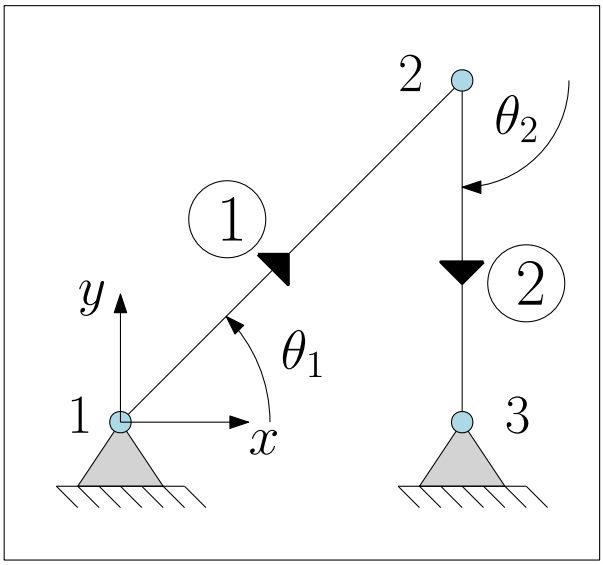
Mesh structure

The structure is composed of two bars. There are two degrees of freedom for each node : the displacement along the horizontal and vertical direction. The structure is isostatic and the bars only work in traction-compression.

The node are numbered according to the following picture:



The global reference frame is located in correspondance of node 1.

Two information are required to construct the mesh

1. The table of nodal coordinates is the following

Global node number	x	y
1	0	0
2	L	L
3	L	0

2. The table of connectivity

Element	**Left node (1)**	**Right Node (2)**
1	1	2
2	2	3

The table of connectivity describes the topology of the mesh, i.e. how the elements are associated to the nodes and how they are oriented.

The orientation provides the information on the angle by which a local frame is rotated with respect to the global frame. The x axis of local frame is aligned along an element and oriented from the left to right node. For this case the angles are $\theta_1 = \pi/4$ and $\theta_2 = -\pi/2$.

N.B. Different types of elements may be contained in a single mesh (1D, 2D, 3D).

The coordinates and connectivity table can be declared by the following code (remember that in python the indexing starts at 0):

In [1]:
```
import numpy as np

L = 1 # length given by the data

coordinates = np.array([[0, 0],
                        [L, L],
                        [L, 0]])

connectivity_table = np.array([[0, 1],
                               [1, 2]])
```

Forces and nodal displacement

The vector of nodal displacement will contain the horizontal $u_i,\ i \in \{1, 2, 3\}$ and vertical $v_i,\ i \in \{1, 2, 3\}$ displacements. The displacements at nodes 1 and 3 are zero because of the boundary conditions. An exernal forces is applied on node 2 and reaction forces enforce the boundary conditions on node 1 and 3.

$$\mathbf{q} = \begin{pmatrix} u_1 = 0 \\ v_1 = 0 \\ u_2 \\ v_2 \\ u_3 = 0 \\ v_3 = 0 \end{pmatrix}, \qquad \mathbf{f} = \begin{pmatrix} X_1 \\ Y_1 \\ F \\ 0 \\ X_3 \\ Y_3 \end{pmatrix}$$

The finite element system is constructed considering the minimization of the total elastic energy (principle of virtual work):

$$\delta E = \delta W_{\text{ext}}.$$

Upon introduction of the finite element discretization, this translate into the following algebraic system

$$\boldsymbol{\delta q}^\top \mathbf{K q} = \boldsymbol{\delta q}^\top \mathbf{f}, \quad \forall \boldsymbol{\delta q} \qquad \Longrightarrow \mathbf{K q} = \mathbf{f}.$$

Local stiffness matrix construction

In this example the energy is the traction deformation energy, given b

$$E_{\mathrm{ax}} = \sum_{1}^{n_{\mathrm{el}}} \int_0^{L_e} EA_e \left(\frac{du^l}{dx^l} \right)^2 dx^l$$

where u^l is the axial displacement and x^l is the local coordinate (aligned with the element axis). The axial displacement field is approximated using Lagrange polynomials in the local reference frame as follows

$$u^l(x^l) = \begin{pmatrix} 1 - x^l/L_e & x^l/L_e \end{pmatrix} \begin{pmatrix} u_1^l \\ u_2^l \end{pmatrix} = \mathbf{N}\mathbf{u}^l.$$

where L_e is the length of the element under consideration and l stands for local frame. The local stiffness matrix is computed as

$$\mathbf{K}_e^l = \int_0^{L_e} EA_e \frac{\mathrm{d}\mathbf{N}^\top}{\mathrm{dx^l}} \frac{\mathrm{d}\mathbf{N}}{\mathrm{dx^l}} \mathrm{dx^l}$$

In finite elements technology, the description of a finite element is typically given in a simple reference configuration. For truss elements, reference element is a segment of unitary length. The reference element coordinate is simply given by $\xi = L_e^{-1} x^l$. The bases functions rewrite as follows

$$u(\xi) = \begin{pmatrix} 1 - \xi & \xi \end{pmatrix} \begin{pmatrix} u_1^l \\ u_2^l \end{pmatrix}$$

The stiffness matrix for a generic element in its local reference frame, is computed using the reference coordinate by taking into account the Jacobian of the transformation ($J = \frac{\mathrm{dx^l}}{\mathrm{d}\xi} = L_e$) and the chain rule

$$\mathbf{K}_e^l = \int_0^1 EA_e \frac{\mathrm{d}\mathbf{N}^\top}{\mathrm{d}\xi} \frac{\mathrm{d}\mathbf{N}}{\mathrm{d}\xi} J^{-1} \mathrm{d}\xi = \frac{\mathrm{EA_e}}{\mathrm{L_e}} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}. \qquad \text{This is valid in 1D.}$$

The inverse of the Jacobian appear because of the chain rule in the computation of the derivative of the basis function $\frac{\mathrm{d}\mathbf{N}}{\mathrm{dx_l}} = \frac{\mathrm{d}\mathbf{N}}{\mathrm{d}\xi} \frac{\mathrm{d}\xi}{\mathrm{dx_l}} = \frac{\mathrm{d}\mathbf{N}}{\mathrm{d}\xi} J^{-1}$.

Since the problem is 2D, both longitudinal and trasversal displacement need to be accounted for. Since this is a truss structure (bars only work in traction compression) there is no need to account for the bending stiffness, (the pivots will not transmit torques).

$$\mathbf{K}_e^l = \int_0^1 EA_e \frac{\mathrm{d}\mathbf{N}^\top}{\mathrm{d}\xi} \frac{\mathrm{d}\mathbf{N}}{\mathrm{d}\xi} J^{-1} \mathrm{d}\xi = \frac{\mathrm{EA_e}}{\mathrm{L_e}} \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

From local to global stiffness matrix

To obtain the global stiffness matrix, the local stiffness matrix must be transformed to the global reference frame. This is achieve by considering a rotation from the local coordinates to the global coordinates. If the local frame of element e is rotated by an angle θ_e then the rotation matrix is given by

$$\mathbf{R}_e^{l\to g} = \begin{bmatrix} \cos(\theta_e) & -\sin(\theta_e) \\ \sin(\theta_e) & \cos(\theta_e) \end{bmatrix}$$

This matrix converts the local to the global coordinates

$$\begin{pmatrix} u^g \\ v^g \end{pmatrix} = \mathbf{R}_e^{l\to g} \begin{pmatrix} u^l \\ v^l \end{pmatrix}.$$

The left and the right nodes are collected in vector

$$\mathbf{q}_e^l = \begin{pmatrix} u_1^l & v_1^l & u_2^l & v_2^l \end{pmatrix}^\top.$$

The rotation needs to be applied to both nodes, so two rotation matrices are piled to obtain the displacement of the nodes in the global reference frame

$$\mathbf{q}_e^g = \mathbf{T}_e^{l\to g} \mathbf{q}_e^l, \qquad \text{where} \quad \mathbf{T}_e^{l\to g} := \begin{bmatrix} \mathbf{R}_e^{l\to g} & 0 \\ 0 & \mathbf{R}_e^{l\to g} \end{bmatrix}$$

The conversion from the local to the global stiffness matrix is obtained considering the fact the variation of the energy is the same in both references

$$[\boldsymbol{\delta}\mathbf{q}_e^l]^\top \mathbf{K}_e^l \mathbf{q}_e^l = [\boldsymbol{\delta}\mathbf{q}_e^g]^\top \mathbf{K}_e^g \mathbf{q}_e^g$$

This implies the following transformation rule for the stiffness matrix

$$\mathbf{K}_e^g = \mathbf{T}_e^{l\to g} \mathbf{K}_e^l [\mathbf{T}_e^{l\to g}]^\top$$

Deloping the computation one obtains

$$\mathbf{K}_e^g = \frac{EA_e}{L_e}\begin{bmatrix} \cos^2(\theta_e) & \cos(\theta_e)\sin(\theta_e) & -\cos^2(\theta_e) & -\cos(\theta_e)\sin(\theta_e) \\ & \sin^2(\theta_e) & -\cos(\theta_e)\sin(\theta_e) & -\sin^2(\theta_e) \\ & & \cos^2(\theta_e) & \cos(\theta_e)\sin(\theta_e) \\ \text{sym.} & & & \sin^2(\theta_e) \end{bmatrix}$$

The following python code compute the stiffness matrix given the coordinates of the two nodes of an element and its axial stiffness.

In [2]:
```python
def truss_2d_element(coord1, coord2, EA):
    """
    Compute the element stiffness matrix for a 2D truss bar in global coordi
    Function translated to python from the MATLAB code in https://people.duk

    Parameters
    ----------
    coord1 : 2*1 array
        x1, y1 coordinates of the first node
    coord2 : 2*1 array
        x2, y2 coordinates of the second node
    EA : float
        axial stiffness of the bar

    Returns
    -------
    K : numpy.ndarray

    """

    L = np.linalg.norm(coord2 - coord1) # length of the bar

    x1, y1 = coord1
    x2, y2 = coord2

    c = ( x2 - x1 ) / L # cosine of bar angle
    s = ( y2 - y1 ) / L # sine of bar angle

    theta = np.arctan2(s, c)

    K = EA/L * np.array([[c**2, c*s, - c**2, - c*s],
                         [c * s, s**2,  - c*s, - s**2],
                         [-c**2, - c*s, c**2, c*s],
                         [ - c*s, - s**2, c*s, s**2 ] ] )
    return K, theta
```

Once the global stiffness matrix for each element is obtained the assembly of the global stiffness matrix is obtained by considering the connectivity table.

For instance for element 1, the local nodes correspond to the global nodes 1, 2.

$$\mathbf{K}_1 = \begin{bmatrix} [\mathbf{K}_1^g]_{11} & [\mathbf{K}_1^g]_{12} & [\mathbf{K}_1^g]_{13} & [\mathbf{K}_1^g]_{14} & 0 & 0 \\ [\mathbf{K}_1^g]_{21} & [\mathbf{K}_1^g]_{22} & [\mathbf{K}_1^g]_{23} & [\mathbf{K}_1^g]_{24} & 0 & 0 \\ [\mathbf{K}_1^g]_{31} & [\mathbf{K}_1^g]_{32} & [\mathbf{K}_1^g]_{33} & [\mathbf{K}_1^g]_{34} & 0 & 0 \\ [\mathbf{K}_1^g]_{41} & [\mathbf{K}_1^g]_{42} & [\mathbf{K}_1^g]_{43} & [\mathbf{K}_1^g]_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where $[\mathbf{K}_e^g]_{ij}$ denotes the ij component of the stiffess matrix of element e.

For element 2 the local nodes correspond to the global nodes 2, 3

$$\mathbf{K}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & [\mathbf{K}_2^g]_{11} & [\mathbf{K}_2^g]_{12} & [\mathbf{K}_2^g]_{13} & [\mathbf{K}_2^g]_{14} \\ 0 & 0 & [\mathbf{K}_2^g]_{21} & [\mathbf{K}_2^g]_{22} & [\mathbf{K}_2^g]_{23} & [\mathbf{K}_2^g]_{24} \\ 0 & 0 & [\mathbf{K}_2^g]_{31} & [\mathbf{K}_2^g]_{32} & [\mathbf{K}_2^g]_{33} & [\mathbf{K}_2^g]_{34} \\ 0 & 0 & [\mathbf{K}_2^g]_{41} & [\mathbf{K}_2^g]_{42} & [\mathbf{K}_2^g]_{43} & [\mathbf{K}_2^g]_{44} \end{bmatrix}$$

Another way of getting this result is for energy invariance one can get:

$$[\boldsymbol{\delta}\mathbf{q}_e^l]^\top \mathbf{K}_e^l \mathbf{q}_e^l = [\boldsymbol{\delta}\mathbf{q}_e^g]^\top \mathbf{K}_e^g \mathbf{q}_e^g = [\boldsymbol{\delta}\mathbf{q}]^\top \mathbf{K}_e \mathbf{q}$$

Considering the relationships between the element global degrees of freedom and the truss degrees of freedom vector.

$$\mathbf{q}_e^g = \mathbf{T}_e \mathbf{q}$$

Which implies:

$$\mathbf{K}_e = \mathbf{T}_e^\top \mathbf{K}_e^g \mathbf{T}_e = \mathbf{T}_e^\top \mathbf{T}_e^{l\to g} \mathbf{K}_e^l [\mathbf{T}_e^{l\to g}]^\top \mathbf{T}_e$$

With :

$$\mathbf{T}_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \qquad \mathbf{T}_2 = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Summarizing

In summary to solve a 2D truss problem, one needs to:

- establish a global numbering of the nodes;
- decide the orientation of each element;

- for each find the stiffness matrix in global coordinates;
- assemble the overall stiffness matrix considering the global numbering of dofs.

Costruction of the stiffness matrix in python

Recall the data for the problem

- $E = 210$ [GPa] (steel) for each element;
- $A = 4 \times 10^{-4} \, [\mathrm{m}^2], \; \mathrm{A}_1 = \sqrt{2}\mathrm{A}, \; \mathrm{A}_2 = \mathrm{A}$;
- $F = 50 \, [\mathrm{kN}]$.
- $L = 1 \, [\mathrm{m}]$.

The stiffness matrix can be obtained by looping over the elements, calling the function that construct the stiffness matrix in the global reference frame and assembling it in the overall stiffness matrix.

This procedure is illustrated by the following code:

In [3]:
```
E = 210 * 1e9
A = 4 * 1e-4

A_1 = np.sqrt(2) * A
A_2 = A

array_EA = np.array([E*A_1, E*A_2])

F = 50 * 1e3

n_elements = connectivity_table.shape[0]
n_nodes = coordinates.shape[0]
n_dofs = 2 * n_nodes
K = np.zeros((n_dofs, n_dofs))

theta_vec = np.zeros((n_elements, ))

for ii in range(n_elements):

    left_node, right_node = connectivity_table[ii]

    K_element, theta_element = truss_2d_element(coordinates[left_node],
                                coordinates[right_node],
                                array_EA[ii])

    theta_vec[ii] = theta_element

    dof_left = 2*left_node
    dof_right = 2*right_node
```

```
    K[dof_left:dof_left+2, dof_left:dof_left+2] += K_element[0:2,0:2]
    K[dof_right:dof_right+2, dof_right:dof_right+2] += K_element[2:4,2:4]

    K[dof_left:dof_left+2, dof_right:dof_right+2] += K_element[0:2,2:4]
    K[dof_right:dof_right+2, dof_left:dof_left+2] += K_element[2:4,0:2]


print(f"Stiffness matrix normalized by axial stiffness: \n{K/(E*A/(L))}\n")
```

```
Stiffness matrix normalized by axial stiffness:
[[ 0.5  0.5 -0.5 -0.5  0.   0. ]
 [ 0.5  0.5 -0.5 -0.5  0.   0. ]
 [-0.5 -0.5  0.5  0.5  0.   0. ]
 [-0.5 -0.5  0.5  1.5  0.  -1. ]
 [ 0.   0.   0.   0.   0.   0. ]
 [ 0.   0.   0.  -1.   0.   1. ]]
```

Use sparse format for stiffness matrix assembly

Given the sparse nature of Stiffness Matrix, one can use an assemble it using a sparse format see the scipy library documentation here. This representation is computationnally less expensive in terms of memory required to store it. Matrix sparsity can be used by the solver to accelerate the problem resolution.

In [4]:
```
from scipy.sparse import csr_array

# This may be constructed automatically in the following loop
elementary_dofs = np.array([[0,1,2,3],[2,3,4,5]])

I_vec,J_vec,K_vec = [], [], []
for ii in range(n_elements):

    left_node, right_node = connectivity_table[ii]

    K_element, theta_element = truss_2d_element(coordinates[left_node],
                                coordinates[right_node],
                                array_EA[ii])
    i,j = np.nonzero(K_element)
    k = K_element[i,j]
    I_vec+=elementary_dofs[ii,i].tolist()
    J_vec+=elementary_dofs[ii,j].tolist()
    K_vec+=k.tolist()

# csr (compressed sparse raw) is a particular format of sparse matrix and
# its particularly efficient for matrices. Contiguous elements in a row are
# stored contiguosly in memory

K = csr_array((K_vec,(I_vec,J_vec)),shape=(n_dofs,n_dofs))
print(f"Stiffness matrix normalized by axial stiffness: \n{K/(E*A/(L))}\n")
```

```
Stiffness matrix normalized by axial stiffness:
  (0, 0)        0.49999999999999994
  (0, 1)        0.49999999999999994
  (0, 2)        -0.49999999999999994
  (0, 3)        -0.49999999999999994
  (1, 0)        0.49999999999999994
  (1, 1)        0.49999999999999994
  (1, 2)        -0.49999999999999994
  (1, 3)        -0.49999999999999994
  (2, 0)        -0.49999999999999994
  (2, 1)        -0.49999999999999994
  (2, 2)        0.49999999999999994
  (2, 3)        0.49999999999999994
  (3, 0)        -0.49999999999999994
  (3, 1)        -0.49999999999999994
  (3, 2)        0.49999999999999994
  (3, 3)        1.5
  (3, 5)        -1.0
  (5, 3)        -1.0
  (5, 5)        1.0
```

Solving the FEM problem in python

To solve the FEM problem the external forces and the boundary conditions are applied. The global equilibrium equations can be used:

$$\mathbf{K}\mathbf{q} = \mathbf{f} + \mathbf{r}$$

Where $\mathbf{r}$ is the vector of reaction forces. One can partition the full system in free degrees of freedom *red* and the constrained degrees of freedom *fixed*.

$$\begin{bmatrix} \mathbf{K}_{\text{red}} & \mathbf{K}_{\text{red,fixed}} \\ \mathbf{K}_{\text{fixed,red}} & \mathbf{K}_{\text{fixed,fixed}} \end{bmatrix} \begin{pmatrix} \mathbf{q}_{\text{red}} \\ \mathbf{q}_{\text{fixed}} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_{\text{red}} \\ \mathbf{0} \end{pmatrix} + \begin{pmatrix} \mathbf{0} \\ \mathbf{r}_{\text{fixed}} \end{pmatrix}$$

With the first block of equation one can get the solution in terms of displacements:

$$\mathbf{K}_{\text{red}}\mathbf{q}_{\text{red}} = \mathbf{f}_{\text{red}} - \mathbf{K}_{\text{red,fixed}}\mathbf{q}_{\text{fixed}}$$

With $\mathbf{q}_{\mathbf{fixed}} = \mathbf{0}$ in our example. In other terms the solution is obtained by solving the linear system of equations after eliminating the degrees of freedom that are fixed. In these example the nodes undergoing boundary conditions are the nodes 1 and 3. Therefore the dofs 1,2,5,6 are eliminated.

So the Stiffness matrix without the constrained nodes is

$$\mathbf{K}_{\text{red}} = \begin{bmatrix} [\mathbf{K}]_{33} & [\mathbf{K}]_{34} \\ [\mathbf{K}]_{43} & [\mathbf{K}]_{44} \end{bmatrix}$$

The corresponding force vector is simpy given by $\mathbf{f}_{\text{red}} = \begin{pmatrix} F & 0 \end{pmatrix}^\top$

In [5]:
```
from scipy.sparse.linalg import spsolve
nodes_bcs = [0, 2]
dofs_bcs = [2*nodes_bcs[i] for i in range(len(nodes_bcs))] \
         + [2*nodes_bcs[i]+1 for i in range(len(nodes_bcs))]
dofs_bcs.sort()
dofs = np.arange(n_dofs)

dofs_no_bcs = list(set(dofs) - set(dofs_bcs))
K_red = K[np.ix_(dofs_no_bcs, dofs_no_bcs)]
f_red = np.array([F, 0])
q_red = spsolve(K_red, f_red)

print(f"Normalized displacement at node 2: \n {q_red/(F*L/(E*A))} \n")
```

```
Normalized displacement at node 2:
 [ 3. -1.]
```

Postprocessing: computation of the reaction forces, normal stresses and forces

With the second block of equations of global equilibrium one can get by back sobstitution the value of the unknown reaction forces:

$$\mathbf{R}_{\text{fixed}} = \mathbf{K}_{\text{fixed,red}}\mathbf{q}_{\text{red}} + \mathbf{K}_{\text{fixed,fixed}}\mathbf{q}_{\text{fixed}}$$

In [6]:
```
q = np.zeros((n_dofs, ))
q[dofs_no_bcs] = q_red
f = K @ q

for node in nodes_bcs:
    reaction = f[2*node:2*node+2]
    print(f"Normalized reaction at node {node + 1}: \n {reaction/F} \n")
```

```
Normalized reaction at node 1:
 [-1. -1.]

Normalized reaction at node 3:
 [0. 1.]
```

The strain in each element can be computed via the following foruma

$$\varepsilon_x(x^l) = \frac{du^l}{dx^l} = \left(\frac{dx^l}{d\xi}\right)^{-1}\frac{du^l}{d\xi}$$

where again ξ is the reference coordinate. The first contribution is

$$\frac{dx^l}{d\xi} = \begin{bmatrix} \frac{d}{d\xi}N_1 & \frac{d}{d\xi}N_2 \end{bmatrix} \begin{pmatrix} x_1^l \\ x_2^l \end{pmatrix}$$
$$= \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{pmatrix} x_1^g \cos\theta_e + y_1^g \sin\theta_e \\ x_2^g \cos\theta_e + y_2^g \sin\theta_e \end{pmatrix}$$

Similarly for the second contribution

$$\frac{du^l}{d\xi} = \begin{bmatrix} \frac{d}{d\xi}N_1 & \frac{d}{d\xi}N_2 \end{bmatrix} \begin{pmatrix} u_1^l \\ u_2^l \end{pmatrix}$$
$$= \begin{bmatrix} -1 & 1 \end{bmatrix} \begin{pmatrix} u_1^g \cos\theta_e + v_1^g \sin\theta_e \\ u_2^g \cos\theta_e + v_2^g \sin\theta_e \end{pmatrix}$$

So the strain in each element is given by

$$\varepsilon_x(x^l) = \frac{u_2^l - u_1^l}{x_2^l - x_1^l} = \frac{(u_2^g - u_1^g)\cos\theta_e + (v_2^g - v_1^g)\sin\theta_e}{(x_2^g - x_1^g)\cos\theta_e + (y_2^g - y_1^g)\sin\theta_e}$$

Being $(x_2^g - x_1^g) = L_e \cos\theta_e$ and $(y_2^g - y_1^g) = L_e \sin\theta_e$:

$$\varepsilon_x(x^l) = \frac{(u_2^g - u_1^g)\cos\theta_e + (v_2^g - v_1^g)\sin\theta_e}{L_e}$$

In [7]:
```
strain_vec = np.zeros((n_elements, ))
stress_vec = np.zeros((n_elements, ))
for ii in range(n_elements):

    left_node, right_node = connectivity_table[ii]
    x_glo_1, y_glo_1 = coordinates[left_node]
    x_glo_2, y_glo_2 = coordinates[right_node]

    u_glo_1, v_glo_1 = q[2*left_node:2*left_node+2]
    u_glo_2, v_glo_2 = q[2*right_node:2*right_node+2]

    strain_vec[ii] = ((u_glo_2 - u_glo_1)*np.cos(theta_vec[ii]) + (v_glo_2 -
                     /((x_glo_2 - x_glo_1)*np.cos(theta_vec[ii]) + (y_glo_2 -

stress_vec = array_EA * strain_vec


print(f"Normalized stress inside each element: \n {stress_vec/F} \n ")
```

```
Normalized stress inside each element:
 [ 1.41421356 -1.        ]
```

Then element one undergoes traction whereas element 2 undergos compression.

Sparse Linear Algebra in Python:

In Python, you can perform sparse matrix algebra using libraries like `scipy.sparse` for handling sparse matrices efficiently. Here's how you can transform the dense matrix equation `q = np.linalg.solve(K, f)` into a sparse matrix equivalent using `scipy.sparse`:

1. **Use `scipy.sparse.linalg.spsolve`** for solving the sparse system of linear equations `K*q = f`.
2. **Create Sparse Matrices** using `scipy.sparse.csr_matrix`.

In [8]:
```
import numpy as np
from scipy.sparse import csr_matrix
from scipy.sparse.linalg import spsolve

# Example dense matrix K and vector f
K_dense = np.array([[4, -1, 0], [-1, 4, -1], [0, -1, 3]])
f = np.array([15, 10, 10])

# Convert the dense matrix K to sparse form
K_sparse = csr_matrix(K_dense)
# Solve the sparse system K*q = f using sparse solver
q = spsolve(K_sparse, f)

# Display the result
print("Solution vector q:", q)
```

```
Solution vector q: [5. 5. 5.]
```

Explanation:

1. **Sparse Matrix Creation**: `csr_matrix()` converts a dense matrix (like `K_dense`) into a compressed sparse row (CSR) format, which is efficient for sparse algebra operations.

2. **Solving the Sparse System**: `spsolve(K_sparse, f)` solves the sparse linear system `K*q = f` without explicitly computing the inverse of `K`, which is more efficient.

This method is analogous to `q = K\f` in MATLAB when `K` is sparse.

Benefits:

- **Efficiency**: Sparse matrix operations save memory and speed up computations, especially for large systems with a lot of zero elements.

Resources

Check the LMS to obtain the PDF version of the notebook.

[Lecture notes from Duke University]
(https://people.duke.edu/~hpgavin/cee421/truss-method.pdf)