Préambule :

La résolution des systèmes linéaires est un problème récurent en ingénierie. La taille des systèmes augmente régulièrement en raison de l'amélioration continue des performances de calcul des ordinateurs (ce qui incite à améliorer la finesse de la discrétisation (spatiale et/ou temporelle) utilisée dans les modèles mathématiques)

Dans la majorité des cas, les systèmes linéaires sont présentés sous forme matricielle, soit $A \cdot X = B$.

Si la matrice A *est inversible, le système est dit de "Cramer", et la solution recherchée est* $X = A^{-1} \cdot B$

Historiquement Gauss (1777-1855) puis Jordan (1842-1899) ont été les premiers à proposer une méthode itérative pour obtenir la solution de tels systèmes matriciels.

Problèmatique:

On considère uniquement des systèmes linéaires de n *équations à* n *inconnues que l'on note sous la forme :*

$$(S): \left\{ \begin{array}{c} a_{1,1}x_1 + \ldots + a_{1,n}x_n = b_1 \\ \ldots \\ a_{n,1}x_1 + \ldots + a_{n,n}x_n = b_n \end{array} \right\}, \quad n \in \mathbb{N}^*,$$

où :
- $\left(a_{i,j}\right)_{\substack{1 \le i \le n \\ 1 \le j \le n}} \in \mathbb{R}^n$ *sont les coefficients du système,*
- $\left(b_i\right)_{1 \le i \le n} \in \mathbb{R}^n$ *sont les valeurs des seconds membres,*
- $\left(x_i\right)_{1 \le i \le n} \in \mathbb{R}^n$ *sont les inconnues.*

L'écriture matricielle est alors $A * X = B$ *, avec les conventions de notations :*

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,n} \end{pmatrix} \in M_{n,n}(\mathbb{R}) \quad X = \begin{pmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{pmatrix} \in M_{n,1}(\mathbb{R}) \quad \text{et} \quad B = \begin{pmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{pmatrix} \in M_{n,1}(\mathbb{R}) .$$

Objectif :

L'objectif de ce TD est de mettre en œuvre la stratégie du pivot partiel de Gauss pour la résolution d'un système linéaire (système carré de type Cramer).

A cette occasion, on utilisera les outils de création et de manipulation de "tableaux"
(ou "array" qui se distinguent des "listes") qu'offre la bibliothèque scientifique "Numpy".

A - Méthode du pivot de Gauss.

Définitions :

- *Si le système (S) n'admet pas de solution, on dit qu'il est incompatible. Dans le cadre du programme d'informatique, on appliquera la méthode de Gauss à des systèmes $n \times n$ (n équations, n inconnues) admettant une solution unique (Système de Cramer).*

- *Le système (S) est un système de Cramer si la matrice A est inversible,*
  la solution est alors $X = A^{-1}B$.

- *La matrice augmentée du système (S) est la matrice notée $(A \mid B)$:*

$$(A \mid B) = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} & b_2 \\ \dots & \dots & \dots & \dots & \dots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} & b_n \end{pmatrix} \in M_{n,n+1}(\mathbb{K})$$

  cette matrice permet des manipulations sur les lignes du système sans modifier la solution recherchée,
  en effet, les lignes $L_{i,1 \le i \le n}$, de la matrice augmentée $(A \mid B)$ sont également celles de (S) ,
  de même que les colonnes $C_{j,1 \le j \le n+1}$, avec $C_{n+1} = B$.

Opérations élémentaires :

Les opérations élémentaires sont :

- *l'échange des lignes L_i et L_j , soit ($L_i \leftrightarrow L_j$),*
- *la dilatation, multiplication d'une ligne L_i par un scalaire non nul $\lambda \neq 0$, soit $L_i \leftarrow \lambda L_i$,*
- *la transvection, ajout de λL_j à la ligne $L_{i \neq j}$, soit $L_i \leftarrow L_i + \lambda L_j$*

Ces opérations modifient l'écriture matricielle du système, mais ne modifient pas la solution associée.

Algorithme du pivot de GAUSS :

L'algorithme du pivot de Gauss permet de résoudre le système (S) par une suite finie d'opérations élémentaires sur les lignes. Il procède en deux étapes principales :

- *la première consiste à échelonner le système, étape dite de "descente",*
  c'est-à-dire obtenir une représentation du système avec une matrice triangulaire,
- *la seconde à le réduire, étape dite de "remontée",*
  c'est-à-dire résoudre le système en commençant par le calcul de x_n ,
  et "remonter" en ré-injectant dans la ligne i les valeurs des $x_k, k \in [i+1, n]$ pour calculer x_i .

Echelonnement du système (ou descente) :

Pour un système de Cramer de taille n , la stratégie d'obtention d'une matrice échelonnée par lignes (MEL) peut se résumer à la suite d'étapes :

- *(a) : On sélectionne une ligne dont le premier terme est non nul (premier pivot). On permute cette ligne avec la première (si elle n'est pas la première !).*
- *(b) : On effectue des opérations $L_i \leftarrow L_i + \lambda L_1$ pour i allant de 2 à n de manière à faire disparaître l'inconnue x_1 des équations correspondant aux lignes 2 à n .*
- *(c) : On recommence les étapes (a) et (b) avec les lignes 2 à n et l'inconnue x_2 (on cherche donc un pivot sur la 2ème colonne dans les lignes $L_{2\leq i\leq n}$, soit : $L_2 ... L_n$).*
- *Ainsi de suite jusqu'à la (n-1)^{ième} étape.*

Remarque :

Le fait que la matrice A soit inversible (système de Cramer) nous assure qu'il est toujours possible, à l'étape i , de trouver un pivot (non nul) sur la $i^{ième}$ colonne dans les lignes $L_i ... L_n$.

Pseudo-code correspondant :

$i \leftarrow 1, j \leftarrow 1$
tant que $i \leq n-1$ *et* $j \leq n$:
    si $\exists k \geq i$ *tel que* $a_{kj} \neq 0$:
        on choisit un tel k et on fait $L_k \leftrightarrow L_i$ (*choix du pivot*)
        pour $k \in [i+1, n]$:
            $L_k \leftarrow L_k - a_{kj} / a_{ij} * L_i$
        $i \leftarrow i+1$
    $j \leftarrow j+1$

Nouveau système :

A l'issue de ces opérations, la matrice augmentée $(A \mid B)$ *est échelonnée par ligne (MEL), on obtient ainsi le système triangulé : (on parle de matrice triangulaire supérieure)*

$$(S') : \left\{ \begin{array}{ccccccccccc} a'_{1,1}x_1 & + & \dots & + & a'_{1,i}x_i \dots & + & \dots & + & a'_{1,n}x_n & = & b'_1 \\ & & & & \dots & & & & \dots & & \dots \\ & & & & a'_{i,i}x_i & + & \dots & + & a'_{i,n}x_n & = & b'_i \\ & & & & & & & & \dots & & \dots \\ & & & & & & & & a'_{n,n}x_n & = & b'_n \end{array} \right\}$$

Réduction du système (ou remontée) :

La résolution du système peut se faire de deux manières différentes :

- *(I) :Méthode analytique (pas transposable algorithmiquement parlant)*

  Par substitution de bas en haut : la $n^{ième}$ *équation donne* x_n
  que l'on reporte dans la $(n-1)^{ième}$ *équation pour obtenir* x_{n-1} *et ainsi de suite…*

- *(II) :Méthode du Pivot de Gauss-Jordan*

  (transposable en un algorithme car reposant sur une suite d'opérations élémentaires) :

  (1) : Multiplication des lignes par $\frac{1}{a'_{i,i}}$ *, les pivots (termes diagonaux) valent alors tous 1.*

  (2) : La dernière ligne donne directement la valeur de x_n *.*
  On utilise le pivot (valant 1) sur la dernière ligne et la dernière colonne pour éliminer x_n *des équations* $L_{n-1}, \dots, L_1$ *par des transvections de la forme* $L_i \leftarrow L_i + \lambda L_n, i \in [n-1, 1]$ *.*

  (3) : La ligne L_{n-1} *donne alors la valeur de* x_{n-1} *.*
  On utilise le pivot valant 1 en position $(n-1, n-1)$ *pour éliminer* x_{n-1} *des équations* $L_{n-2}, \dots, L_1$ *.*

  (4) : On recommence jusqu'à la ligne L_2 *(ainsi* L_1 *se présente comme une équation triviale* $x_1 = \dots$ *).*

Système résolu :

On obtient ainsi un système échelonné réduit de la forme :

$$\begin{cases} x_1 = b_1'' \\ \dots \\ x_i = b_i'' \\ \dots \\ x_n = b_n'' \end{cases}$$

La solution n'est autre que la dernière colonne de la matrice augmentée échelonnée réduite $(A \mid B)$ *!*

Choix du pivot :

Pour illustrer l'incidence du choix du pivot sur les erreurs d'arrondis dans l'application de la méthode de Gauss, étudions le système suivant (S) : $\begin{cases} 10^{-4}x_1 + x_2 = 1 \\ x_1 + x_2 = 2 \end{cases}$

Une résolution exacte donne : $x_1 = \frac{1}{1-10^{-4}} \approx 1{,}0001$ *et* $x_2 = 2 - \frac{1}{1-10^{-4}} \approx 0{,}9999$

Une résolution en virgule flottante avec 3 chiffres significatifs en considérant $a_{1,1} = 10^{-4}$ *comme pivot donne :*

$L_2 \leftarrow L_2 - 10^4 L_1$: $-9999x_2 = -9998$

Compte tenu des erreurs d'arrondi, on a : $-9.99 \times 10^3 x_2 = -9.99 \times 10^3$ *donc* $x_2 = 1$,
ce qui, reporté dans L_1 *, donne* $x_1 = 0$ *. Cette solution ne vérifie pas* $x_1 + x_2 = 2$ *.*

Une résolution en virgule flottante avec 3 chiffres significatifs en considérant $a_{2,1} = 1$ *comme pivot donne :*

$L_1 \leftarrow L_1 - 10^{-4} L_2$: $-0{,}9999x_2 = -0{,}9998$,

Compte tenu des erreurs d'arrondi, on a : $x_2 = 1$ *, ce qui, reporté dans* L_2 *donne* $x_1 = 1$ *.*

Cet exemple montre que considérer des petits pivots peut engendrer des erreurs d'arrondis qui, lors des additions et soustractions, peuvent donner des solutions complètement erronées. Nous admettrons que le choix de petits pivots a de grandes chances d'entraîner une perte de précision dans la soustraction de grands nombres proches.

En conséquence, pour éviter une trop grande instabilité de l'algorithme, il faut choisir le pivot sur la colonne i en choisissant le coefficient $a_{i,j}$ *avec* $i \leq j \leq n$ *ayant la plus grande valeur absolue. C'est la méthode du pivot partiel.*

Pour avoir une plus grande stabilité, on aurait pu choisir la méthode du pivot total qui consiste à autoriser des échanges de lignes et de colonnes et à prendre comme pivot à l'étape i le coefficient $a_{i',j'}$ *avec* $i \leq i' \leq n$ *et* $i \leq j' \leq n$ *ayant la plus grande valeur absolue suivi des échanges de ligne et de colonne* $L_i \leftrightarrow L_{i'}$ *et* $C_i \leftrightarrow C_{j'}$ *.*

Cette méthode présente une difficulté : l'échange de colonne entraîne une modification de l'ordre des inconnues.

B – Programmation "modulaire" des fonctions nécessaires à la mise en œuvre de la méthode du pivot (partiel) de Gauss.

Matrices ?
Tableaux ?
Listes ?

Par défaut, Python dispose de variables du type "liste" comme variables "natives".

Dans un premier temps, nous utiliserons ce type de variable pour représenter les matrices, mais nous verrons que ce type n'est pas adapté pour représenter et manipuler des objets de type "matrices" au sens mathématique.

De quoi parle-t-on ?:

Nous proposerons en fin de TP l'utilisation du type "array" avec la bibliothèque de calcul numérique "numpy", mieux adapté aux matrices et qui dispose d'une palette fonctions d'algèbre linéaire très étendue.

Rappels :

On rappelle ci-dessous quelques syntaxes pour la manipulation des listes.


```
>>> A = [ [ 1,2,3] , [8,7,6] , [-5,-2,-7] ]
>>> print(A)
[[1, 2, 3], [8, 7, 6], [-5, -2, -7]]

>>> print(A[0])
[1, 2, 3]

>>> print(A[1][2])
6
```

En mathématique, on a pour convention de repérer la position d'un élément dans une matrice par ses indices de lignes et de colonnes. Or la transposition de cette convention en informatique n'est pas naturelle si l'on définit une matrice comme "liste de liste" car alors :

- *la longueur de la liste (nombre de sous-liste) correspondant au nombre de lignes,*
- *la longueur de chaque sous-liste (*) correspondant au nombre de colonnes.*

(*) sous réserve qu'elles soient toutes de même longueur, sinon il ne s'agit pas d'une matrice !

Pour définir une matrice "colonne" B *qui soit cohérente avec la convention précédente (c'est-à-dire un vecteur colonne), on choisira la notation :*

```
B = [ [1] ,[56] , [31] ]
```

et pas la notation `B = [ 1, 56, 31 ]` *qui correspondrait à une matrice ligne (!)*

Rappels :

Sous Python, la numérotation commence à 0 et se termine à $n-1$ *pour une taille (ligne ou colonne) de* n *.*

Hypothèses :

On supposera que l'on travaille uniquement avec des matrices donc des listes de listes "rectangulaires".

Mise en garde :

Pour rappel, le type "liste" est mutable c'est à dire qu'il est possible de modifier une partie de l'objet sans changer l'adresse de l'objet (modification sur place).

Pour travailler sur une matrice M *sans la modifier, il faut en créer une copie. La simple affectation* $R = M$ *ne convient pas car* R *et* M *pointent vers la même adresse : la modification de l'une des matrice se répercute sur l'autre.*

Le script suivant permet de créer une vraie copie de la matrice M .
Cette fonction sera à utiliser dans la suite du TP.

```
def Copie(M):
    n=len(M)
    R=[None]*n
    for i in range(n):
        R[i]=M[i][:]
    return R
```

1. Construire une fonction `Dimension(M)` qui retourne le couple de valeurs (n,m), correspondant respectivement au nombre de lignes et de colonnes dans le cas d'une matrice, et qui retourne le couple $(n,None)$ si la variable M passée en argument n'est pas une matrice (*liste non rectangulaire, par exemple*).

Cette fonction permet de détecter d'éventuelles erreurs d'écritures dans les matrices de grandes tailles saisie manuellement et pour lesquelles un terme manque (*ou est dupliqué*) sans que visuellement on s'en rende compte.

```
>>> A = [ [1,2,3,4] , [8,7,6] , [5,-5,-2,-7,-1] ]
>>> Dimension(A)
3, None
>>> A = [ [ 1,2,3,4] , [8,7,6,5] , [-5,-2,-7,-1] ]
>>> Dimension(A)
3, 4
```

2. Construire une fonction `Affiche(M)` qui produit l'affichage d'une matrice par ligne (*sans crochets ni virgules*). La tester sur une matrice définie comme une liste de listes, vérifier également qu'elle est adaptée à l'affichage d'une matrice qui ne soit pas carrée.

indication: la commande `print( … , end='')` permet d'afficher une valeur sans retour à la ligne.

```
>>> A = [ [1,2,3,4] , [8,7,6,5] , [-5,-2,-7,-1] ]
>>> Affiche(A)
1 2 3 4
8 7 6 5
-5 -2 -7 -1
```

3. Construire une fonction `Colonne(M,i)` qui retourne la liste des termes de la colonne i de la matrice M.

```
>>> Colonne(A,2)
[3, 6, -2]
```

4. Construire une fonction `Augmente(M,N)` qui retourne la matrice augmentée $(M\,|\,N)$ et affiche un message d'erreur si les tailles des matrices M et N ne sont pas compatibles.

```
>>> A = [ [1,2,3] , [38,75,62] , [-5,-2,-7] ]
>>> B = [ [145,2] , [256,8] , [335,4] ]
>>> Affiche(Augmente(A,B))
1 2 3 145 2
38 75 62 256 8
-5 -2 -7 335 4
```

5. Construire une fonction `Echange(M,i,j)` qui retourne la matrice M dans laquelle a été effectuée l'échange des lignes $L_i \leftrightarrow L_j$.

```
>>> A = [ [1,2,3] , [38,75,62] , [-5,-2,-7] ]
>>> Affiche(Echange(A,0,2))
-5 -2 -7
38 75 62
1 2 3
```

6. Construire une fonction `Dilat(M,i,a)` qui retourne la matrice M dans laquelle a été effectuée la dilatation $L_i \leftarrow aL_i$.

```
>>> A = [ [1,2,3] , [38,75,62] , [-5,-2,-7] ]
>>> Affiche(Dilat(A,2,-2))
1  2  3
38  75  62
10  4  14
```

7. Construire une fonction `Transvec(M,i,j,a)` qui retourne la matrice M dans laquelle a été effectuée la transvection $L_i \leftarrow L_i + aL_j$.

```
>>> A = [ [1,2,3] , [38,75,62] , [-5,-2,-7] ]
>>> Affiche(Transvec(A,0,2,5))
-24 -8 -32
38 75 62
-5 -2 -7
```

8. Construire une fonction `Choixpivot(M,i,j)` qui parcourt la colonne j de la matrice M à partir de la ligne i et renvoie l'indice de la ligne comportant l'élément dont la valeur absolue est la plus grande.

```
>>> A = [ [1,2,3] , [38,75,62] , [-5,-2,-7] ]
>>> Choixpivot(A,0,2)
1
```

9. Construire une fonction `Echelon(M)` qui transforme la matrice M en une matrice échelonnée par lignes (MEL) par une suite d'opérations élémentaires sur les lignes.

```
>>> A = [ [1,2,3] , [4,5,6] , [7,8,9] ]
>>> Affiche(Echelon(A))
1 2 3
0.0 -3.0 -6.0
0.0 0.0 1.0
```

10. Construire une fonction `Diagnorm(M)` qui normalise les coefficients diagonaux d'une matrice échelonnée par lignes M par dilatation des lignes.

```
>>> A = [ [1,2,3] , [0,2,4] , [0,0,5] ]
>>> Affiche(Diagnorm(A))
1.0 2.0 3.0
0.0 1.0 2.0
0.0 0.0 1.0
```

11. Construire une fonction `Arrondi(M,eps)` qui permet d'annuler les termes de la matrice situés sous la diagonale et dont la valeur absolue est inférieur à ε (*afin de corriger les erreurs d'arrondis qui peuvent apparaître lors des étapes précédentes et qui "brouillent" la lisibilité d'une matrice après échelonnement par lignes*).

```
>>> A = [ [1,2,3] , [4,5,6] , [7,8,10] ]
>>> Echelon(A)
>>> Affiche(A)
>>> Arrondi(A)
>>> Affiche(A)
…
```

Synthèse :

12. En utilisant les fonctions élémentaires précédentes, construire une fonction `Gauss(A,B)` qui retourne X, solution du système d'équations $AX = B$.

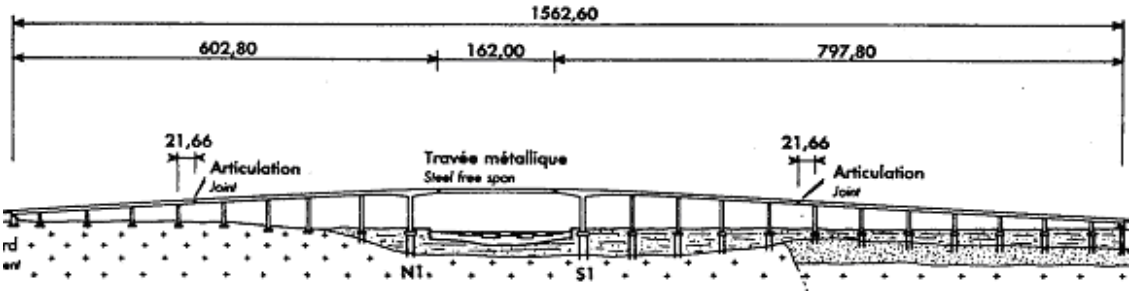
indication : la réduction se fera par opérations élémentaires sur les lignes.

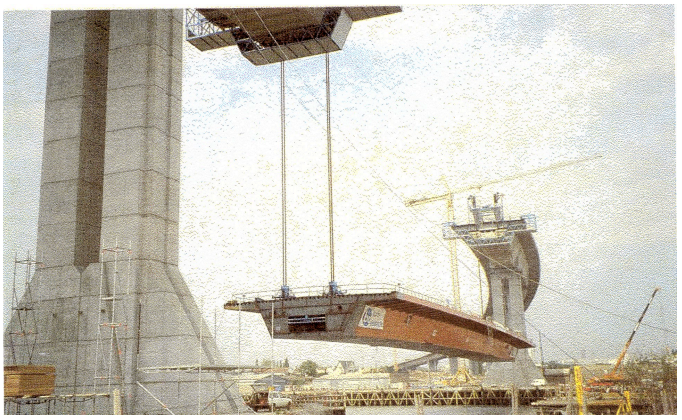
```
>>> A = [ [1,2,3] , [4,5,6] , [7,8,10 ]
>>> B = [ [1] , [4] , [-1] ]
>>> X = Gauss(A,B)
>>> X
[ [-7.0] , [16.0] , [-8.0] ]
```

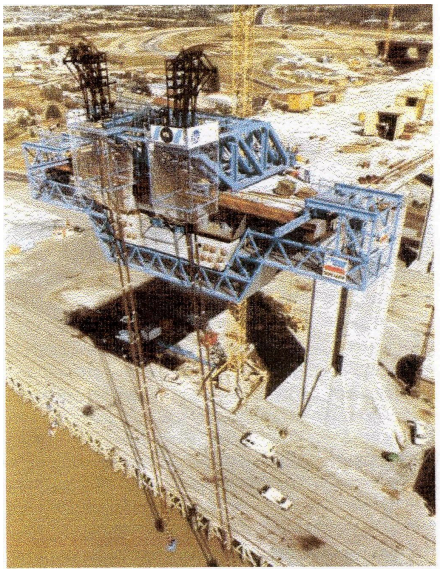
C - Problème proposé : Détermination des tensions dans les poutres d'une structure métallique sous charge.
Application de la méthode de Gauss à la détermination des tensions dans une structure métallique.

De très nombreuses structures d'ouvrages d'art (*ponts, viaducs, traverses, passerelles, etc*…) utilisent pour leur ossature des poutres constituées d'un assemblage de barres articulées (*ou soudées selon les cas*) les unes aux autres; on parle alors de **poutres-treillis**.

Dans le cadre de la construction du pont de Cheviré qui enjambe la Loire en aval de Nantes, les ingénieurs en charge des travaux ont "hissé" le tablier central à plus de 50m de hauteur pour la fixer aux travées en béton existantes. Ce tablier est une structure composite (*acier et béton pré-contraint*) de 162 m de long et d'une masse totale de **2.400 tonnes**. L'opération de hissage s'est fait à partir des deux derniers piliers construits de part et d'autre de la berge sur lesquels avaient préalablement été installé des structures métalliques en porte-à-faux capable de supporter la charge du tablier central.

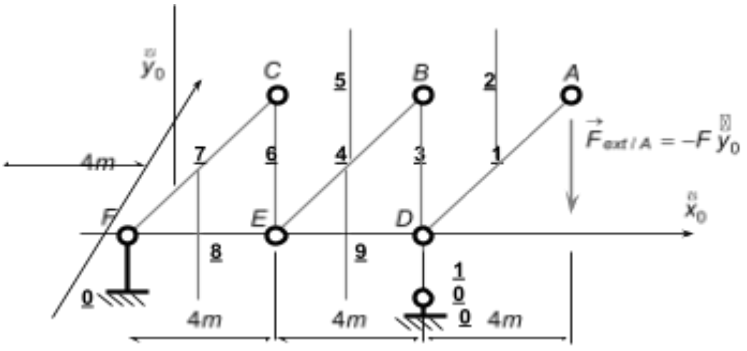






Modèle simplifié et Paramétrage de la structure de hissage :

L'ossature de cette structure est construite à partir de poutres disposées en triangles isocèles rectangles. Chaque articulation (*appelé également nœud*) est repérée sur la figure par une lettre majuscule, elle est considérée comme le centre d'une liaison pivot entre les poutres qui y convergent (*2, 3 ou 4 selon les nœuds*).



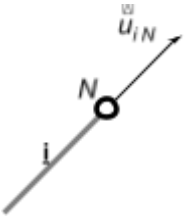
Le tablier a été hissé à partir de 8 câbles suspendus chacun à une structure métallique (*comme celle schématisé ci-dessus*). Chacune des structures a été dimensionnée pour supporter "*1000 tonnes*", soit $10^7\,N$ de tension correspondant à $\vec{F}_{ext/A} = -F\vec{y}_0$. La charge supportée par chaque câble est donc d'environ : $\frac{1}{8} \times 2400.10^3(Kg) \times 10(N/Kg) = 3.10^6\,N$, soit le $1/3$ de sa charge maximale.

Obtention du système d'équations correspondant au problème :

Le problème est ici de déterminer les actions mécaniques de tensions ou compression auxquelles sont soumises les différentes poutres de la structure en fonction de l'action mécanique extérieure F pour en déduire quelles sont les barres les plus sollicitées et les dimensionner correctement.

Pour obtenir le système d'équations représentatif du modèle, on fait les hypothèses suivantes :

- masse des poutres négligeables face à l'intensité des efforts de tensions en jeu,
- problème supposé plan.



Par ailleurs, on adopte pour convention de noter : $\vec{T}_{i/N} = T_i \vec{u}_{iN}$ l'action de la poutre **i** sur le nœud **N** où $\vec{u}_{iN}$ est un vecteur unitaire orienté de la barre **i** vers le nœud **N**. (*cf. figure ci-contre*).

Avec cette convention, si $T_i > 0$, la poutre est sollicitée en compression
et inversement, si $T_i > 0$, la poutre est sollicitée en traction.

Dans ces conditions, l'écriture du PFS appliqué à chacun des nœuds (*excepté le nœud* F) permet d'obtenir avec le théorème de la résultante en projection sur $(\vec{x}_0, \vec{y}_0)$ le système d'équations ci-après, (*système "brut" à gauche et sous forme "matricielle" à droite*).

$$\begin{cases} \frac{\sqrt{2}}{2}T_1 + T_2 = 0 \\ \frac{\sqrt{2}}{2}T_1 = F \\ -T_2 + \frac{\sqrt{2}}{2}T_4 + T_5 = 0 \\ T_3 + \frac{\sqrt{2}}{2}T_4 = 0 \\ -T_5 + \frac{\sqrt{2}}{2}T_7 = 0 \\ T_6 + \frac{\sqrt{2}}{2}T_7 = 0 \\ -\frac{\sqrt{2}}{2}T_1 + T_9 = 0 \\ -\frac{\sqrt{2}}{2}T_1 - T_3 + T_{10} = 0 \\ -\frac{\sqrt{2}}{2}T_4 + T_8 - T_9 = 0 \\ -\frac{\sqrt{2}}{2}T_4 - T_6 = 0 \end{cases} \Leftrightarrow \begin{pmatrix} \frac{\sqrt{2}}{2} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & \frac{\sqrt{2}}{2} & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -\frac{\sqrt{2}}{2} & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & 0 & -1 & 0 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \\ T_6 \\ T_7 \\ T_8 \\ T_9 \\ T_{10} \end{pmatrix} = \begin{pmatrix} 0 \\ F \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

On constate que l'on obtient un système de 10 équations à 10 inconnues de la forme $A \cdot T = B$, où :

- $A = (A_{ij})$ représente la matrice correspondant à l'application du PFS,
- $T = (T_i)$ représente le vecteur colonne correspondant aux inconnues (*tensions des 10 poutres*),
- $B = (B_i)$ représente le vecteur second membre correspondant aux actions mécaniques extérieures à la structure.

Mathématiquement, la résolution consisterait à déterminer la matrice A^{-1} de manière à calculer $T = A^{-1} \cdot B$. La méthode du pivot de Gauss a pour effet de construire $A^{-1} \cdot B$ sans explicitement déterminer A^{-1} , mais nous verrons qu'il est aisé d'obtenir A^{-1} .

Pour simplifier la saisie des données du problème, on propose de noter $r = \frac{\sqrt{2}}{2}$, et de construire la matrice A avec le script suivant :

```
r = sqrt(2)/2

A = [ [ r, 1, 0, 0, 0, 0, 0, 0, 0, 0],
      [ r, 0, 0, 0, 0, 0, 0, 0, 0, 0],
      [ 0,-1, 0, r, 1, 0, 0, 0, 0, 0],
      [ 0, 0, 1, r, 0, 0, 0, 0, 0, 0],
      [ 0, 0, 0, 0,-1, 0, r, 0, 0, 0],
      [ 0, 0, 0, 0, 0, 1, r, 0, 0, 0],
      [-r, 0, 0, 0, 0, 0, 0, 0, 1, 0],
      [-r, 0,-1, 0, 0, 0, 0, 0, 0, 1],
      [ 0, 0, 0,-r, 0, 0, 0, 1,-1, 0],
      [ 0, 0, 0,-r, 0,-1, 0, 0, 0, 0] ]
```

Pour résoudre le problème avec le chargement proposé, il suffit de construire la matrice B représentative du 2nd membre comme :

```
F = 3e6

B = [ [0], [F], [0], [0], [0], [0], [0], [0], [0], [0] ]
```

13. Saisir les données du problème sous python à la fin du script et afficher la solution du problème avec la fonction `Affiche(T)`.

Vérifier que c'est bien la poutre n°10 qui est la plus sollicitée.

Pour vérifier que le programme ainsi construit permet de résoudre le système avec plusieurs 2nd membres, modifier la matrice B de manière à simuler un second cas de chargement où la charge est suspendue pour moitié au nœud A et pour moitié au nœud E, ce qui correspond à un 2nd membre B :

```
B=[[0,0], [F,F/2], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,0], [0,F/2]]
```

14. Compiler votre script, afficher les résultats et conclure si le chargement est ainsi mieux réparti ou pas.

Pour se rendre compte de l'influence que peut avoir tel ou tel chargement sur la structure, il est pertinent de chercher à avoir la matrice inverse A^{-1}, en effet, puisque $T = A^{-1} \cdot B$, la connaissance de A^{-1} permet de prédire les tensions dans une barre à partir du chargement appliqué (*représenté par la matrice* B, 2nd *membre du système*).

15. Quelle matrice B faut-il construire pour que le résultat de la fonction `Gauss(A,B)`, corresponde à A^{-1} ?

Ecrire une routine (fonction) qui permette d'obtenir une telle matrice comme une liste en compréhension (*).
(* *aide du prof si besoin* !)

Compiler votre script et demander l'affichage du résultat avec la fonction `Affiche()`.

On notera `invA` *la matrice inverse*.

D – Utilisation de la bibliothèque Numpy (pour la création de tableaux et la résolution de systèmes linéaires).

Intérêt :

La bibliothèque "Numpy" ajoute le type "array", semblable au type liste (`list`) *mais avec deux conditions :*

- *les tableaux ainsi construits sont rectangulaires (uni, bi, tri… dimensionnels)*
- *les éléments sont tous du même type.*

Ce type est particulièrement adapté à la création et la manipulation des matrices (tableaux uni ou bidimensionnels)

De plus, l'affichage d'un tableau ("array") est beaucoup plus lisible que celui d'une liste ("list") (!). Par exemple :

(*)
```
>>> B = np.array([ [1+i**3-j**2+5*i*j for j in range(5)]
                   for i in range(3)])
>>> B
array([[  1,   0,  -3,  -8, -15],
       [  2,   6,   8,   8,   6],
       [  9,  18,  25,  30,  33]])
```

L'accès à un élément d'un tableau `B` *se fait indifféremment avec les commandes :*

```
>>> B[i,j] ou B[i][j]
```

La conversion d'une liste de liste `A` *en un tableau* `A1` *de type "array" se fait avec la commande :*

```
>>> A1 = np.array(A)
```

Commandes usuelles

`np.zeros(n) ((n,p))`	vecteur nul de taille `n`, matrice nulle de taille `n,p`
`np.ones(n) ((n,p))`	vecteur de taille `n`, matrice de taille `n,p` remplie de 1
`np.eye(n) ((n,p))`	matrice (`n,n`) ou (`n,p`) avec des 1 sur la diagonale et des zéros ailleurs
`np.diag(v)`	matrice diagonale dont la diagonale est le vecteur `v`
`np.random.rand(n) ((n,p))`	vecteur (`n`), matrice (`n,p`) à coefficients aléatoires uniformes sur [0,1]

Opérateurs d'algèbre linéaire :

La bibliothèque "Numpy" dispose d'une très large palette d'opérateurs d'algèbre linéaire, en particulier :

`linalg.solve(a, b)`	Solve a linear matrix equation, or system of linear scalar equations.
`linalg.inv(a)`	Compute the (multiplicative) inverse of a matrix.

Toutes ces routines sont optimisées et sont donc particulièrement performantes (!)

Utilisation de ces opérateurs pour résoudre le problème précédent :

Le problème précédent peut être résolu avec les outils de la bibliothèque "Numpy".

Afin de bien distinguer les résultats, on renommera toutes les variables nécessaires à l'utilisation de "*numpy*" avec le suffixe 1.

16. Vérifier que la routine `np.linalg.solve` permet d'obtenir les résultats précédents (*c'est-à-dire résolution du problème avec un unique 2nd membre, un 2nd membre multiple et obtention de la matrice inverse* `invA1`,)

17. Vérifier que la routine `np.linalg.inv` donne bien la matrice inverse obtenue avec la méthode de Gauss.

(*) dans l'exemple ci-dessus, B *est une matrice (3 x 5), construite comme une liste de listes (en compréhension !)*

avec un terme générique de la forme : $b_{i\,j} = 1 + i^3 - j^2 + 5\,i\,j$

E – Comparaison des performances "Gauss" / "numpy.linalg".

Pour évaluer les performances relatives (en temps CPU) des deux méthodes, on propose de travailler sur des matrices de tailles importantes afin d'avoir des temps CPU "consistants".

Pour construire des matrices de tailles importantes automatiquement, on décide d'utiliser les outils de génération des nombres aléatoires de manière à obtenir des matrices inversibles (et bien conditionnées). En effet, la probabilité d'obtenir ainsi des matrices dont les lignes ou les colonnes sont des combinaisons linéaires les unes des autres est extrêmement faible, ce qui nous "assure" de travailler avec des matrices inversibles.

Modules nécessaires :

Pour pouvoir travailler avec la génération de nombres aléatoires et les utilitaires de mesures de temps CPU, charger les deux modules correspondants en début de script:

```
import time
import random
```

Génération "aléatoire" :

Pour générer une matrice aléatoire de taille (n x m) sous forme de liste de listes, on propose le script suivant :

```
# fonction pour créer une matrice aléatoire d'entiers (int) de taille (n x m)
# sous la forme d'une liste de listes avec des termes compris entre a et b

def Mat_alea(a,b,n,m):
    M = [ [random.randint(a,b) for j in range(m)] for i in range(n) ]
    return M
```

18. Saisir la fonction et vérifier qu'elle donne le résultat escompté avec la fonction `Affiche()`.

Choisir des tailles de matrices "raisonnables", c'est-à-dire inférieure à $n = 20$ (!).

Estimation de la durée d'une opération en temps CPU :

Pour estimer et afficher le temps CPU d'un processus (suite d'opérations), on propose le script suivant :

Remarque : les durées mesurées par la fonction `time` *sont en µs, elles sont multipliées par 1000 pour obtenir des durées en ms (!)*

```
t0 = time.perf_counter()

opération 1
opération 2
...

t1 = time.perf_counter()
print("Temps CPU (en ms) pour faire [...] :",(t1-t0)*1e3)
```

19. Compléter votre script de manière à comparer les performances en temps CPU pour l'inversion d'une matrice A de taille paramétrable, obtenue par génération avec des nombres aléatoires, avec les trois méthodes suivantes :

(a) – méthode du pivot de Gauss programmée,
(b) – méthode `np.linalg.solve`,
(c) – méthode `np.linalg.inv`.

Le résultat sera affiché sous une forme semblable à celle ci-dessous.

```
Estimation des temps CPU (en ms) pour les différentes méthodes
Taille de la matrice A à inverser:  100 x 100
Pivot de Gauss (pivot partiel): 696.4506149076611
numpy.linalg.solve : 1.479204520957289
numpy.linalg.inv : 0.5674935583548368
```

20. Vérifier l'efficacité des méthodes du module "numpy" comparée à la méthode du pivot de Gauss programmée.

ATTENTION: ne pas choisir des tailles "déraisonnables" et se limiter à $n = 50$ *dans un premier temps, avant d'entreprendre des tailles supérieures, et toujours procéder avec parcimonie* !

Eléments de corrigé :

$$A=\begin{pmatrix} r & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ r & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & r & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & r & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & r & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & r & 0 & 0 & 0 \\ -r & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -r & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -r & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & -r & 0 & -1 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad B=\begin{pmatrix} 0 \\ F \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad T=\begin{pmatrix} T_1=+\sqrt{2}F \\ T_2=-F \\ T_3=\frac{1}{2}F \\ T_4=-\frac{\sqrt{2}}{2}F \\ T_5=-\frac{1}{2}F \\ T_6=\frac{1}{2}F \\ T_7=-\frac{\sqrt{2}}{2}F \\ T_8=\frac{1}{2}F \\ T_9=F \\ T_{10}=\frac{3}{2}F \end{pmatrix}$$