8, y, 0.6, 0, head_width=0.15, head_length=0.2, fc='indigo',
              ec='indigo', zorder=4)

plt.plot([xi-0.8, xf-0.8], [yi, yf], '--', color='indigo', linewidth=2, zorder=3)
plt.text(-2, 1.5, '8 kN/m', color='indigo', fontsize=11, fontweight='bold', zorder=5)

# Carga puntual en elemento 2
plt.arrow(1.5, 4.1, 0, -0.8, head_width=0.1, head_length=0.2, fc='steelblue',
          ec='steelblue', linewidth=3, zorder=4)
plt.text(1.6, 3.5, '-30 kN', color='steelblue', fontweight='bold', fontsize=11,
         zorder=5)

# Carga puntual en nodo 5
plt.arrow(2, 7.1, 0, -0.8, head_width=0.1, head_length=0.2, fc='steelblue',
          ec='steelblue', linewidth=3, zorder=4)
plt.text(2.1, 6.5, '-30 kN', color='steelblue', fontweight='bold', fontsize=11,
         zorder=5)

# Momento en nodo 9 (representación gráfica con símbolo ⌣)
plt.text(-0.3, 4.2, "⌣", color="goldenrod", fontsize=30, fontweight='bold', zorder=5)
plt.text(-1.5, 4.5, "20 kN·m", color="goldenrod", fontsize=11, fontweight='bold',
         zorder=5)

# Configuración del gráfico
plt.xlabel('X [m]', fontsize=12)
plt.ylabel('Y [m]', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

```
# -----
# CONFIGURACIÓN DEL ANÁLISIS
# -----
ops.system("BandSPD")      # Solver para matrices banda simétrica definida positiva
ops.numberer("RCM")        # Numeración de grados de libertad (Reverse Cuthill-McKee)
                             # optimiza el ancho de banda
ops.constraints("Plain")    # Imposición directa de restricciones (método de
                             # eliminación)
ops.integrator("LoadControl", 1.0) # Control por carga: aplica el 100% de la carga
                             # en 1 paso
ops.algorithm("Linear")     # Algoritmo de solución lineal (suficiente para análisis
                             # elástico lineal)
ops.analysis("Static")     # Tipo de análisis: estático
ops.analyze(1)             # Ejecutar análisis (1 paso)

# -----
# RESULTADOS
# -----
print("\n===== RESULTADOS =====")

print("\n=== REACCIONES ===")
ops.reactions() # Calcular reacciones en los nodos con restricciones
for i in [1, 3, 4, 6]:
    Rx = ops.nodeReaction(i, 1) # Reacción en X (kN)
    Ry = ops.nodeReaction(i, 2) # Reacción en Y (kN)
    Rz = ops.nodeReaction(i, 3) # Momento de reacción (kN-m)
    print(f"Node {i}: Rx = {Rx:.3f} kN, Ry = {Ry:.3f} kN, Mz = {Rz:.3f} kN-m")

print("\n=== DESPLAZAMIENTOS NODALES ===")
for i in range(1, 10):
    disp = ops.nodeDisp(i) # [dx, dy, θz] en metros y radianes
    print(f"Node {i}: Ux = {disp[0]:.3e} m, Uy = {disp[1]:.3e} m, θz = {disp[2]:.3e}
rad")

# -----
# GRÁFICA SISTEMA DEFORMADO
# -----
fig, ax = plt.subplots(figsize=(8, 6)) # Crear figura para la deformada
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar la deformada usando opsv
# El parámetro 'ax' especifica los ejes donde dibujar
opsv.plot_defo(ax=ax)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

```
print("\n=== FUERZAS INTERNAS ===")
for element_data in Elementos:
    ele_id = element_data["ID"]
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    fuerzas = ops.eleResponse(ele_id, 'localForces')

    print(f"Elemento {ele_id}:")

    # Elementos de MARCO (dispBeamColumn) - IDs 1-4
    # Estos elementos tienen comportamiento flexional y axial
    if ele_id in [1, 2, 3, 4]:
        # Formato típico 2D: [P_i, V_i, M_i, P_j, V_j, M_j]
        # P: fuerza axial, V: cortante, M: momento flector
        Pi, Vi, Mi, Pj, Vj, Mj = fuerzas
        print(f"Nodo {Ni}: N = {Pi:.3f} kN, V = {Vi:.3f} kN, M = {Mi:.3f} kN-m")
        print(f"Nodo {Nf}: N = {Pj:.3f} kN, V = {Vj:.3f} kN, M = {Mj:.3f} kN-m\n")

    # Elementos tipo CERCHA (IDs 5-11)
    # Con inercia despreciable, solo tienen fuerza axial significativa
    else:
        # Solo axial (P_i = -P_j por equilibrio)
        P = fuerzas[0]
        # Determinar tipo de esfuerzo según el signo (negativo = tracción en OpenSees)
        tipo = "Fuerza axial nula" if abs(P)<1e-6 else "Tracción" if P<0 else
            "Compresión"
        print(f" Axial = {P:.3f} kN - {tipo}\n")

# -----
# DIAGRAMAS DE ESFUERZOS
# -----
# --- Diagrama de Fuerza Axial ---
fig_n = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA AXIAL (kN)', fontsize=14, fontweight='bold')
ax_n = plt.gca()

# sf_type: tipo de fuerza ('N' = axial)
# sfac: factor de escala para visualización (ajustado para este modelo)
# nep: número de puntos para dibujar el diagrama
opsv.section_force_diagram_2d(sf_type='N', sfac=0.008, nep=20, ax=ax_n)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

```
# --- Diagrama de Fuerza Cortante ---
fig_v = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE FUERZA CORTANTE (kN)', fontsize=14, fontweight='bold')
ax_v = plt.gca()

opsv.section_force_diagram_2d(sf_type='V', sfac=0.05, nep=20, ax=ax_v)

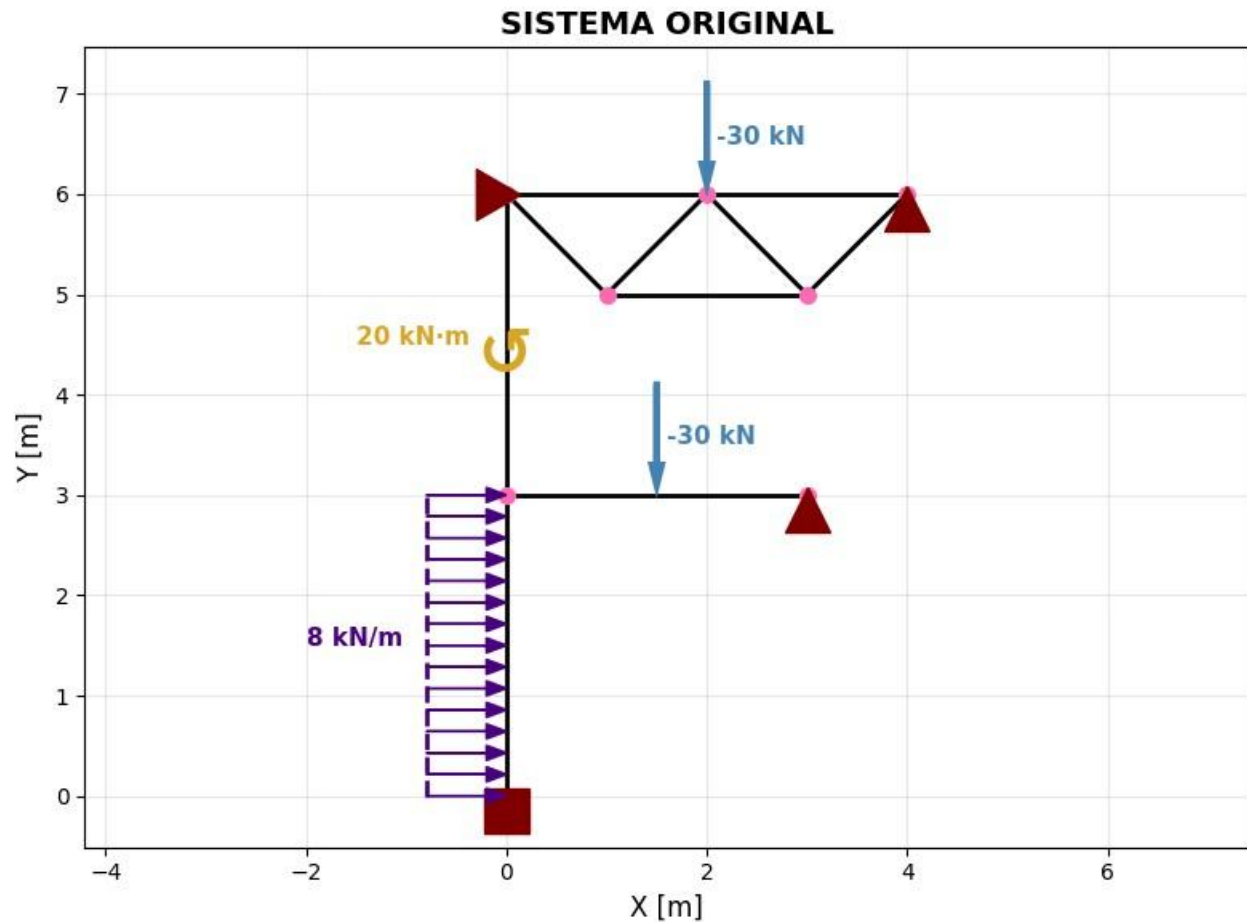
plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

# --- Diagrama de Momento Flector ---
fig_m = plt.figure(figsize=(8, 6))
plt.title('DIAGRAMA DE MOMENTO FLECTOR (kN-m)', fontsize=14, fontweight='bold')
ax_m = plt.gca()

opsv.section_force_diagram_2d(sf_type='M', sfac=0.05, nep=20, ax=ax_m)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()
```

MODELO EN OPENSEES DEL EJERCICIO 2 – ESTRUCTURAS MIXTAS



===== RESULTADOS =====

=== REACCIONES ===

Nodo 1: $R_x = -9.898 \text{ kN}$, $R_y = 9.390 \text{ kN}$, $M_z = 3.985 \text{ kN-m}$

Nodo 3: $R_x = -20.420 \text{ kN}$, $R_y = 11.220 \text{ kN}$, $M_z = -0.000 \text{ kN-m}$

Nodo 4: $R_x = -8.682 \text{ kN}$, $R_y = 24.390 \text{ kN}$, $M_z = 0.000 \text{ kN-m}$

Nodo 6: $R_x = 15.000 \text{ kN}$, $R_y = 15.000 \text{ kN}$, $M_z = -0.000 \text{ kN-m}$

=== DESPLAZAMIENTOS NODALES ===

Nodo 1: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = 0.000e+00 \text{ rad}$

Nodo 2: $U_x = 2.346e-05 \text{ m}$, $U_y = -1.259e-05 \text{ m}$, $\theta_z = -2.035e-04 \text{ rad}$

Nodo 3: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = 4.245e-04 \text{ rad}$

Nodo 4: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = -1.099e-04 \text{ rad}$

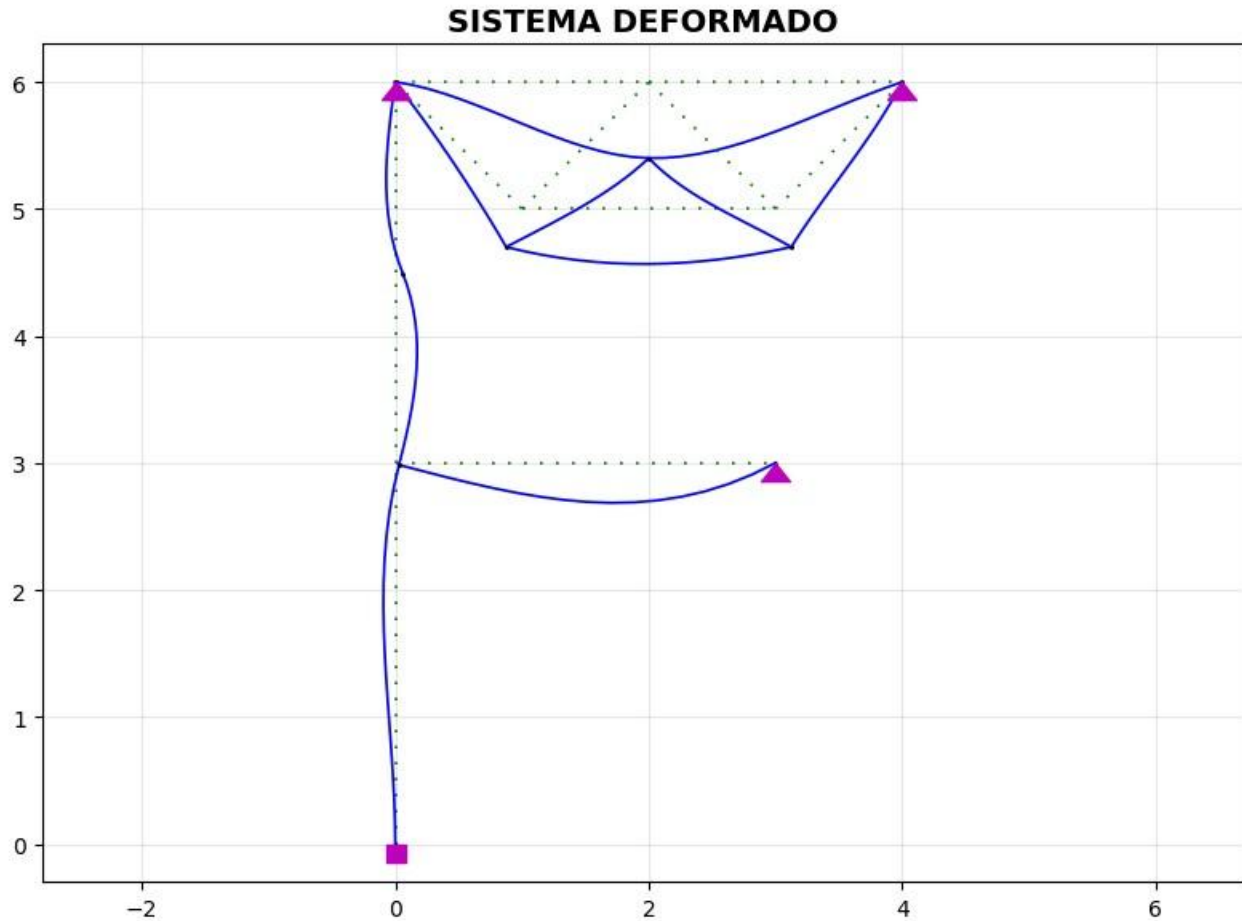
Nodo 5: $U_x = 8.911e-17 \text{ m}$, $U_y = -4.715e-04 \text{ m}$, $\theta_z = -1.914e-05 \text{ rad}$

Nodo 6: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = 2.663e-04 \text{ rad}$

Nodo 7: $U_x = -9.766e-05 \text{ m}$, $U_y = -2.358e-04 \text{ m}$, $\theta_z = -2.244e-04 \text{ rad}$

Nodo 8: $U_x = 9.766e-05 \text{ m}$, $U_y = -2.358e-04 \text{ m}$, $\theta_z = 1.956e-04 \text{ rad}$

Nodo 9: $U_x = 4.681e-05 \text{ m}$, $U_y = -6.293e-06 \text{ m}$, $\theta_z = 3.135e-04 \text{ rad}$



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 9.390 \text{ kN}$, $V = 9.898 \text{ kN}$, $M = 3.985 \text{ kN-m}$

Nodo 2: $N = -9.390 \text{ kN}$, $V = 14.102 \text{ kN}$, $M = -10.292 \text{ kN-m}$

Elemento 2:

Nodo 2: $N = 20.420 \text{ kN}$, $V = 18.780 \text{ kN}$, $M = 11.339 \text{ kN-m}$

Nodo 3: $N = -20.420 \text{ kN}$, $V = 11.220 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 3:

Nodo 2: $N = -9.390 \text{ kN}$, $V = 6.318 \text{ kN}$, $M = -1.047 \text{ kN-m}$

Nodo 9: $N = 9.390 \text{ kN}$, $V = -6.318 \text{ kN}$, $M = 10.524 \text{ kN-m}$

Elemento 4:

Nodo 9: $N = -9.390 \text{ kN}$, $V = 6.318 \text{ kN}$, $M = 9.476 \text{ kN-m}$

Nodo 4: $N = 9.390 \text{ kN}$, $V = -6.318 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 5:

Axial = -0.000 kN - Fuerza axial nula

Elemento 6:

Axial = 0.000 kN - Fuerza axial nula

Elemento 7:

Axial = -30.000 kN - Tracción

Elemento 8:

Axial = -21.213 kN - Tracción

Elemento 9:

Axial = 21.213 kN - Compresión

Elemento 10:

Axial = 21.213 kN - Compresión

Elemento 11:

Axial = -21.213 kN - Tracción

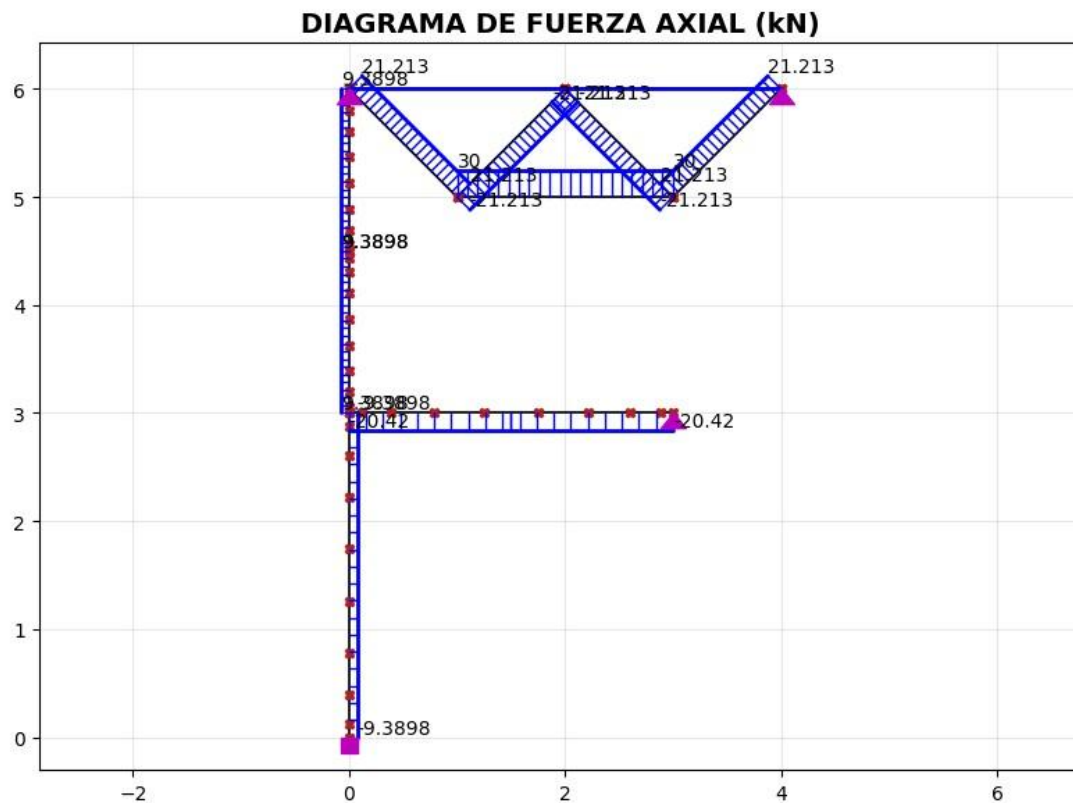


DIAGRAMA DE FUERZA CORTANTE (kN)

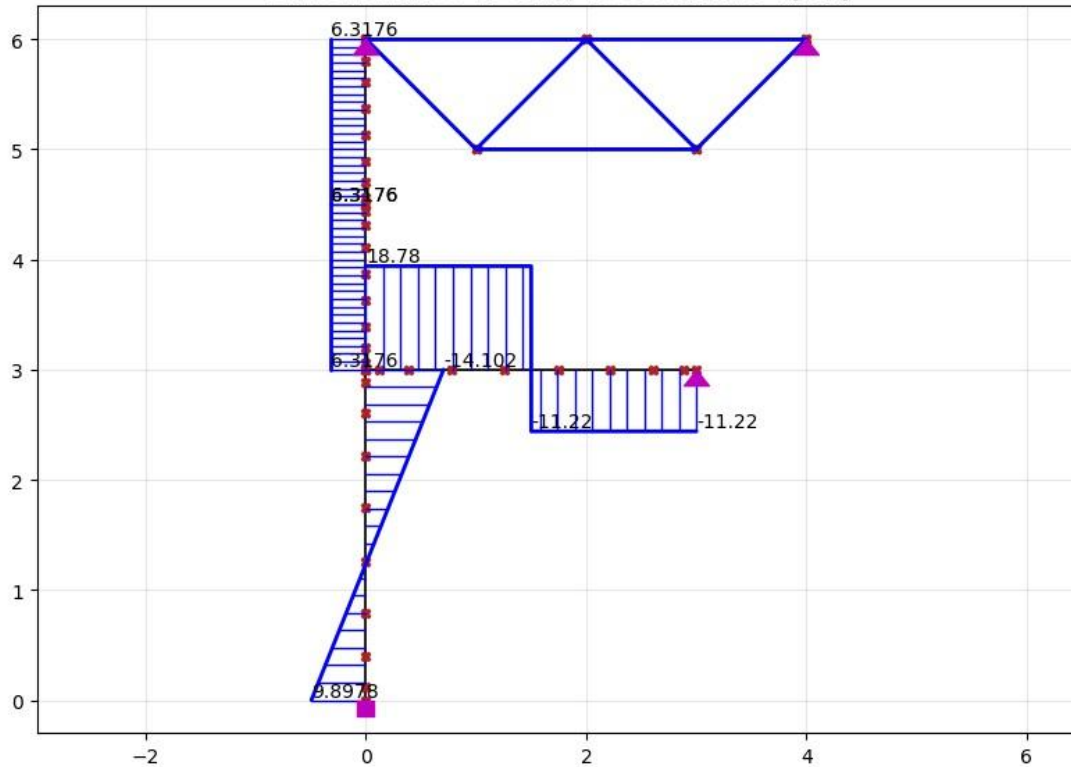
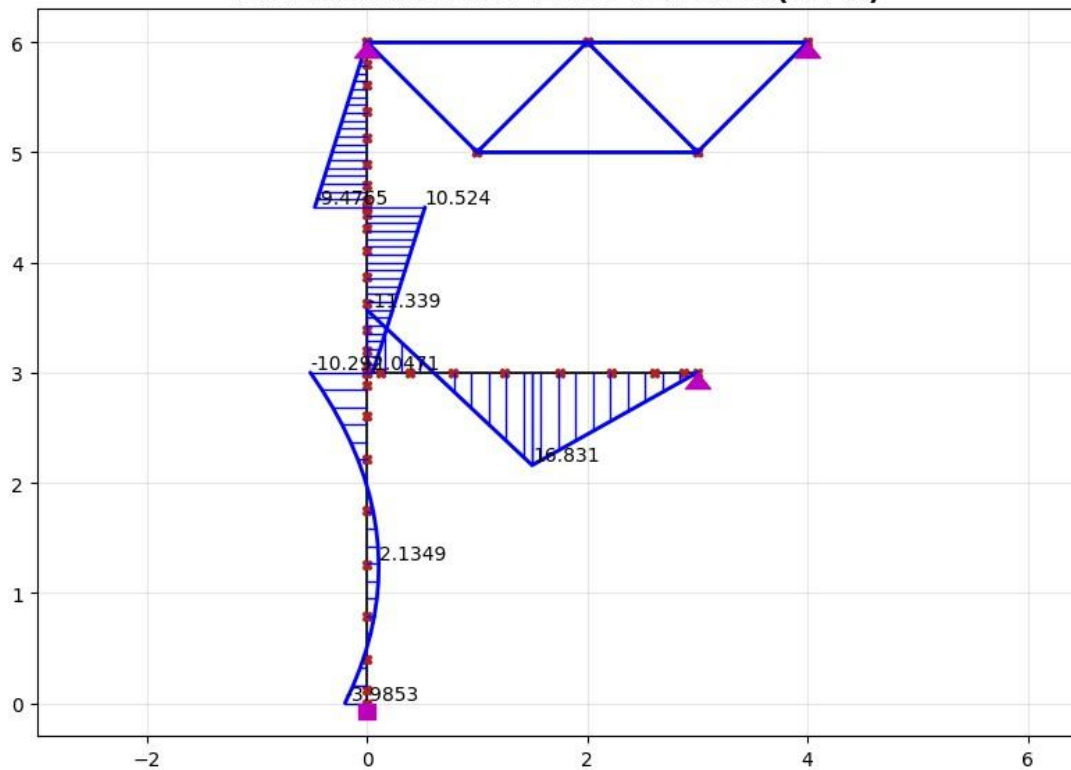


DIAGRAMA DE MOMENTO FLECTOR (kN-m)



MÉTODO MATRICIAL DE RIGIDEZ EN PYTHON SISTEMAS MIXTOS: EJERCICIO 3

```
import numpy as np
import math
import matplotlib.pyplot as plt

print("MÉTODO MATRICIAL DE RIGIDEZ PARA ESTRUCTURAS MIXTAS: EJERCICIO 3")

# -----
# DATOS DE ENTRADA
# -----
print("\n==== DATOS DE ENTRADA =====")

# Propiedades de Los materiales y sección transversal
# La estructura combina elementos de hormigón (Ec) y acero (Ea)
Ec = 24870062.324 # Módulo de elasticidad del hormigón (kN/m²)
Es = 2e8          # Módulo de elasticidad del acero (kN/m²)

# Vector de módulos para cada elemento (17 elementos)
E = np.array([Ec, Ec, Ec, Es, Es, Es, Ec, Es, Ec, Ec, Ec, Es, Es, Es, Es, Ec, Ec])

# Definición de secciones transversales para diferentes tipos de elementos
# VIGA: 0.3m x 0.35m (sección rectangular de hormigón)
Av = 0.3 * 0.35 # Área (m²) = base x altura
Iv = (1/12) * 0.3 * (0.35**3) # Inercia (m⁴) = (b·h³)/12

# COLUMNA: 0.3m x 0.3m (sección cuadrada de hormigón)
Acl = 0.3 * 0.3 # Área (m²)
Icl = (1/12) * 0.3 * (0.3**3) # Inercia (m⁴)

# BARRA DE ACERO (perfil tubular): 100mm de lado, espesor 4mm
Ab = (0.1**2) - ((0.1 - (2 * 0.004))**2) # Área neta (m²)
Ib = 1e-16 # Inercia despreciable (solo trabaja axialmente)

# CABLES DE ACERO: sección pequeña
Acb = 0.001 # Área (m²)
Icb = 1e-16 # Inercia despreciable

# RESORTE DE ACERO
Ar = 0.0003 # Área (m²) (Para rigidez equivalente)
Ir = 1e-16 # Inercia despreciable

# Área de la sección transversal (m²) para cada elemento
A = np.array([Acl, Acl, Acl, Acb, Ab, Ab, Av, Ar, Av, Av, Acl, Acl, Acb, Ab, Ab, Av, Acl])

# Inercia de la sección transversal (m⁴) para cada elemento
I = np.array([Icl, Icl, Icl, Icb, Ib, Ib, Iv, Ir, Iv, Iv, Icl, Icl, Icb, Ib, Ib, Iv, Icl])
```

```
# Longitudes de Los elementos (m)
L = np.array([6, 3, 3, 5, 4, 5, 8, 6, 8, 8, 3, 3, 5, 4, 5, 8, 6])

# Ángulos de inclinación de Los elementos respecto a La horizontal (°)
a1 = math.degrees(math.atan(3/4)) # Ángulo de Las diagonales (°)
a = np.array([90, 90, 90, -a1, 0, a1, 0, 90, 0, 0, 90, 90, a1, 0, 180-a1, 0, 90])

m = 17 # Número de elementos
n = 12 # Número de nodos
GL = n * 3 # Grados de Libertad totales (36: 12 nodos x 3 GL: Dx, Dy, θz)

# Impresión de datos de entrada (formato científico para valores pequeños)
print(f"Módulo de elasticidad: {np.array2string(E, formatter={'float_kind': lambda x: f'{x:.2e}'})} kN/m²")
print(f"Área sección transversal: {np.array2string(A, formatter={'float_kind': lambda x: f'{x:.2e}'})} m²")
print(f"Inercia sección transversal: {np.array2string(I, formatter={'float_kind': lambda x: f'{x:.2e}'})} m⁴")
print(f"Longitud de los elementos: {L} m")
print(f"Ángulo de inclinación de los elementos: {np.round(a, 2)} °")
print(f"Número de elementos: {m}")
print(f"Número de nodos: {n}")
print(f"Número de grados de libertad: {GL}")

# -----
# COORDENADAS DE LOS NODOS
# -----
# Matriz de coordenadas nodales en el plano X-Y (unidades: metros)
# Organización: filas = nodos (1 a 12), columnas = coordenadas (x, y)
coordenadas_nodos = np.array([
    [0, 0], # Nodo 1
    [0, 6], # Nodo 2
    [0, 9], # Nodo 3
    [0, 12], # Nodo 4
    [4, 9], # Nodo 5
    [8, 6], # Nodo 6
    [8, 12], # Nodo 7
    [16, 12], # Nodo 8
    [16, 9], # Nodo 9
    [12, 9], # Nodo 10
    [16, 6], # Nodo 11
    [16, 0]]) # Nodo 12
```

```
# -----
# CONECTIVIDAD DE ELEMENTOS
# -----
# Cada fila: [nodo_inicial, nodo_final]
Elementos = np.array([
    [1, 2],      # Elemento 1
    [2, 3],      # Elemento 2
    [3, 4],      # Elemento 3
    [4, 5],      # Elemento 4
    [3, 5],      # Elemento 5
    [2, 5],      # Elemento 6
    [2, 6],      # Elemento 7
    [6, 7],      # Elemento 8
    [4, 7],      # Elemento 9
    [7, 8],      # Elemento 10
    [9, 8],      # Elemento 11
    [11, 9],     # Elemento 12
    [10, 8],     # Elemento 13
    [10, 9],     # Elemento 14
    [11, 10],    # Elemento 15
    [6, 11],     # Elemento 16
    [12, 11]])

# -----
# GRADOS DE LIBERTAD POR ELEMENTO - NUMERACIÓN AUTOMÁTICA
# -----
print("\n==== GRADOS DE LIBERTAD =====")
Nx, Ny, Nz, Fx, Fy, Fz = [], [], [], [], [], []

for i in range(m):
    # Fórmula: GL = (nodo-1)*3 + 1,2,3 para Dx, Dy, Dz
    ni = Elementos[i][0] # Nodo inicial
    nf = Elementos[i][1] # Nodo final

    Nx.append(ni*3 - 2) # GL desplazamiento X - nodo inicial (1-based)
    Ny.append(ni*3 - 1) # GL desplazamiento Y - nodo inicial
    Nz.append(ni*3)     # GL rotación Z - nodo inicial
    Fx.append(nf*3 - 2) # GL desplazamiento X - nodo final
    Fy.append(nf*3 - 1) # GL desplazamiento Y - nodo final
    Fz.append(nf*3)     # GL rotación Z - nodo final

nombres = ['Nx:', 'Ny:', 'Nz:', 'Fx:', 'Fy:', 'Fz:']
datos = [Nx, Ny, Nz, Fx, Fy, Fz]

# Imprimir encabezado de la tabla
print(f"{'':6}", end="") # Espacio para los nombres
for elemento in range(len(datos[0])): # m = número de elementos
    print(f"E{elemento+1:<3d}", end="")
print()
```

```
# Imprimir cada fila de datos (tabla de grados de libertad)
for nombre, lista in zip(nombres, datos):
    print(f"{nombre:4}", end="") # Nombre de la fila alineado
    for valor in lista:
        print(f"{valor:4d}", end="")
    print()

# -----
# DEFINICIÓN DE CARGAS POR ELEMENTO
# -----
# Formato: [tipo, dirección, color, w_inicial, w_final]
# - 'Uniforme': carga distribuida uniforme (w_inicial = w_final)
# - 'Local_y': dirección perpendicular al elemento
# Nota: Solo el elemento 17 tiene carga definida
Cargas = [[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [],
           ['Uniforme', 'Local_y', 'navy', 5, 5]] # Elem 17: carga uniforme de 5 kN/m

# -----
# PREPROCESAMIENTO GEOMÉTRICO DE TODOS LOS ELEMENTOS
# -----
# Calcula propiedades geométricas fundamentales para cada elemento:
geom = [] # Almacena en lista 'geom' para uso posterior en gráficas

for e in range(m):
    ni, nf = Elementos[e][0], Elementos[e][1]

    xi, yi = coordenadas_nodos[ni-1] # Coordenadas nodo inicial (índice 0-based)
    xf, yf = coordenadas_nodos[nf-1] # Coordenadas nodo final

    dx = xf - xi # Diferencia en X
    dy = yf - yi # Diferencia en Y
    Le = math.sqrt((dx**2) + (dy**2)) # Longitud del elemento

    # Vector director unitario (apunta del nodo inicial al final)
    ex = dx / Le
    ey = dy / Le

    # Vector normal unitario (perpendicular, rotado 90° antihorario)
    # Útil para visualización de diagramas perpendiculares al elemento
    nx = -ey
    ny = ex
    geom.append([ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny])
```

```
# -----
# GRÁFICA SISTEMA ORIGINAL
# -----
plt.figure(figsize=(9, 5))
plt.title('SISTEMA ORIGINAL', fontsize=14, fontweight='bold')

# Dibujar elementos estructurales (líneas negras)
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '-', color='black', linewidth=2, zorder=1)

# Dibujar nodos (puntos dorados)
for i, (x, y) in enumerate(coordenadas_nodos):
    plt.plot(x, y, 'o', color='gold', markersize=7, zorder=2)

# Dibujar apoyos empotrados (cuadrados marrones)
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=2) # Nodo 1
plt.plot(16, -0.15, 's', color='maroon', markersize=20, zorder=2) # Nodo 12

# --- Dibujo de cargas (solo visualización gráfica) ---
# Carga distribuida en columna derecha inferior (Elemento 17)
xi, yi = coordenadas_nodos[11] # Nodo 12 (índice 11)
xf, yf = coordenadas_nodos[10] # Nodo 11 (índice 10)

dx = xf - xi
dy = yf - yi
Le = math.sqrt(dx**2 + dy**2)

# Puntos a lo largo del elemento para dibujar flechas
x_vals = np.linspace(xi, xf, 15)
y_vals = np.linspace(yi, yf, 15)

# Flechas horizontales representan carga distribuida horizontal
for x, y in zip(x_vals, y_vals):
    plt.arrow(x + 0.8, y, -0.6, 0, head_width=0.15, head_length=0.2,
              fc='navy', ec='navy', zorder=4)

plt.plot([xi + 0.8, xf + 0.8], [yi, yf], '--', color='navy', linewidth=2, zorder=3)
plt.text(17, 3, '5 kN/m', color='navy', fontsize=11, fontweight='bold', zorder=5)

# Carga puntual en nodo 5 (20 kN hacia abajo)
plt.arrow(4, 9, 0, -1.8, head_width=0.1, head_length=0.2,
          fc='seagreen', ec='seagreen', linewidth=3, zorder=4)
plt.text(4.2, 7.8, '-20 kN', color='seagreen', fontweight='bold', fontsize=11, zorder=5)

# Carga puntual en nodo 10 (70 kN hacia abajo)
plt.arrow(12, 9, 0, -1.5, head_width=0.1, head_length=0.2,
          fc='seagreen', ec='seagreen', linewidth=3, zorder=4)
plt.text(9.8, 7.8, '-70 kN', color='seagreen', fontweight='bold', fontsize=11, zorder=5)
```

```
# Carga puntual diagonal en nodo 6 (60 kN en dirección 45°)
plt.arrow(8, 6, -1.8, -1.5, head_width=0.1, head_length=0.2,
          fc='seagreen', ec='seagreen', linewidth=3, zorder=4)
plt.text(6.2, 4, '-60 kN', color='seagreen', fontweight='bold', fontsize=11, zorder=5)

# Configuración del gráfico
plt.xlabel('X [m]', fontsize=12)
plt.ylabel('Y [m]', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.show()

# -----
# ENSAMBLE DE LA MATRIZ DE RIGIDEZ GLOBAL
# -----
print("\n==== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL =====")
kG = np.zeros((GL, GL)) # Inicializar matriz de rigidez global (36x36) con ceros

# Listas para almacenar matrices de cada elemento (para post-procesamiento)
kL_elementos = [] # Matrices de rigidez local (6x6) en coordenadas locales
T_elementos = [] # Matrices de transformación (6x6)

# Ensamblar matriz de rigidez de los elementos
for i in range(m):
    # Ángulo de inclinación (conversión a radianes)
    theta = math.radians(a[i])
    c = math.cos(theta)
    s = math.sin(theta)

    # Propiedades de rigidez del elemento
    AE = A[i] * E[i] # Rigidez axial (EA) en kN
    EI = E[i] * I[i] # Rigidez flexional (EI) en kN·m²
    L2 = (L[i])**2 # Longitud al cuadrado (m²)
    L3 = (L[i])**3 # Longitud al cubo (m³)

    # Matriz de rigidez LOCAL para elemento biempotrado (6x6)
    # Para elementos con I=0 (barras, cables, resortes), los términos flexionales
    # serán cercanos a cero
    kL = [[AE/L[i], 0, 0, -AE/L[i], 0, 0],
          [0, (12*EI)/L3, (6*EI)/L2, 0, -(12*EI)/L3, (6*EI)/L2],
          [0, (6*EI)/L2, (4*EI)/L[i], 0, -(6*EI)/L2, (2*EI)/L[i]],
          [-AE/L[i], 0, 0, AE/L[i], 0, 0],
          [0, -(12*EI)/L3, -(6*EI)/L2, 0, (12*EI)/L3, -(6*EI)/L2],
          [0, (6*EI)/L2, (2*EI)/L[i], 0, -(6*EI)/L2, (4*EI)/L[i]]]
    kL_elementos.append(kL)
```

```
# Matriz de transformación de coordenadas
T = [[c, s, 0, 0, 0, 0],
      [-s, c, 0, 0, 0, 0],
      [0, 0, 1, 0, 0, 0],
      [0, 0, 0, c, s, 0],
      [0, 0, 0, -s, c, 0],
      [0, 0, 0, 0, 0, 1]]
T_elementos.append(T)

# Transformar matriz LOCAL a GLOBAL:  $K_{global} = T^T \cdot K_{local} \cdot T$ 
T_T = np.transpose(T) # Transformación inversa (local a global)
kg_e = np.matmul(np.matmul(T_T, kL), T) # Matriz de rigidez del elemento en
coordenadas globales (6x6)

# Ensamblar en matriz global - Ajuste por indexación Python (0-based)
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]

# Sumar contribución del elemento a la matriz global
for ii, gl_i in enumerate(GL_elem):
    for jj, gl_j in enumerate(GL_elem):
        kG[gl_i, gl_j] += kg_e[ii, jj] # Ensamblaje directo de rigidez

# Código comentado para depuración (útil para verificar cada elemento)
#print(f"ELEMENTO {i+1}")
#print(f"Longitud: {L[i]} m, Ángulo: {a[i]:.0f}°, Módulo de elasticidad: {E[i]:.2f} kN/m²")
#print(f"Área: {A[i]} m², Inercia: {I[i]:.3e} m⁴")
#print(f"Matriz global del elemento: \n {np.round(kg_e, 0)}\n")
#print(f"Matriz global del sistema después del elemento {i+1}: \n {np.round(kG, 0)}")
#print(f"Matriz global del sistema: \n {np.round(kG, 0)}")

# -----
# VECTOR DE FUERZAS DE EMPOTRAMIENTO PERFECTO (FEM)
# -----

print("\n== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO ==")
FEM_L_elementos = [] # FEM en coordenadas locales por elemento
FEM_G = np.zeros(GL) # FEM en coordenadas globales (vector de 36 componentes)

for i in range(m):
    # Verificar si el elemento tiene carga
    if len(Cargas[i])>0 and Cargas[i][0] == 'Uniforme' and Cargas[i][1] == 'Local_y':
        # Carga en dirección Y local (perpendicular al elemento)
        wi = -Cargas[i][3] # w inicial positiva hacia abajo (kN/m)
        wf = -Cargas[i][4] # w final positiva hacia abajo (kN/m)

        # Fórmulas analíticas para viga biempotrada con carga trapezoidal
        # Estas fórmulas se obtienen integrando las funciones de forma de la viga
        Ryi = ((7*wi*L[i])/20)+((3*wf*L[i])/20) # Reacción vertical en nodo inicial (kN)
        Mzi = ((wi*(L[i]**2))/20) + ((wf*(L[i]**2))/30) # Momento en nodo inicial (kN-m)
        Ryf = ((3*wi*L[i])/20) + ((7*wf*L[i])/20) # Reacción vertical en nodo final (kN)
        Mzf = -(((wi*(L[i]**2))/30)+((wf*(L[i]**2))/20)) # Momento en nodo final (kN-m)
```



```
FEM_L = [0, Ryi, Mzi, 0, Ryf, Mzf]

else:
    FEM_L = [0, 0, 0, 0, 0, 0] # Elementos sin carga o tipos no implementados

FEM_L_elementos.append(FEM_L) # Almacenar para post-procesamiento

# Transformar FEM Local a global: FEM_global = T^T · FEM_Local
# T^T transforma de coordenadas Locales a globales
FEM_Ge = np.matmul(np.transpose(T_elementos[i]), FEM_L)

# Ensamblar en vector global
GL_elem = [Nx[i]-1, Ny[i]-1, Nz[i]-1, Fx[i]-1, Fy[i]-1, Fz[i]-1]

for jj, gl_j in enumerate(GL_elem):
    FEM_G[gl_j] += FEM_Ge[jj] # Sumar contribución del elemento

print(np.round(FEM_G, 2)) # Mostrar FEM global redondeado a 2 decimales

# -----
# CONDICIONES DE CONTORNO (RESTRICCIONES)
# -----
print("\n==== DEFINICIÓN DE RESTRICCIONES =====")
# Vector de restricciones: 1 = restringido (desplazamiento conocido = 0)
#                               0 = Libre (desplazamiento desconocido)
# Nodos 1 y 12: EMPOTRADOS (Dx, Dy, Dz = 1)
# Todos Los demás nodos: LIBRES
#
# GL: 1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20
restricciones = np.array([1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
                          0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1])

# Nodo 1 (GL1-3): 1,1,1 = empotrado
# Nodos 2-11 (GL4-33): todos Libres
# Nodo 12 (GL34-36): 1,1,1 = empotrado

# Estado de cada grado de libertad
for i in range(n):
    rtx = "(Restringido)" if restricciones[i*3] == 1 else "(Libre)"
    rty = "(Restringido)" if restricciones[i*3+1] == 1 else "(Libre)"
    rtz = "(Restringido)" if restricciones[i*3+2] == 1 else "(Libre)"

    print(f"Nodo {i+1}: Rx={restricciones[i*3]} {rtx}, Ry={restricciones[i*3+1]} {rty}, Rz={restricciones[i*3+2]} {rtz}")
```

```
# -----
# VECTOR DE FUERZAS EXTERNAS
# -----
print("\n===== VECTOR DE FUERZAS EXTERNAS =====")
# Fuerzas nodales aplicadas directamente (cargas puntuales en nodos)
# -20 kN vertical en nodo 5 (GL 14)
# -70 kN vertical en nodo 10 (GL 29)
# -60 kN diagonal (45°) en nodo 6 (GL 16 y 17)
P6 = -60 # kN (negativo = hacia abajo/izquierda)
a_P6 = math.radians(45) # Ángulo de 45° (diagonal) [rad]

# Vector de fuerzas externas (36 componentes)
F = np.array([
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -20, 0, # GL 1-15: -20 kN en GL14 (nodo 5 Dy)
    P6*math.cos(a_P6), P6*math.sin(a_P6), 0, # GL 16-18: carga diagonal en nodo 6 (Dx y Dy)
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -70, 0, # GL 19-30: -70 kN en GL29 (nodo 10 Dy)
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, # GL 31-36: sin carga

])

print(f"Vector de fuerzas externas: {np.round(F, 2)} kN")

# -----
# REDUCCIÓN DEL SISTEMA
# -----
print("\n=== REDUCCIÓN DEL SISTEMA ===")
# Identificar grados de libertad activos (no restringidos)
GL_activos = np.where(restricciones == 0)[0] # Índices donde restricciones = 0
n_GL_activos = len(GL_activos)
print(f"Grados de libertad activos: {n_GL_activos}")
print(f"Índices: {GL_activos+1}") # +1 para mostrar numeración 1-based

# Extraer submatrices correspondientes a GL activos (partición de la matriz global)
kG_reducida = kG[np.ix_(GL_activos, GL_activos)] # Matriz de rigidez reducida
F_reducido = F[GL_activos] # Vector de fuerzas reducido
FEM_G_reducido = FEM_G[GL_activos] # Vector FEM reducido

#print(f"\nMatriz global reducida:\n {np.round(kG_reducida, 0)}")
print(f"Vector de fuerzas reducido: {np.round(F_reducido, 2)} kN")
print(f"Vector FEM reducido: {np.round(FEM_G_reducido, 2)} kN")

# -----
# SOLUCIÓN DEL SISTEMA
# -----
print("\n\n===== SOLUCIÓN DEL SISTEMA =====")
# Calcular matriz inversa de la matriz global reducida
kG_r_inv = np.linalg.inv(kG_reducida)

# Calcular desplazamientos desconocidos:  $U = K^{-1} \cdot (F - FEM)$ 
# La ecuación de equilibrio:  $K \cdot U = F - FEM$  (Los FEM actúan como fuerzas equivalentes)
U_desconocidos = np.matmul(kG_r_inv, F_reducido - FEM_G_reducido)
```

```
# Reconstruir vector completo de desplazamientos (incluyendo ceros en GL restringidos)
U_totales = np.zeros(GL) # Inicializar con ceros
U_totales[GL_activos]=U_desconocidos # Asignar desplazamientos calculados a GL activos

# Calcular reacciones en apoyos:  $R = K \cdot U + FEM - F$ 
# En Los GL restringidos, esta ecuación da las fuerzas de reacción
Reacciones = np.matmul(kG, U_totales) + FEM_G

# -----
# RESULTADOS
# -----
print("\n=== REACCIONES ===")
# Mostrar reacciones solo en nodos con restricciones
for i in range(n):
    if np.any(restricciones[i*3:i*3+3] == 1):
        Rx = Reacciones[i*3] # Reacción horizontal (kN)
        Ry = Reacciones[i*3+1] # Reacción vertical (kN)
        Mz = Reacciones[i*3+2] # Momento de reacción (kN-m)
        print(f"Node {i+1}: Rx = {Rx:.2f} kN, Ry = {Ry:.2f} kN, Mz = {Mz:.2f} kN-m")

print("\n=== DESPLAZAMIENTOS NODALES ===")
# Desplazamientos nodales (notación científica para valores pequeños)
for i in range(n):
    Ux = U_totales[i*3] # Desplazamiento horizontal (m)
    Uy = U_totales[i*3+1] # Desplazamiento vertical (m)
    Tz = U_totales[i*3+2] # Rotación (radianes)
    print(f"Node {i+1}: Ux = {Ux:.3e} m, Uy = {Uy:.3e} m,  $\theta_z$  = {Tz:.3e} rad")

# -----
# GRÁFICA SISTEMA DEFORMADO
# -----
FS = 10 # Factor de escala para visualización
plt.figure(figsize=(6, 5))
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Dibujar sistema original (líneas punteadas grises) como referencia
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], '--', color='grey', linewidth=2, alpha=0.5,
label='Original' if i == 0 else "")

# Dibujar sistema deformado (líneas verdes) con desplazamientos amplificados
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
```

```
# Desplazamientos de los nodos (del vector solución)
desp_i = np.array([U_totales[(ni-1)*3], U_totales[(ni-1)*3+1], U_totales[(ni-1)*3+2]))
desp_f = np.array([U_totales[(nf-1)*3], U_totales[(nf-1)*3+1], U_totales[(nf-1)*3+2]))

# Coordenadas deformadas (original + desplazamiento × factor de escala)
xi_def = xi + desp_i[0] * FS
yi_def = yi + desp_i[1] * FS
xf_def = xf + desp_f[0] * FS
yf_def = yf + desp_f[1] * FS
plt.plot([xi_def, xf_def], [yi_def, yf_def], '-', color='g', linewidth=2,
label='Deformada' if i == 0 else "")

# Dibujar apoyos en posición original (referencia)
plt.plot(0, -0.15, 's', color='maroon', markersize=20) # Apoyo empotrado Nodo 1
plt.plot(16, -0.15, 's', color='maroon', markersize=20) # Apoyo empotrado Nodo 12

plt.xlabel('X (m)')
plt.ylabel('Y (m)')
plt.legend()
plt.grid(True, alpha=0.5)
plt.axis('equal')
plt.show()

print("\n=== FUERZAS INTERNAS ===")
# Cálculo de fuerzas internas en los extremos de cada elemento
F_int_elementos = [] # Lista para almacenar las fuerzas internas de cada elemento

for i in range(m):
    # Extraer desplazamientos globales del elemento (del vector U_totales)
    Ue = np.array([
        U_totales[Nx[i]-1], # Dx nodo inicial
        U_totales[Ny[i]-1], # Dy nodo inicial
        U_totales[Nz[i]-1], # Dz nodo inicial
        U_totales[Fx[i]-1], # Dx nodo final
        U_totales[Fy[i]-1], # Dy nodo final
        U_totales[Fz[i]-1]]) # Dz nodo final

    # Transformar desplazamientos a coordenadas Locales:  $U_L = T \cdot U_G$ 
    # T transforma de global a local
    Ue_L = np.matmul(T_elementos[i], Ue)

    # Calcular fuerzas internas en coordenadas Locales:  $F = K_L \cdot U_L + FEM_L$ 
    Fe_L = np.matmul(kL_elementos[i], Ue_L) + FEM_L_elementos[i]
    F_int_elementos.append(Fe_L)
```

```
# Presentar resultados
# Formato: [N_i, V_i, M_i, N_j, V_j, M_j] en sistema local del elemento
print(f"Elemento {i+1}:")
print(f"  Nodo {Elementos[i][0]}: N = {Fe_L[0]:.3f} kN, V = {Fe_L[1]:.3f} kN, M = {Fe_L[2]:.3f} kN-m")
print(f"  Nodo {Elementos[i][1]}: N = {Fe_L[3]:.3f} kN, V = {Fe_L[4]:.3f} kN, M = {Fe_L[5]:.3f} kN-m\n")

# -----
# DIAGRAMA AXIAL
# -----
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA AXIAL", fontsize=14, fontweight="bold")
escala_axial = 0.01 # Factor de escala para visualización

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=3) # Línea del elemento (negro)

    # Extraer fuerzas axiales en extremos (del cálculo de fuerzas internas)
    Ni = F_int_elementos[i][0] # Fuerza axial en nodo inicial (kN)
    Nf = F_int_elementos[i][3] # Fuerza axial en nodo final (kN)

    # Puntos para dibujar el prisma del diagrama axial
    # El diagrama se dibuja perpendicular al elemento (dirección del vector normal)
    P1 = np.array([xi, yi]) # Punto base inicial (sobre el elemento)
    P2 = P1 + escala_axial*Ni*np.array([nx, ny]) # Despl. perpendicular según Ni
    P3 = P2 + Le * np.array([ex, ey]) # Avanzar a lo largo del elemento
    P4 = P3 + escala_axial*Nf*np.array([nx, ny]) # Despl. perpendicular según Nf

    # Dibujar relleno y contorno del diagrama
    # fill() dibuja un polígono relleno, plot() dibuja el contorno superior
    plt.fill([P1[0], P2[0], P3[0], P4[0]], [P1[1], P2[1], P3[1], P4[1]], alpha=0.3)
    plt.plot([P2[0], P3[0]], [P2[1], P3[1]])

    # Texto en punto medio indicando magnitud y tipo de esfuerzo
    xm = xi + ex * Le / 2 # Coordenada X del punto medio
    ym = yi + ey * Le / 2 # Coordenada Y del punto medio
    tipo = "Nula" if abs(Nf)<1e-6 else "Tracción" if Nf>0 else "Compresión" if Nf<0
    else "" # Determinar tipo de esfuerzo

    plt.text(xm, ym, f"{Nf:.2f} kN\n({tipo})", fontsize=8, ha='center',
            bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()
```

```
# -----
# DIAGRAMA CORTANTE
# -----
escala_cortante = 0.05 # Factor de escala para visualización
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA CORTANTE (kN)", fontsize=14, fontweight="bold")
for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]
    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2) # Línea del elemento

    # Extraer fuerzas cortantes en extremos (del cálculo de fuerzas internas)
    Vi = F_int_elementos[i][1] # Cortante en nodo inicial (kN)
    Vf = F_int_elementos[i][4] # Cortante en nodo final (kN)

    x = np.linspace(0, Le, 30) # Puntos de evaluación a lo largo del elemento

    # Obtener información de carga si existe (para diagrama parabólico)
    if len(Cargas[i])>0:
        tipo, direccion, color, w1, w2 = Cargas[i]
    else:
        w1 = 0
        w2 = 0

    # Cálculo de la fuerza cortante variable  $V(x) = \int w_y(x) dx + V_i$ 
    # donde  $w_y(x) = w1 + ((w2 - w1)/Le) * x$  (carga lineal)
    # La integral resulta en:  $V(x) = ((w2 - w1)/Le)*(x^2/2) + w1*x + Vi$ 
    V = (((w2 - w1)/Le)*((x**2)/2)) + (w1*x) + Vi

    # Coordenadas del diagrama (desplazamiento perpendicular al elemento)
    # X_diag, Y_diag son los puntos del diagrama (desplazados según V(x))
    X_diag = xi + ex*x + escala_cortante * V * nx
    Y_diag = yi + ey*x + escala_cortante * V * ny

    # Línea base del elemento (sin desplazamiento)
    X_base = xi + ex*x
    Y_base = yi + ey*x

    # Dibujar diagrama con relleno
    # plot() dibuja la línea del diagrama, fill() rellena el área entre base y diagrama
    plt.plot(X_diag, Y_diag, linewidth=1.5)
    plt.fill(np.concatenate([X_base, X_diag[:-1]]),
            np.concatenate([Y_base, Y_diag[:-1]]), alpha=0.4)

    # Etiquetas con los valores de cortante en los extremos
    # Se muestra -Vf en el extremo j por convención gráfica
    plt.text(X_diag[0], Y_diag[0], f"{{Vi:.2f}}", fontsize=8,
            bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
    plt.text(X_diag[-1], Y_diag[-1], f"{{-Vf:.2f}}", fontsize=8,
            bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
```

```
plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()

# -----
# DIAGRAMA MOMENTO
# -----

escala_momento = 0.02 # Factor de escala para visualización
plt.figure(figsize=(8, 6))
plt.title("DIAGRAMA MOMENTO (kN-m)", fontsize=14, fontweight="bold")

for i in range(m):
    ni, nf, xi, yi, xf, yf, dx, dy, Le, ex, ey, nx, ny = geom[i]

    plt.plot([xi, xf], [yi, yf], 'k', linewidth=2) # Línea del elemento

    # Extraer fuerzas internas necesarias
    Vi = F_int_elementos[i][1] # Cortante en nodo inicial (kN)
    Vf = F_int_elementos[i][4] # Cortante en nodo final (kN)
    Mi = F_int_elementos[i][2] # Momento en nodo inicial (kN-m)
    Mf = F_int_elementos[i][5] # Momento en nodo final (kN-m)

    x = np.linspace(0, Le, 30) # Puntos de evaluación a lo largo del elemento

    # Obtener información de carga si existe (para diagrama parabólico)
    if len(Cargas[i])>0:
        tipo, direccion, color, w1, w2 = Cargas[i]
    else:
        w1 = 0
        w2 = 0

    # Integración de V(x) (que ya incluye w1, w2) para obtener M(x)
    #  $M(x) = ((w2 - w1)/Le)*(x^3/6) + w1*(x^2/2) + Vi*x - Mi$ 
    M = (((w2 - w1)/Le)*((x**3)/6)) + (w1*((x**2)/2)) + Vi*x - Mi

    # Coordenadas del diagrama (desplazamiento perpendicular al elemento)
    X_diag = xi + ex*x + escala_momento * M * nx
    Y_diag = yi + ey*x + escala_momento * M * ny

    # Línea base del elemento
    X_base = xi + ex*x
    Y_base = yi + ey*x

    # Dibujar diagrama con relleno
    plt.plot(X_diag, Y_diag, linewidth=1.5)
    plt.fill(np.concatenate([X_base, X_diag[::-1]]),
             np.concatenate([Y_base, Y_diag[::-1]]), alpha=0.4)
```

```
# Etiquetas con los valores de momento en los extremos
plt.text(X_diag[0], Y_diag[0], f"{-Mi:.2f}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))
plt.text(X_diag[-1], Y_diag[-1], f"{Mf:.2f}", fontsize=8,
         bbox=dict(facecolor='white', alpha=0.9, edgecolor='none'))

plt.axis("equal")
plt.grid(True, alpha=0.3)
plt.show()
```



MÉTODO MATRICIAL DE RIGIDEZ PARA ESTRUCTURAS MIXTAS: EJERCICIO 3

===== DATOS DE ENTRADA =====

Módulo de elasticidad: [2.49e+07 2.49e+07 2.49e+07 2.00e+08 2.00e+08 2.00e+08 2.49e+07
2.00e+08 2.49e+07 2.49e+07 2.49e+07 2.49e+07 2.00e+08 2.00e+08 2.00e+08 2.49e+07
2.49e+07] kN/m²

Área sección transversal: [9.00e-02 9.00e-02 9.00e-02 1.00e-03 1.54e-03 1.54e-03
1.05e-01 3.00e-04 1.05e-01 1.05e-01 9.00e-02 9.00e-02 1.00e-03 1.54e-03 1.54e-03
1.05e-01 9.00e-02] m²

Inercia sección transversal: [6.75e-04 6.75e-04 6.75e-04 1.00e-16 1.00e-16 1.00e-16
1.07e-03 1.00e-16 1.07e-03 1.07e-03 6.75e-04 6.75e-04 1.00e-16 1.00e-16 1.00e-16
1.07e-03 6.75e-04] m⁴

Longitud de los elementos: [6 3 3 5 4 5 8 6 8 8 3 3 5 4 5 8 6] m

Ángulo de inclinación de los elementos: [90. 90. 90. -36.87 0. 36.87
0. 90. 0. 0. 90. 90. 36.87 0. 143.13 0. 90.] °

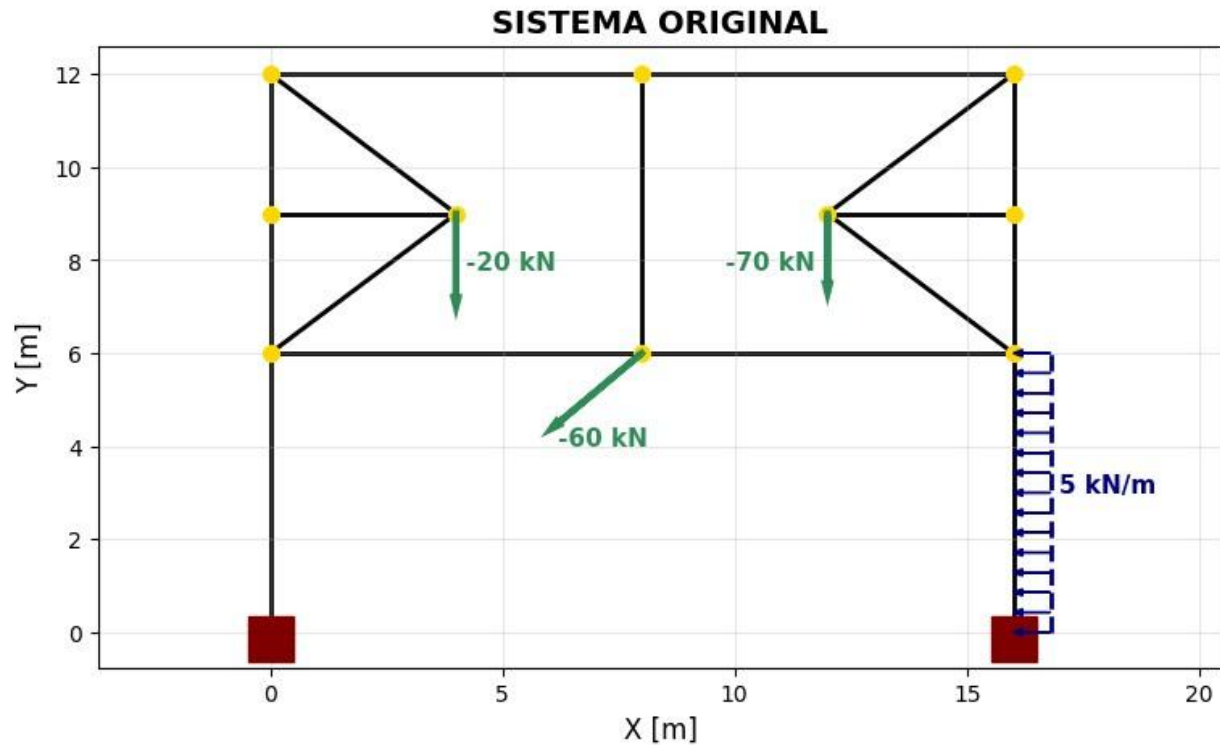
Número de elementos: 17

Número de nodos: 12

Número de grados de libertad: 36

===== GRADOS DE LIBERTAD =====

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E16	E17
Nx:	1	4	7	10	7	4	4	16	10	19	25	31	28	28	31	16	34
Ny:	2	5	8	11	8	5	5	17	11	20	26	32	29	29	32	17	35
Nz:	3	6	9	12	9	6	6	18	12	21	27	33	30	30	33	18	36
Fx:	4	7	10	13	13	13	16	19	19	22	22	25	22	25	28	31	31
Fy:	5	8	11	14	14	14	17	20	20	23	23	26	23	26	29	32	32
Fz:	6	9	12	15	15	15	18	21	21	24	24	27	24	27	30	33	33



===== ENSAMBLE MATRIZ DE RIGIDEZ GLOBAL =====

```
=== VECTOR FUERZAS DE EMPOTRAMIENTO PERFECTO === [ 0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  15. -0.  15.  15. -0. -15.]
```

===== DEFINICIÓN DE RESTRICCIONES =====

```
Nodo 1: Rx=1 (Restringido), Ry=1 (Restringido), Rz=1 (Restringido)
Nodo 2: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 3: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 4: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 5: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 6: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 7: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 8: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 9: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 10: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 11: Rx=0 (Libre), Ry=0 (Libre), Rz=0 (Libre)
Nodo 12: Rx=1 (Restringido), Ry=1 (Restringido), Rz=1 (Restringido)
```

===== VECTOR DE FUERZAS EXTERNAS =====

```
Vector de fuerzas externas: [ 0.  0.  0.  0.  0.  0.  0.  0.
0.  0.  0.  0.  0. -20.  0. -42.43 -42.43  0.  0.  0.  0.
0.  0.  0.  0.  0.  0.  0. -70.  0.  0.  0.  0.  0.
0.  0. ] kN
```

=== REDUCCIÓN DEL SISTEMA ===

Grados de libertad activos: 30

```
Indices: [ 4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
          28 29 30 31 32 33]
```

Vector de fuerzas reducido: $\begin{bmatrix} 0. & 0. & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & -20. & 0. & -42.43 & -42.43 & 0. & 0. & 0. & 0. & 0. & 0. \\ 0. & 0. & 0. & 0. & -70. & 0. & 0. & 0. & 0. & 0. & 0. \end{bmatrix}$ kN

Vector FEM reducido: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 15. -0. 15.] kN

===== SOLUCIÓN DEL SISTEMA =====

=== REACCIONES ===

Nodo 1: $R_x = 32.54 \text{ kN}$, $R_y = 60.49 \text{ kN}$, $M_z = -118.32 \text{ kN-m}$

Nodo 12: $R_x = 39.89 \text{ kN}$, $R_y = 71.94 \text{ kN}$, $M_z = -117.85 \text{ kN-m}$

=== DESPLAZAMIENTOS NODALES ===

Nodo 1: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $\theta_z = 0.000e+00$ rad

Nodo 2: $U_x = -5.708e-02$ m, $U_y = -1.621e-04$ m, $\theta_z = 7.397e-03$ rad

Nodo 3: $U_x = -8.954e-02$ m, $U_y = -1.889e-04$ m, $\theta_z = 1.291e-02$ rad

Nodo 4: $U_x = -1.210e-01$ m, $U_y = -2.157e-04$ m, $\theta_z = 4.877e-03$ rad

Nodo 5: $U_x = -8.927e-02$ m, $U_y = 4.195e-02$ m, $\theta_z = 1.147e-02$ rad

Nodo 6: $U_x = -5.706e-02$ m, $U_y = -2.442e-02$ m, $\theta_z = -4.372e-03$ rad

Nodo 7: $U_x = -1.211e-01$ m, $U_y = -2.254e-02$ m, $\theta_z = -3.348e-03$ rad

Nodo 8: $U_x = -1.212e-01$ m, $U_y = -2.983e-04$ m, $\theta_z = 8.483e-03$ rad

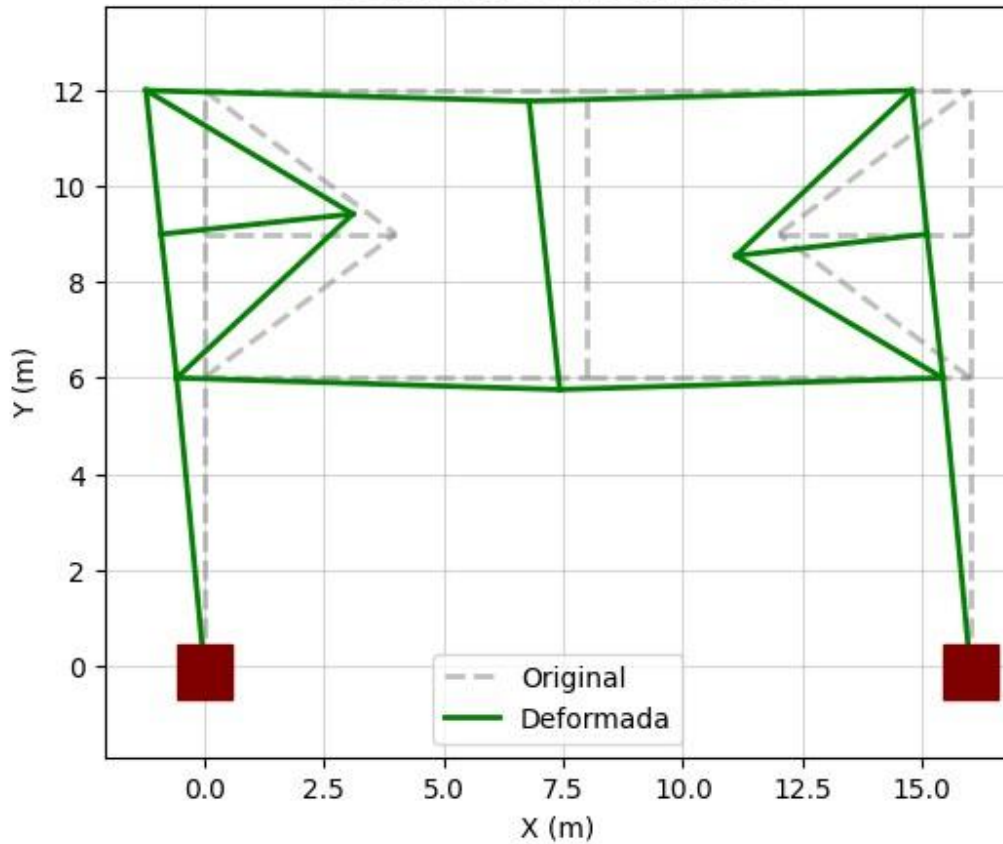
Nodo 9: $U_x = -8.969e-02$ m, $U_y = -2.456e-04$ m, $\theta_z = 1.144e-02$ rad

Nodo 10: $U_x = -8.958e-02$ m, $U_y = -4.517e-02$ m, $\theta_z = 1.162e-02$ rad

Nodo 11: $U_x = -5.692e-02$ m, $U_y = -1.928e-04$ m, $\theta_z = 1.008e-02$ rad

Nodo 12: $U_x = 0.000e+00$ m, $U_y = 0.000e+00$ m, $\theta_z = 0.000e+00$ rad

SISTEMA DEFORMADO



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 60.487 \text{ kN}$, $V = -32.541 \text{ kN}$, $M = -118.320 \text{ kN-m}$

Nodo 2: $N = -60.487 \text{ kN}$, $V = 32.541 \text{ kN}$, $M = -76.928 \text{ kN-m}$

Elemento 2:

Nodo 2: $N = 19.965 \text{ kN}$, $V = -14.868 \text{ kN}$, $M = -53.162 \text{ kN-m}$

Nodo 3: $N = -19.965 \text{ kN}$, $V = 14.868 \text{ kN}$, $M = 8.557 \text{ kN-m}$

Elemento 3:

Nodo 3: $N = 19.965 \text{ kN}$, $V = -35.679 \text{ kN}$, $M = -8.557 \text{ kN-m}$

Nodo 4: $N = -19.965 \text{ kN}$, $V = 35.679 \text{ kN}$, $M = -98.481 \text{ kN-m}$

Elemento 4:

Nodo 4: $N = -3.660 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 5: $N = 3.660 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 5:

Nodo 3: $N = -20.811 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 5: $N = 20.811 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 6:

Nodo 2: $N = 29.673 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 5: $N = -29.673 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 7:

Nodo 2: $N = -6.066 \text{ kN}$, $V = 22.719 \text{ kN}$, $M = 130.090 \text{ kN-m}$

Nodo 6: $N = 6.066 \text{ kN}$, $V = -22.719 \text{ kN}$, $M = 51.659 \text{ kN-m}$

Elemento 8:

Nodo 6: $N = -18.830 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 7: $N = 18.830 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 9:

Nodo 4: $N = 38.607 \text{ kN}$, $V = 17.769 \text{ kN}$, $M = 98.481 \text{ kN-m}$

Nodo 7: $N = -38.607 \text{ kN}$, $V = -17.769 \text{ kN}$, $M = 43.669 \text{ kN-m}$

Elemento 10:

Nodo 7: $N = 38.607 \text{ kN}$, $V = -1.061 \text{ kN}$, $M = -43.669 \text{ kN-m}$

Nodo 8: $N = -38.607 \text{ kN}$, $V = 1.061 \text{ kN}$, $M = 35.178 \text{ kN-m}$

Elemento 11:

Nodo 9: $N = 39.331 \text{ kN}$, $V = -12.418 \text{ kN}$, $M = -2.077 \text{ kN-m}$

Nodo 8: $N = -39.331 \text{ kN}$, $V = 12.418 \text{ kN}$, $M = -35.178 \text{ kN-m}$

Elemento 12:

Nodo 11: $N = 39.331 \text{ kN}$, $V = -3.701 \text{ kN}$, $M = -13.179 \text{ kN-m}$

Nodo 9: $N = -39.331 \text{ kN}$, $V = 3.701 \text{ kN}$, $M = 2.077 \text{ kN-m}$

Elemento 13:

Nodo 10: $N = -63.782 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 8: $N = 63.782 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 14:

Nodo 10: $N = 8.718 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Nodo 9: $N = -8.718 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = 0.000 \text{ kN-m}$

Elemento 15:

Nodo 11: $N = 52.885 \text{ kN}$, $V = -0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Nodo 10: $N = -52.885 \text{ kN}$, $V = 0.000 \text{ kN}$, $M = -0.000 \text{ kN-m}$

Elemento 16:

Nodo 6: $N = -48.492 \text{ kN}$, $V = -0.878 \text{ kN}$, $M = -51.659 \text{ kN-m}$

Nodo 11: $N = 48.492 \text{ kN}$, $V = 0.878 \text{ kN}$, $M = 44.638 \text{ kN-m}$

Elemento 17:

Nodo 12: $N = 71.939 \text{ kN}$, $V = -39.885 \text{ kN}$, $M = -117.852 \text{ kN-m}$

Nodo 11: $N = -71.939 \text{ kN}$, $V = 9.885 \text{ kN}$, $M = -31.459 \text{ kN-m}$

DIAGRAMA AXIAL

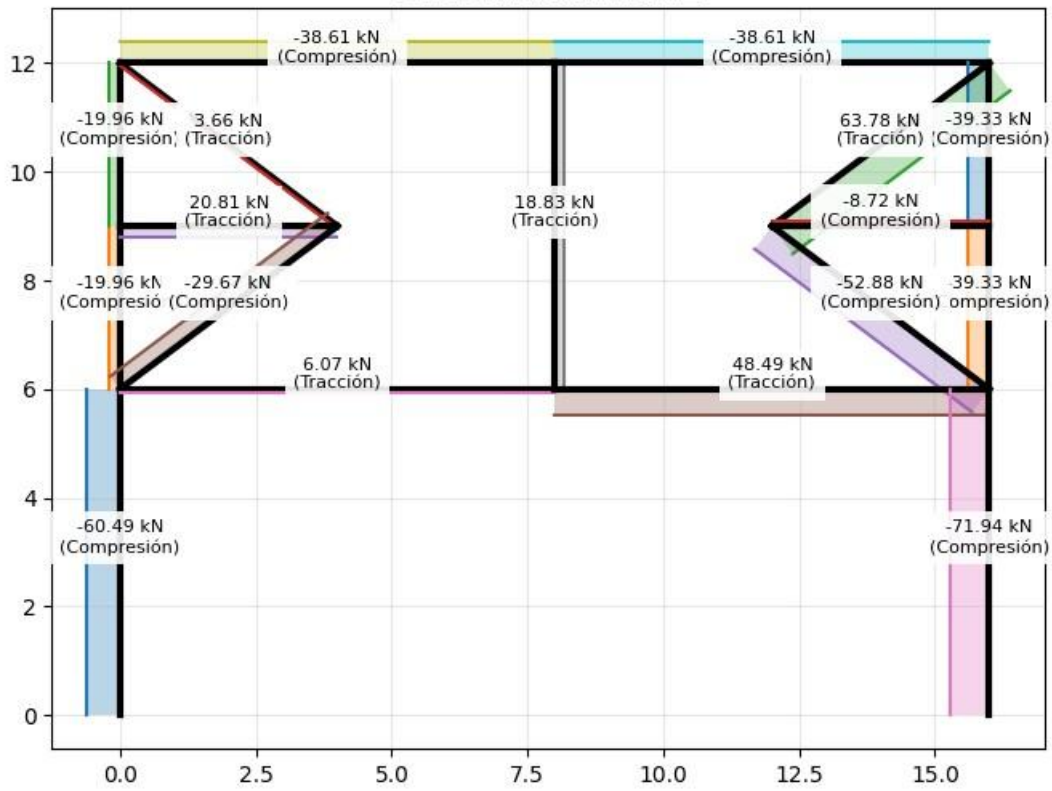


DIAGRAMA CORTANTE (kN)

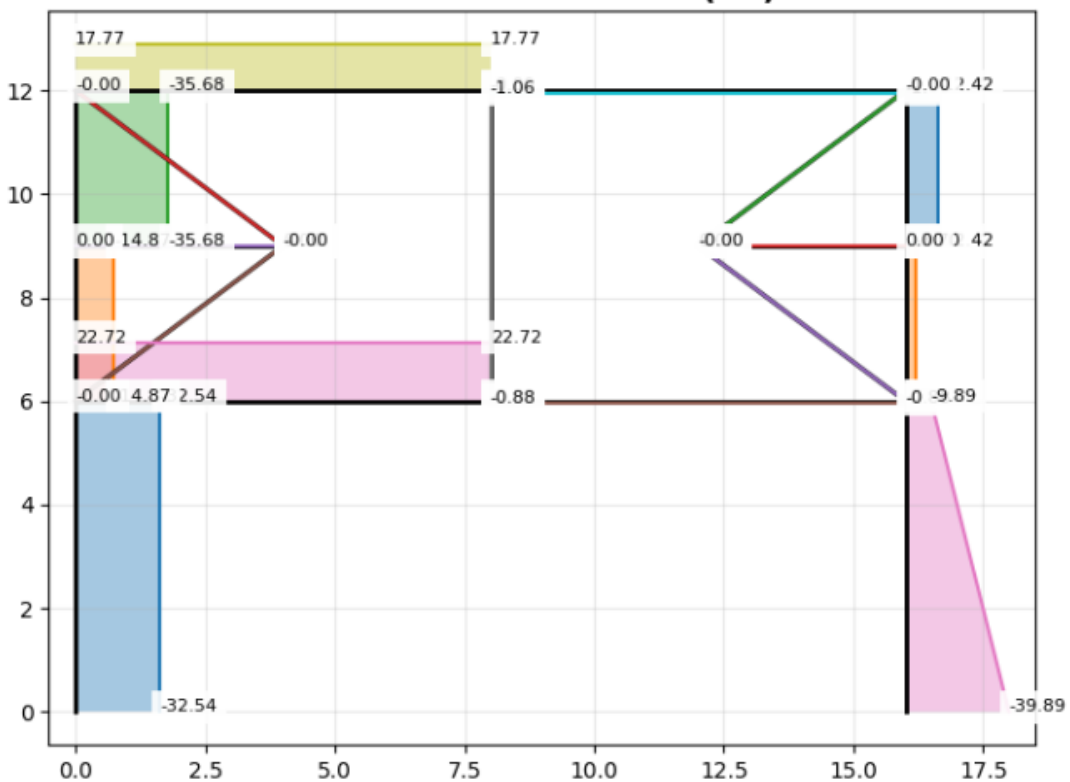
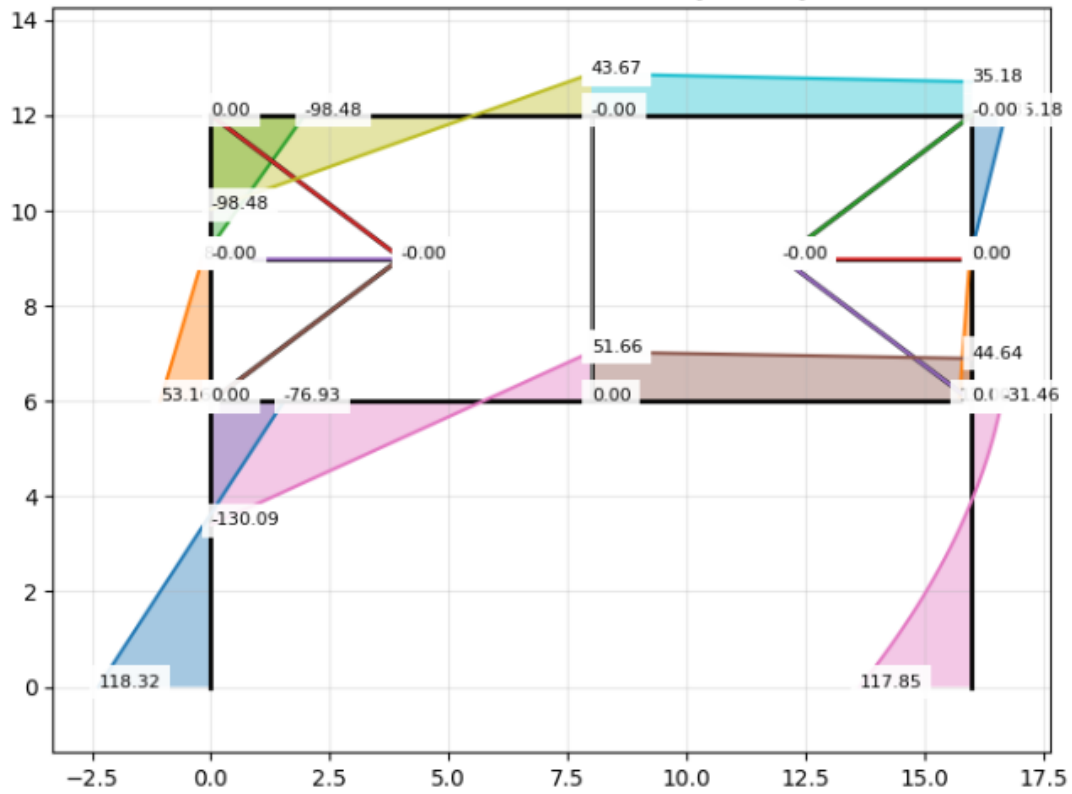


DIAGRAMA MOMENTO (kN-m)



MODELO OPENSEES

SISTEMAS MIXTOS: EJERCICIO 3

```
# -----
# IMPORTACIÓN DE LIBRERÍAS
# -----
import openseespy.opensees as ops # Biblioteca principal de OpenSees para análisis
import opsviz as opsv            # Extensión para visualización de resultados
import numpy as np               # Para operaciones numéricas y arreglos
import math                      # Para funciones matemáticas básicas
import matplotlib.pyplot as plt  # Para visualización personalizada de resultados

print("\033[1mMODELO EN OPENSEES PARA EL EJERCICIO 3 - SISTEMAS MIXTOS (17\nELEMENTOS)\033[0m")

# -----
# INICIALIZACIÓN DEL MODELO
# -----
ops.wipe() # Limpiar memoria de análisis previos (eliminar modelos anteriores)

# Modelo 2D con 3 grados de Libertad por nodo (dx, dy, dz)
ops.model("Basic", "-ndm", 2, "-ndf", 3)

# -----
# DEFINICIÓN DE NODOS
# -----
# Coordenadas en metros (X, Y)
ops.node(1, 0, 0)      # Nodo 1
ops.node(2, 0, 6)      # Nodo 2
ops.node(3, 0, 9)      # Nodo 3
ops.node(4, 0, 12)     # Nodo 4
ops.node(5, 4, 9)      # Nodo 5
ops.node(6, 8, 6)      # Nodo 6
ops.node(7, 8, 12)     # Nodo 7
ops.node(8, 16, 12)    # Nodo 8
ops.node(9, 16, 9)     # Nodo 9
ops.node(10, 12, 9)    # Nodo 10
ops.node(11, 16, 6)    # Nodo 11
ops.node(12, 16, 0)    # Nodo 12

# -----
# CONDICIONES DE CONTORNO (RESTRICCIONES)
# -----
# --- Apoyos principales: empotramientos ---
ops.fix(1, 1, 1, 1)    # Nodo 1: empotrado (Dx, Dy, Dz fijos)
ops.fix(12, 1, 1, 1)   # Nodo 12: empotrado (Dx, Dy, Dz fijos)
```



```
# -----
# DEFINICIÓN DE MATERIALES
# -----
Ec = 24870062.32      # Módulo de elasticidad del hormigón (kN/m²)
Es = 2e8              # Módulo de elasticidad del acero (kN/m²)

# -----
# DEFINICIÓN DE SECCIONES
# -----
# Las secciones se definen con el módulo de elasticidad directamente
# Formato: ops.section('Elastic', tag, E, A, I)

# --- Sección de VIGA (hormigón) ---
Av = 0.105            # Área (m²)
Iv = 0.001071875      # Inercia (m⁴)
ops.section('Elastic', 1, Ec, Av, Iv)

# --- Sección de COLUMNA (hormigón) ---
Ac = 0.09             # Área (m²)
Ic = 0.000675         # Inercia (m⁴)
ops.section('Elastic', 2, Ec, Ac, Ic)

# --- Sección de BARRA (acero) ---
Ab = 0.001536         # Área (m²)
Ib = 1e-16            # Inercia despreciable (solo trabaja axialmente)
ops.section('Elastic', 3, Es, Ab, Ib)

# --- Sección de CABLE (acero) ---
Acb = 0.001           # Área (m²)
Icb = 1e-16           # Inercia despreciable
ops.section('Elastic', 4, Es, Acb, Icb)

# --- Sección de RESORTE (acero) ---
Ar = 0.0003           # Área (m²) - para rigidez equivalente
Ir = 1e-16            # Inercia despreciable
ops.section('Elastic', 5, Es, Ar, Ir)

# -----
# TRANSFORMACIÓN GEOMÉTRICA
# -----
# Transformación lineal para elementos viga-columna
# Define la orientación y comportamiento geométrico de los elementos
ops.geomTransf("Linear", 1) # Transformación única para todos los elementos
```

```
# -----
# ESQUEMA DE INTEGRACIÓN
# -----
# Integración de Lobatto a lo largo del elemento para elementos viga-columna
# Parámetros: ('Lobatto', tag_integración, tag_sección, puntos_integración)
ops.beamIntegration('Lobatto', 1, 1, 10) # Para vigas (10 puntos de integración)
ops.beamIntegration('Lobatto', 2, 2, 10) # Para columnas (10 puntos de integración)
ops.beamIntegration('Lobatto', 3, 3, 10) # Para barras (10 puntos de integración)
ops.beamIntegration('Lobatto', 4, 4, 10) # Para cables (10 puntos de integración)
ops.beamIntegration('Lobatto', 5, 5, 10) # Para resorte (10 puntos de integración)

# -----
# DEFINICIÓN DE ELEMENTOS
# -----
Elementos = [] # Lista para almacenar información de elementos

# --- ELEMENTOS dispBeamColumn ---
# Todos los elementos se modelan como dispBeamColumn (viga-columna con desplazamientos)
# Unos tienen inercia real (vigas y columnas) y otros inercia despreciable (barras,
# cables, resortes)

# Vigas y Columnas (elementos con rigidez flexional)
ops.element('dispBeamColumn', 1, 1, 2, 1, 2) # Elemento 1: Columna
Elementos.append({"ID": 1, "Nodo_i": 1, "Nodo_j": 2})

ops.element('dispBeamColumn', 2, 2, 3, 1, 2) # Elemento 2: Columna
Elementos.append({"ID": 2, "Nodo_i": 2, "Nodo_j": 3})

ops.element('dispBeamColumn', 3, 3, 4, 1, 2) # Elemento 3: Columna
Elementos.append({"ID": 3, "Nodo_i": 3, "Nodo_j": 4})

ops.element('dispBeamColumn', 7, 2, 6, 1, 1) # Elemento 7: Viga
Elementos.append({"ID": 7, "Nodo_i": 2, "Nodo_j": 6})

ops.element('dispBeamColumn', 9, 4, 7, 1, 1) # Elemento 9: Viga
Elementos.append({"ID": 9, "Nodo_i": 4, "Nodo_j": 7})

ops.element('dispBeamColumn', 10, 7, 8, 1, 1) # Elemento 10: Viga
Elementos.append({"ID": 10, "Nodo_i": 7, "Nodo_j": 8})

ops.element('dispBeamColumn', 11, 9, 8, 1, 2) # Elemento 11: Columna
Elementos.append({"ID": 11, "Nodo_i": 9, "Nodo_j": 8})

ops.element('dispBeamColumn', 12, 11, 9, 1, 2) # Elemento 12: Columna
Elementos.append({"ID": 12, "Nodo_i": 11, "Nodo_j": 9})

ops.element('dispBeamColumn', 16, 6, 11, 1, 1) # Elemento 16: Viga
Elementos.append({"ID": 16, "Nodo_i": 6, "Nodo_j": 11})
```

```
ops.element('dispBeamColumn', 17, 12, 11, 1, 2) # Elemento 17: Columna
Elementos.append({"ID": 17, "Nodo_i": 12, "Nodo_j": 11})

# --- ELEMENTOS tipo BARRA/CABLE/RESORTE (inercia despreciable) ---
# Aunque son dispBeamColumn, su inercia es ~0, por lo que se comportan como elementos
# axiales
ops.element('dispBeamColumn', 4, 4, 5, 1, 4) # Elemento 4: Cable
Elementos.append({"ID": 4, "Nodo_i": 4, "Nodo_j": 5})

ops.element('dispBeamColumn', 5, 3, 5, 1, 3) # Elemento 5: Barra
Elementos.append({"ID": 5, "Nodo_i": 3, "Nodo_j": 5})

ops.element('dispBeamColumn', 6, 2, 5, 1, 3) # Elemento 6: Barra
Elementos.append({"ID": 6, "Nodo_i": 2, "Nodo_j": 5})

ops.element('dispBeamColumn', 8, 6, 7, 1, 5) # Elemento 8: Resorte
Elementos.append({"ID": 8, "Nodo_i": 6, "Nodo_j": 7})

ops.element('dispBeamColumn', 13, 10, 8, 1, 4) # Elemento 13: Cable
Elementos.append({"ID": 13, "Nodo_i": 10, "Nodo_j": 8})

ops.element('dispBeamColumn', 14, 10, 9, 1, 3) # Elemento 14: Barra
Elementos.append({"ID": 14, "Nodo_i": 10, "Nodo_j": 9})

ops.element('dispBeamColumn', 15, 10, 11, 1, 3) # Elemento 15: Barra
Elementos.append({"ID": 15, "Nodo_i": 10, "Nodo_j": 11})

# -----
# DEFINICIÓN DE CARGAS
# -----
ops.timeSeries("Linear", 1) # Serie temporal lineal (carga proporcional al factor de tiempo)
ops.pattern("Plain", 1, 1) # Patrón de carga estático (tag=1, timeSeries=1)

# Elemento 17: Carga uniformemente distribuida en columna derecha inferior
# Carga horizontal de 5 kN/m hacia la izquierda
# El signo positivo depende de la orientación del sistema local
ops.eleLoad("-ele", 17, "-type", "-beamUniform", 5)

# Nodo 5: Carga puntual vertical
ops.load(5, 0.0, -20.0, 0.0) # -20 kN en Y (hacia abajo)

# Nodo 10: Carga puntual vertical
ops.load(10, 0.0, -70.0, 0.0) # -70 kN en Y (hacia abajo)

# Nodo 6: Carga puntual diagonal (45°)
a = math.radians(45) # Ángulo de inclinación de la carga (45° respecto a la horizontal)
Fx = -60 * math.cos(a) # Componente horizontal (negativo = hacia la izquierda)
Fy = -60 * math.sin(a) # Componente vertical (negativo = hacia abajo)
ops.load(6, Fx, Fy, 0.0) # Aplicar carga en nodo 6
```

```
# -----
# GRÁFICA SISTEMA ORIGINAL
# -----

plt.figure(figsize=(7, 5.6))
plt.title('SISTEMA ORIGINAL', fontsize=16, fontweight='bold')

# Dibujar elementos estructurales (líneas negras)
for element_data in Elementos:
    Ni = element_data["Nodo_i"]
    Nf = element_data["Nodo_j"]

    # Obtener coordenadas de los nodos del elemento
    xi, yi = ops.nodeCoord(Ni)
    xf, yf = ops.nodeCoord(Nf)

    # Dibujar la barra
    plt.plot([xi, xf], [yi, yf], 'k-', lw=2)

plt.plot([8, 8], [6, 12], ':', color='white', lw=3) # Diferenciar el elemento tipo
resorte

# Dibujar nodos como puntos azules
for i in range(1, 13):
    x, y = ops.nodeCoord(i)
    plt.plot(x, y, 'o', color='navy', markersize=7, zorder=2)

# Dibujar apoyos empotrados (cuadrados marrones)
plt.plot(0, -0.15, 's', color='maroon', markersize=20, zorder=3) # Nodo 1
plt.plot(16, -0.15, 's', color='maroon', markersize=20, zorder=3) # Nodo 12

# --- Dibujo de cargas (solo visualización gráfica) ---
# Carga distribuida en el elemento 17 (columna derecha inferior)
xi, yi = ops.nodeCoord(12) # Nodo 12 (inferior)
xf, yf = ops.nodeCoord(11) # Nodo 11 (superior)

x_vals = np.linspace(xi, xf, 15)
y_vals = np.linspace(yi, yf, 15)

# Flechas horizontales representan carga distribuida horizontal
for x, y in zip(x_vals, y_vals):
    plt.arrow(x+0.8, y, -0.6, 0, head_width=0.15, head_length=0.2, fc='orange',
              ec='orange', zorder=4)

plt.plot([xi+0.8, xf+0.8], [yi, yf], '--', color='orange', linewidth=2, zorder=3)
plt.text(13.5, 3, '5 kN/m', color='orange', fontsize=11, fontweight='bold', zorder=5)
```

```
# Carga puntual en nodo 5 (-20 kN)
plt.arrow(4, 9, 0, -1.8, head_width=0.1, head_length=0.2, fc='green', ec='green',
linewidth=3, zorder=4)
plt.text(4.2, 7.8, '-20 kN', color='green', fontweight='bold', fontsize=11, zorder=5)

# Carga puntual en nodo 10 (-70 kN)
plt.arrow(12, 9, 0, -1.5, head_width=0.1, head_length=0.2, fc='green', ec='green',
linewidth=3, zorder=4)
plt.text(9.8, 7.8, '-70 kN', color='green', fontweight='bold', fontsize=11, zorder=5)

# Carga diagonal en nodo 6 (-60 kN a 45°)
plt.arrow(8, 6, -1.8, -1.5, head_width=0.1, head_length=0.2, fc='green', ec='green',
linewidth=3, zorder=4)
plt.text(6.2, 4, '-60 kN', color='green', fontweight='bold', fontsize=11, zorder=5)

# Configuración del gráfico
plt.xlabel('X (m)', fontsize=12)
plt.ylabel('Y (m)', fontsize=12)
plt.grid(True, alpha=0.3, zorder=0)
plt.axis('equal')
plt.tight_layout()
plt.show()

# -----
# CONFIGURACIÓN DEL ANÁLISIS
# -----
ops.system("BandSPD") # Solver para matrices banda simétrica definida positiva
ops.numberer("RCM") # Numeración de grados de libertad optimiza el ancho de
banda
ops.constraints("Plain") # Imposición directa de restricciones (método de eliminación)
ops.integrator("LoadControl", 1.0) # Control por carga: aplica el 100% de la carga
en 1 paso
ops.algorithm("Linear") # Algoritmo de solución lineal (suficiente para análisis
elástico lineal)
ops.analysis("Static") # Tipo de análisis: estático
ops.analyze(1) # Ejecutar análisis (1 paso)

# -----
# RESULTADOS
# -----
print("\n===== RESULTADOS =====")

print("\n=== REACCIONES ===")
ops.reactions() # Calcular reacciones en los nodos con restricciones
for i in [1, 12]: # Solo nodos con apoyos
    Rx = ops.nodeReaction(i, 1) # Reacción en X (kN)
    Ry = ops.nodeReaction(i, 2) # Reacción en Y (kN)
    Rz = ops.nodeReaction(i, 3) # Momento de reacción (kN-m)
```

```

print(f"Node {i}: Rx = {Rx:.3f} kN, Ry = {Ry:.3f} kN, Mz = {Rz:.3f} kN-m")
print("\n=== DESPLAZAMIENTOS ===")
for i in range(1, 13):
    disp = ops.nodeDisp(i) # [dx, dy, dz] in meters and radians

    print(f"Node {i}: Ux = {disp[0]:.3e} m, Uy = {disp[1]:.3e} m, Uz = {disp[2]:.3e}
rad")

# -----
# GRÁFICA SISTEMA DEFORMADO
# -----
fig, ax = plt.subplots(figsize=(7, 5)) # Create figure for the deformed
plt.title('SISTEMA DEFORMADO', fontsize=14, fontweight='bold')

# Draw the deformed using opsv
# The parameter 'ax' specifies the axes where to draw
opsv.plot_defo(ax=ax)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

print("\n=== FUERZAS INTERNAS ===")
for element_data in Elementos:
    ele_id = element_data["ID"]
    Ni = element_data["Node_i"]
    Nj = element_data["Node_j"]

    fuerzas = ops.eleResponse(ele_id, 'localForces')

    print(f"Elemento {ele_id}:")

    # Elements of MARCO (with significant flexural rigidity)
    if ele_id in [1, 2, 3, 7, 9, 10, 11, 12, 16, 17]:
        # Format: [P_i, V_i, M_i, P_j, V_j, M_j]
        # P: axial force, V: shear, M: bending moment
        Pi, Vi, Mi, Pj, Vj, Mj = fuerzas

        print(f"Node {Ni}: N = {Pi:.3f} kN, V = {Vi:.3f} kN, M = {Mi:.3f} kN-m")
        print(f"Node {Nj}: N = {Pj:.3f} kN, V = {Vj:.3f} kN, M = {Mj:.3f} kN-m\n")

    # Elements type CERCHA (negligible inertia, only axial force significant)
    else:
        P = fuerzas[0] # Only axial (P_i = -P_j for equilibrium)
        # Determine type of stress according to the sign (negative = traction in OpenSees)
        if abs(P) < 1e-6:
            tipo = "Fuerza axial nula"

```

```

elif P < 0:
    tipo = "Tracción"

else:
    tipo = "Compresión"

print(f" Axial = {P:.3f} kN - {tipo}\n")

# -----
# DIAGRAMAS DE ESFUERZOS
# -----
# --- Diagrama de Fuerza Axial ---
fig_n = plt.figure(figsize=(7, 5))
plt.title('DIAGRAMA DE FUERZA AXIAL (kN)', fontsize=14, fontweight='bold')
ax_n = plt.gca()

# sf_type: tipo de fuerza ('N' = axial)
# sfac: factor de escala para visualización (ajustado para este modelo)
# nep: número de puntos para dibujar el diagrama
opsv.section_force_diagram_2d(sf_type='N', sfac=0.008, nep=20, ax=ax_n)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

# --- Diagrama de Fuerza Cortante ---
fig_v = plt.figure(figsize=(7, 5))
plt.title('DIAGRAMA DE FUERZA CORTANTE (kN)', fontsize=14, fontweight='bold')
ax_v = plt.gca()
opsv.section_force_diagram_2d(sf_type='V', sfac=0.05, nep=20, ax=ax_v)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

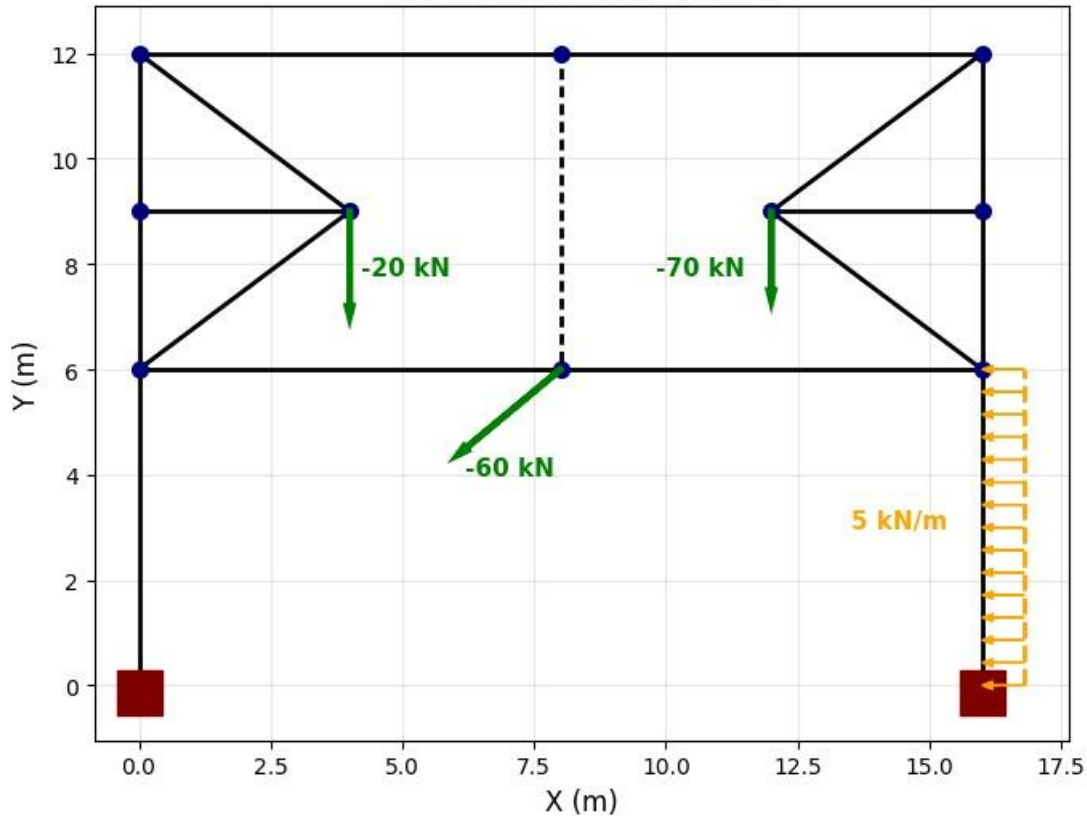
# --- Diagrama de Momento Flector ---
fig_m = plt.figure(figsize=(7, 5))
plt.title('DIAGRAMA DE MOMENTO FLECTOR (kN-m)', fontsize=14, fontweight='bold')
ax_m = plt.gca()
opsv.section_force_diagram_2d(sf_type='M', sfac=0.05, nep=20, ax=ax_m)

plt.grid(True, alpha=0.3)
plt.axis('equal')
plt.tight_layout()
plt.show()

```

MODELO EN OPENSEES PARA EL EJERCICIO 3 - SISTEMAS MIXTOS (17 ELEMENTOS)

SISTEMA ORIGINAL



===== RESULTADOS =====

=== REACCIONES ===

Nodo 1: $R_x = 32.541 \text{ kN}$, $R_y = 60.487 \text{ kN}$, $M_z = -118.320 \text{ kN-m}$

Nodo 12: $R_x = 39.885 \text{ kN}$, $R_y = 71.939 \text{ kN}$, $M_z = -117.852 \text{ kN-m}$

=== DESPLAZAMIENTOS ===

Nodo 1: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = 0.000e+00 \text{ rad}$

Nodo 2: $U_x = -5.708e-02 \text{ m}$, $U_y = -1.621e-04 \text{ m}$, $\theta_z = 7.397e-03 \text{ rad}$

Nodo 3: $U_x = -8.954e-02 \text{ m}$, $U_y = -1.889e-04 \text{ m}$, $\theta_z = 1.291e-02 \text{ rad}$

Nodo 4: $U_x = -1.210e-01 \text{ m}$, $U_y = -2.157e-04 \text{ m}$, $\theta_z = 4.877e-03 \text{ rad}$

Nodo 5: $U_x = -8.927e-02 \text{ m}$, $U_y = 4.195e-02 \text{ m}$, $\theta_z = 1.147e-02 \text{ rad}$

Nodo 6: $U_x = -5.706e-02 \text{ m}$, $U_y = -2.442e-02 \text{ m}$, $\theta_z = -4.372e-03 \text{ rad}$

Nodo 7: $U_x = -1.211e-01 \text{ m}$, $U_y = -2.254e-02 \text{ m}$, $\theta_z = -3.348e-03 \text{ rad}$

Nodo 8: $U_x = -1.212e-01 \text{ m}$, $U_y = -2.983e-04 \text{ m}$, $\theta_z = 8.483e-03 \text{ rad}$

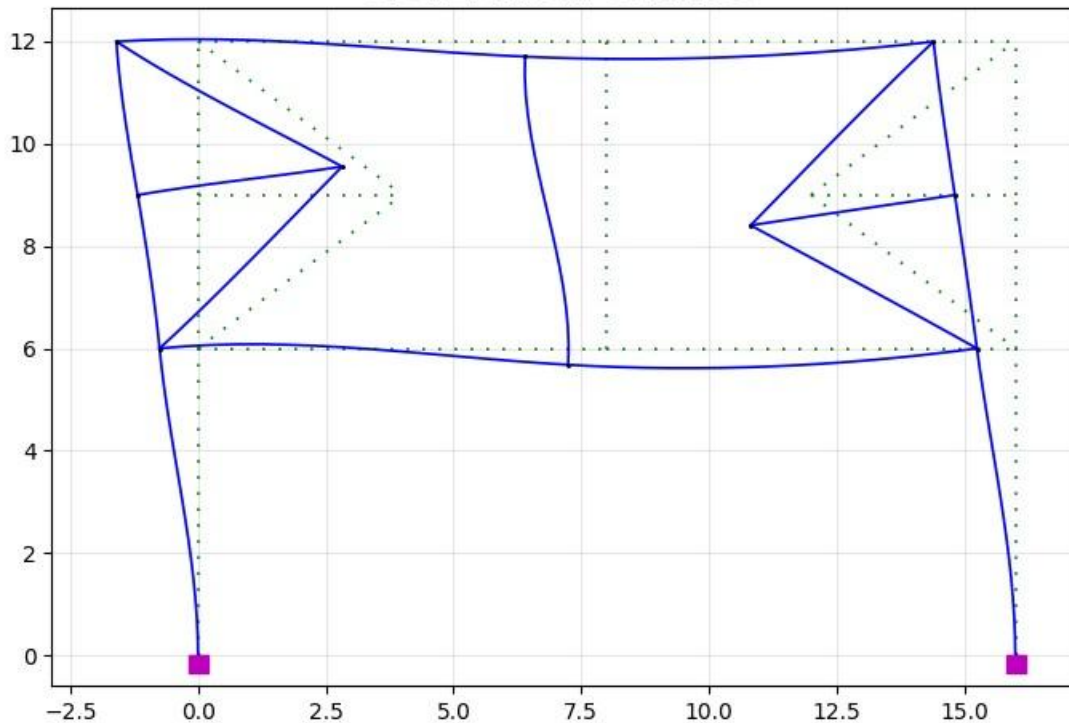
Nodo 9: $U_x = -8.969e-02 \text{ m}$, $U_y = -2.456e-04 \text{ m}$, $\theta_z = 1.144e-02 \text{ rad}$

Nodo 10: $U_x = -8.958e-02 \text{ m}$, $U_y = -4.517e-02 \text{ m}$, $\theta_z = 1.162e-02 \text{ rad}$

Nodo 11: $U_x = -5.692e-02 \text{ m}$, $U_y = -1.928e-04 \text{ m}$, $\theta_z = 1.008e-02 \text{ rad}$

Nodo 12: $U_x = 0.000e+00 \text{ m}$, $U_y = 0.000e+00 \text{ m}$, $\theta_z = 0.000e+00 \text{ rad}$

SISTEMA DEFORMADO



=== FUERZAS INTERNAS ===

Elemento 1:

Nodo 1: $N = 60.487 \text{ kN}$, $V = -32.541 \text{ kN}$, $M = -118.320 \text{ kN-m}$

Nodo 2: $N = -60.487 \text{ kN}$, $V = 32.541 \text{ kN}$, $M = -76.928 \text{ kN-m}$

Elemento 2:

Nodo 2: $N = 19.965 \text{ kN}$, $V = -14.868 \text{ kN}$, $M = -53.162 \text{ kN-m}$

Nodo 3: $N = -19.965 \text{ kN}$, $V = 14.868 \text{ kN}$, $M = 8.557 \text{ kN-m}$

Elemento 3:

Nodo 3: $N = 19.965 \text{ kN}$, $V = -35.679 \text{ kN}$, $M = -8.557 \text{ kN-m}$

Nodo 4: $N = -19.965 \text{ kN}$, $V = 35.679 \text{ kN}$, $M = -98.481 \text{ kN-m}$

Elemento 7:

Nodo 2: $N = -6.066 \text{ kN}$, $V = 22.719 \text{ kN}$, $M = 130.090 \text{ kN-m}$

Nodo 6: $N = 6.066 \text{ kN}$, $V = -22.719 \text{ kN}$, $M = 51.659 \text{ kN-m}$

Elemento 9:

Nodo 4: $N = 38.607 \text{ kN}$, $V = 17.769 \text{ kN}$, $M = 98.481 \text{ kN-m}$

Nodo 7: $N = -38.607 \text{ kN}$, $V = -17.769 \text{ kN}$, $M = 43.669 \text{ kN-m}$

Elemento 10:

Nodo 7: $N = 38.607 \text{ kN}$, $V = -1.061 \text{ kN}$, $M = -43.669 \text{ kN-m}$

Nodo 8: $N = -38.607 \text{ kN}$, $V = 1.061 \text{ kN}$, $M = 35.178 \text{ kN-m}$

Elemento 11:

Nodo 9: $N = 39.331 \text{ kN}$, $V = -12.418 \text{ kN}$, $M = -2.077 \text{ kN-m}$

Nodo 8: $N = -39.331 \text{ kN}$, $V = 12.418 \text{ kN}$, $M = -35.178 \text{ kN-m}$

Elemento 12:

Nodo 11: $N = 39.331 \text{ kN}$, $V = -3.701 \text{ kN}$, $M = -13.179 \text{ kN-m}$

Nodo 9: $N = -39.331 \text{ kN}$, $V = 3.701 \text{ kN}$, $M = 2.077 \text{ kN-m}$

Elemento 16:

Nodo 6: $N = -48.492 \text{ kN}$, $V = -0.878 \text{ kN}$, $M = -51.659 \text{ kN-m}$

Nodo 11: $N = 48.492 \text{ kN}$, $V = 0.878 \text{ kN}$, $M = 44.638 \text{ kN-m}$

Elemento 17:

Nodo 12: $N = 71.939 \text{ kN}$, $V = -39.885 \text{ kN}$, $M = -117.852 \text{ kN-m}$

Nodo 11: $N = -71.939 \text{ kN}$, $V = 9.885 \text{ kN}$, $M = -31.459 \text{ kN-m}$

Elemento 4:

Axial = -3.660 kN - Tracción

Elemento 5:

Axial = -20.811 kN - Tracción

Elemento 6:

Axial = 29.673 kN - Compresión

Elemento 8:

Axial = -18.830 kN - Tracción

Elemento 13:

Axial = -63.782 kN - Tracción

Elemento 14:

Axial = 8.718 kN - Compresión

Elemento 15:

Axial = 52.885 kN - Compresión

DIAGRAMA DE FUERZA AXIAL (kN)

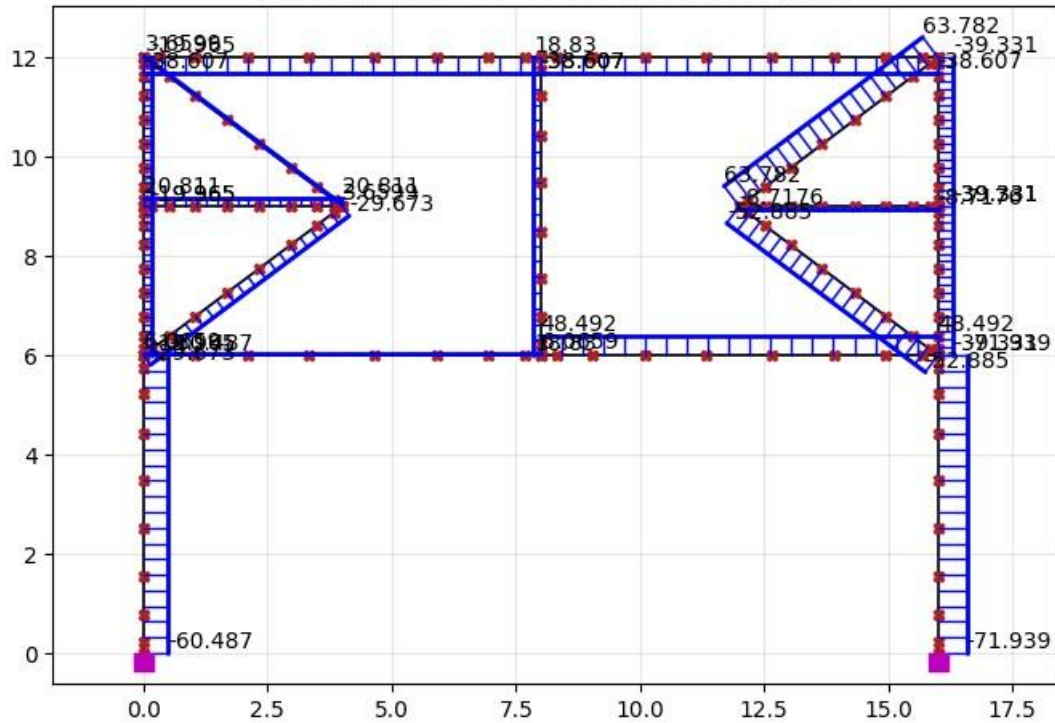


DIAGRAMA DE FUERZA CORTANTE (kN)

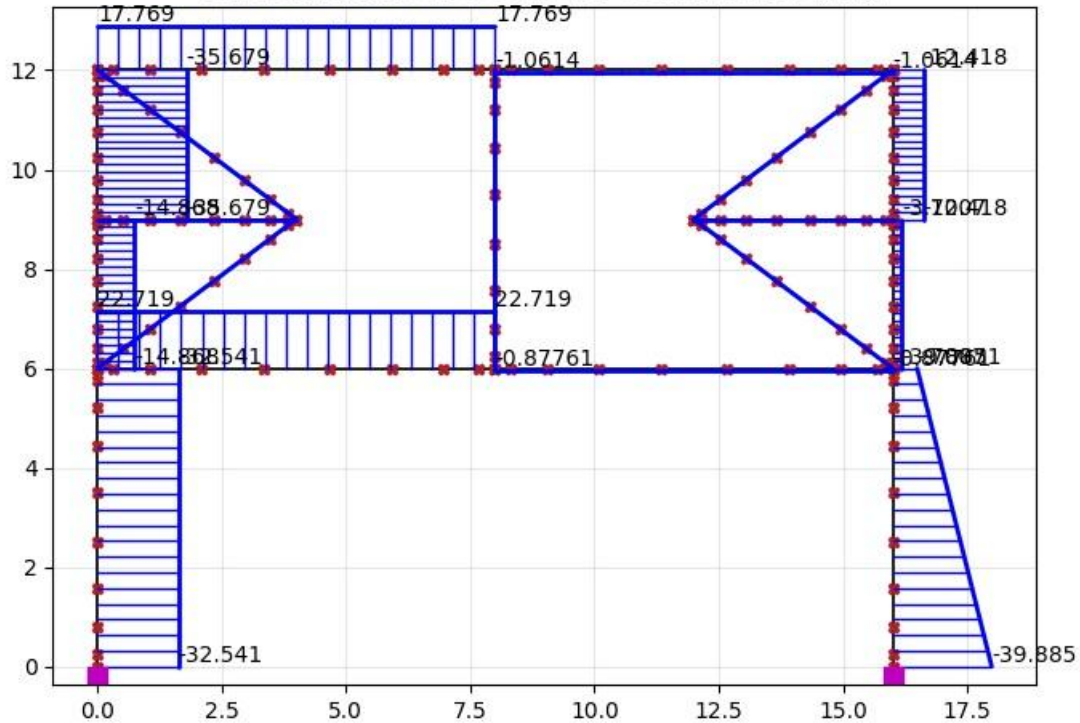


DIAGRAMA DE MOMENTO FLECTOR (kN-m)

