République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie d'Oran Mohamed Boudiaf

Faculté d'Architecture et de Génie Civil
Département de Génie Civil

Polycopié

Complément de programmation

Préparé et présenté par

Dr.BABA HAMED Fatima Zohra

Année universitaire 2021-2022

Tables des matières

Introduction générale …………………………………………………………………………. 10
Chapitre I : Notions de base du langage de programmation python…………………………. 13
I.1. Présentation générale………………………………………………………………………. 14
I.2. Les avantages du langage Python………………………………………………………… 14
I.3. Les caractéristiques du langage Python………………………………………………… 14
I.4. Historique des versions de Python……………………………………………………… 15
I.5. La syntaxe du langage Python……………………………………………………………… 15
I.5.1. Les instructions……………..…………………………………………………………….. 15
I.5.2. L'indentation……………………..……………………………………………………….. 15
I.5.3. Commentaires.……………………………………………………………………………… 15
I.5.4. Types et variables.…………………………………………………….………………….. 16
I.5.5. Les opérateurs..…………………………………………………………………………….. 16
I.5.6. Les instruction de sortie et chaînes de caractères.………………………………………. 16
I.6. Les fonctions et les procédures.……………………………………………………………… 17
I.6.1. Définition et utilisation d'une procédure……………………………………………….. 17
I.6.2. Définition et utilisation d'une fonction…………………………………………………. 17
I.7. Structures de contrôle.……………………………………………………………………… 18
I.7.2. Structure itérative : la boucle WHILE…………………………………………………. 19
I.7.3. Structure itérative : la boucle FOR…………………………………………………… 19
I.8. Les listes……………………………………………………………………………………. 20
I.9. Les librairies du langage Python …………………………………………………………. 22
I.9.1. La librairie Time.…………………………………………………………………………. 22
I.9.2. La librairie NumPy …………………………………………………………………………… 22
I.9.3. La librairie matplotlib………………………………..…………………………………. 25
I.10. Les fichiers….……………………………………………………………………………. 27

Chapitre II : Méthodes classiques de résolution des systèmes linéaires………….………… 28
II.1 Introduction……………………………………………………………………………… 29
II.2. Procédures directes de résolution……..……………………………………………… 29
II.2. 1. Méthode du pivot de Gauss (méthode directe)……………………………………….. 29
II.3. Procédures itératives de résolution ………………………………………………… 34
II.3.1. Méthode Jacobi………………………………………………………………………… 35
II.3.2. Méthode Gauss-Seidel …..…………………………………………………………… 38
Chapitre III : Méthode matricielle des déplacements (Structure en barres et poutres)……..… 41
III.1. Introduction …………………………..........……………………………………….. 42
III.2. Degré de liberté……………………..………………………………………………… 42
III.3. Matrice de rigidité des éléments……………………………………………………… 44
Application……………………………………………………………………………………….. 48
III.4. Elément Barre…………………………………………………………………………….. 45
III.5. Structures planes à treillis..…………………………………………………………. 56
Application……………………………………………………………………………………….. 58
III.6. Elément Poutre…………………………………………………………………………….. 70
Application ……………………………………………………………………….…………… 71
Chapitre IV : Méthode des différences finies……………………………………………….. 80
IV.1. Introduction ……………………………………………………………………………….. 81
IV.2. Théorème de Taylor……………………………………………………….……………… 82
IV.3. Application du théorème de Taylor à la méthode des différences finies (cas monodimensionnel)…………………………………..…………………………… 82
IV.4. Applications ……………………………………………………………………………….. 83
IV.4.1 Application 1…………………..……………………………………………………….. 83
IV.4.2 Application 2………..……………………………………………………………….. 85
IV.4.3 Application 3…………………………………………………………………………….. 89

IV.4.4 Application 4…………………………………………………………………….. 94
Références bibliographiques……………………………………………………………… 98

.

Liste des figures

Figure I.1 : Conteneur sous forme d'une figure vide……………………………………………….. 26
Figure I.2 : Courbe de sinus…………………………………………………………………….. 27
Figure III.1 : Représentation d'une poutre……………………………………………………… 42
Figure III.2 : Déplacements des nœuds au niveau d'une poutre………………………………... 42
Figure III.3 : Efforts induits par chaque déplacement unitaire…………………………………. 46
Figure III.4 : Elément Barre………………………………………………………..………….. 47
Figure III.5 : Elément d'un treillis……………………………………………………………... 56
Figure III.6 : Exemple de structure en treillis…………………….……………………………. 56
Figure III.7 : Elément d'un treillis dans le repère local et global……………………………... 57
Figure III.8 : Elément poutre…………………………………………………………………. 70
Figure IV.1 : Courbe de la solution de l'équation…………..…………………………………... 85
Figure IV. 2: Courbe de la solution de l'équation……………….……………………………. 89
Figure IV.3 : Déformée de la poutre…………………………………………………………... 94
Figure IV. 4: Système non amorti…………………………………………………………. 94
Figure IV.5 : Réponse temporelle libre………………………………………………………... 96

Glossaire

Big Data : Mégadonnées désignant l'ensemble des données numériques produites par l'utilisation des nouvelles technologies. La taille des données dépasse en général les capacités d'une seule et unique machine et nécessitent des traitements parallélisés.

CWI : Centrum voor Wiskunde en Informatica

DDL : Degré de liberté

Matplotlib : Bibliothèque du langage de programmation Python destinée à tracer et visualiser des données sous formes de graphiques.

NumPy : est une bibliothèque pour langage de programmation Python, destinée à manipuler des matrices ou tableaux multidimensionnels ainsi que des fonctions mathématiques opérant sur ces tableaux.

SciPy : projet visant à unifier et fédérer un ensemble de bibliothèques Python à usage scientifique

Introduction générale

Les méthodes matricielles forment une partie importante dans les techniques de calcul dans l'analyse des structures. Fondamentalement, ces méthodes adoptent des notations simples pour représenter les forces et les déplacements imposés à la structure, les contraintes et les déformations induites dans les membrures aussi bien que les relations entre les forces, les déplacement ; les contraintes et les déformations.

Les méthodes matricielles ont une grande popularité grâce à leur facilité d'exécutions sur les ordinateurs.

Dans cet ouvrage, on utilise les méthodes matricielles basées sur les principes de la mécanique classique pour analyser les structures. L'accent a été mis sur les informations de coordination des méthodes matricielles dans les techniques de programmation. Pour accomplir cette tâche, notre choix c'est porté sur le langage Python.

Python est un langage de programmation open-source avancé, largement utilisé par les ingénieurs logiciels du monde entier pour différentes applications. Son utilisation est avantageuse pour résoudre des problèmes qui peuvent être théoriquement représenté par des matrices, en se basant sur les différentes opérations d'algèbre matricielle linéaire et de la représentation relativement simple des graphiques.

Ce polycopié s'adresse aux étudiants de première année Master en Génie-Civil, option Charpente métallique et mixte. Il est rédigé de manière à attirer l'attention du lecteur sur les applications pratiques du sujet traité. Pour faciliter la compréhension des processus présentés, nous présentons des exemples de calcul résolus. D'abord de manière manuelle, ensuite sous forme numérique à l'aide du langage de programmation python.

Le polycopié est divisé en quatre chapitres. Le premier chapitre présente les notions de base de programmation sous Python. Il inclut tous les concepts de base que l'étudiant doit savoir. Les sujets traités comprennent les variables, les variables arithmétiques, les matrices, les matrices de calcul, les structures de contrôle, les fonctions intégrées de la matrice, et bien d'autres notions.

Le deuxième chapitre s'intéresse à l'application des méthodes numériques sous Python. Ce chapitre illustre la puissance de travail du Python étape par étape à travers la formulation et la solution d'applications impliquant la solution des équations linéaires.

Le troisième chapitre présente le calcul des structures selon la méthode matricielle des déplacements par le langage python**.** Les calculs des éléments barres, des systèmes treillis et des éléments poutres sont effectués. Des applications par la méthode des différences finies sont présentées dans le chapitre 4.

Enfin, ce document comporte des références bibliographiques permettant au lecteur d'accéder à des informations plus détaillées s'il en éprouve le besoin.

Chapitre I

Notions de base du langage de programmation python

I.1. Présentation générale

Le 20 février 1991 est apparue la première version du langage de programmation python. Elle a été développée par Guido van Rossum au sein du Centrum voor Wiskunde en Informatica (CWI) d'Amsterdam, aux Pays-Bas.

Le but premier de ce langage était d'automatiser l'écriture de scripts et de réaliser rapidement des prototypes d'applications.

Depuis quelques années, toutefois, ce langage de programmation s'est hissé parmi les plus utilisés dans le domaine du développement de logiciels, de gestion d'infrastructure et d'analyse de données. Il s'agit d'un élément moteur de l'explosion du Big Data.

Python est un langage de programmation interprété, qui ne nécessite donc pas d'être compilé pour fonctionner.

I.2. Les avantages du langage Python

L'un des principaux avantages du langage Python est qu'il fonctionne sur tous les principaux systèmes d'exploitation et plateformes informatiques. Il est facile à apprendre et à utiliser. Ses caractéristiques sont peu nombreuses, ce qui permet de créer des programmes rapidement et avec peu d'efforts. De plus, sa syntaxe est conçue pour être lisible et directe.

Malgré sa simplicité, ce langage est aussi utilisé pour créer des logiciels de qualité professionnelle. Qu'il s'agisse d'applications ou de services Web, le langage Python est utilisé par un grand nombre de développeurs pour créer des logiciels. Ainsi, c'est un langage populaire aussi bien parmi les débutants que les experts.

Enfin, c'est un langage qui offre l'avantage d'être open-source sous licence libre.

I.3. Les caractéristiques du langage Python.

Les principales caractéristiques du langage open-source Python résident dans sa simplicité et sa lisibilité. Il est doté d'une bibliothèque de base très fournie. Un grand nombre de bibliothèques sont disponibles pour le calcul scientifique, les statistiques, les bases de données ou la visualisation.

Il est également caractérisé par sa grande portabilité qui lui offre une indépendance vis à vis du système d'exploitation (linux, windows, MacOS). C'est un langage orienté objet avec un typage dynamique, ce qui permet une grande flexibilité et rapidité de programmation, mais qui se traduit par une surconsommation de mémoire et une perte de performance.

Enfin, il présente un support pour l'intégration d'autres langages.

I.4. Historique des versions de Python

Les versions de Python développées depuis sa création et jusqu'en 2009 sont des versions 2.x. Depuis 2009, Python est passé à la version 3. Le changement de version s'est accompagné de quelques modifications de syntaxe, ne permettant pas un portage immédiat d'un script d'une version à l'autre.
Python 3.x est la version actuelle du langage. Elle apporte constamment de nombreuses fonctionnalités nouvelles et très utiles.

I.5. La syntaxe du langage Python.

I.5.1. Les instructions.

Dans le langage Python, chaque instruction occupe une ligne : il n'y a pas de symbole de fin, Il s'agit de passer simplement à la ligne après chaque instruction.
Les instructions Python peuvent être exécutées en écrivant directement en ligne de commande.

```
>>> print("Département de Génie-civil – USTO-MB")
Département de Génie-civil – USTO-MB
```

Il est également possible de créer un fichier python, en utilisant l'extension de fichier.py, et en l'exécutant en ligne de commande.

I.5.2. L'indentation.

Les blocs d'instructions, eux, ne sont pas délimités par un symbole particulier mais sont identifiés par l'indentation, au lieu d'accolades comme en C ou C++ ; ou de begin ... end comme en Pascal ou Ruby. Une augmentation de l'indentation marque le début d'un bloc, et une réduction de l'indentation marque la fin du bloc courant.

```
if 10 > 5:
    print("dix est plus grand que cinq")
```

I.5.3. Commentaires.

Il est possible d'intégrer des commentaires dans un programme Python. Les commentaires commencent par un # :

```
#Ceci est un commentaire.
```

I.5.4. Types et variables.

Python autorise, sans déclaration aucune, la manipulation de types classiques : booléens, entiers, réels, caractères, chaîne de caractères.

```
pi = 3.14                       # déclaration et affectation d'un nombre
Département  = 'Génie-civil' # affectation d'une chaîne de caractères
```

Le symbole = est réservé à l'affectation d'une valeur à une variable.

Le symbole ==, lui, permet d'exprimer un test d'égalité qui ne modifie en rien les variables.

I.5.5. Les opérateurs.

Le langage Python dispose des opérateurs arithmétiques classiques : +, **-**, *, /, // pour la division entière et **%** pour l'opération *modulo* qui donne le reste de la division.
Sur les chaînes de caractères, nous trouvons l'opérateur de concaténation noté par symbole et un opérateur de répétition des chaînes avec le symbole `*`.

```
i     = 10
i     = i + 8
texte = 'La variable i vaut '+str(i)
texte = texte+"\n"
print(texte)
print(' Génie-civil '*3) # répéter une chaîne
```

Le langage Python dispose aussi des opérateurs logiques : **`and`**, **`or`** et **`not`**.

I.5.6. Les instruction de sortie et chaînes de caractères.

Les affichages en langage Python sont réalisés à l'aide de l'instruction :

`print(`*text*`,end="`*final*`")`

Elle affiche le *text* et termine cet affichage avec *final*. Si la partie `end="`*final*`"` n'est pas précisée Python utilise `end="\n"`, ce qui signifie que l'on passe à la ligne après l'affichage de *text*.
Il est possible de confier en une fois plusieurs éléments à afficher à l'instruction `print`. Voici quelques exemples qui illustrent différentes utilisations possibles de `print`.

```
print("Génie-civil")    # affiche Génie-civil et passe à la ligne
print("Génie","civil")  # civil suit Génie, passage à la ligne
print("Génie",end="+")  # termine par un + et pas par un passage à la
                          ligne
print()                 # simple passage à la ligne
```

Le langage Python permet de délimiter les chaînes de caractères soit par des apostrophes, soit par des guillemets. Il est pratique d'utiliser les guillemets lorsque le texte contient une apostrophe, et vice-versa. Il est aussi possible d'utiliser un *backslash* (signe `\`) pour neutraliser la signification Python d'un caractère.

```
print('aujourd\'hui !')
```

I.6. Les fonctions et les procédures.

Les procédures et fonctions ont en commun d'utiliser des paramètres et de rassembler des instructions quelconques. Par contre, une fonction doit toujours se terminer par le renvoi d'un résultat (à l'aide du mot-clef `return` en Python), ce qui n'est jamais le cas pour une procédure. Cela signifie en particulier qu'un appel à une fonction se trouvera souvent dans la partie droite d'une affectation, par contre un appel à une procédure ne pourra pas être à cette position.

I.6.1. Définition et utilisation d'une procédure

```
# forme générale d'une procédure
def nom_de_la_procédure (paramètres):
    # ...
    # des instructions ici
    # ...

# procédure qui affiche entre étoiles un texte donné
def affiche_joliment (msg):
    print('***',msg,'***')
```

I.6.2. Définition et utilisation d'une fonction

```
# forme générale d'une fonction
def nom_de_la_fonction (paramètres):
    # ...
    # des instructions ici
    # ...
    return(résultat)

# fonction qui calcule la somme de deux entiers et la renvoie
def addition (x,y):
    s = x+y
    return(s)

# on appelle la fonction et on renseigne ses paramètres
somme = addition(2,2)
print('le résultat est',somme)
```

I.7. Structures de contrôle.

Le langage Python comporte les structures de contrôle classiques des langages de programmation.

I.7.1. Structure conditionnelle : le *si alors sinon*

```
# forme générale
if (une condition ici):
    # ...
    # des instructions ici
    # ...
else:
    # ...
    # des instructions ici
    # ...

# affichage si un test est vrai
temperature = 28
if temperature>25:
    print('il fait chaud !')
```

```
# affichages distincts selon un test
temperature = 28
if temperature>25:
    print('il fait chaud !')
else:
    print('il fait frais...')

# enchaînement de tests
temperature = 28
if temperature>25:
    print('il fait chaud !')
elif temperature>20:
    print('il fait bon.')
else:
    print('il fait frais...')
```

I.7.2. Structure itérative : la boucle *WHILE*

```
# forme générale
while (une condition ici):
    # ...
    # des instructions ici
    # ...

# on compte de 1 à 10
i = 1
while (i<=10):
    print(i)
    i = i + 1
```

I.7.3. Structure itérative : la boucle *FOR*

La fonction `range` permet de définir un intervalle, intervalle qui peut être parcouru à l'aide de la boucle `for ... in ....`

```
# forme générale
for i in range(min,max):
    # ...
    # des instructions ici
    # ...
```

```
# affichage de 10 étoiles
for i in range(1,11):
    print('*')

# on compte de 1 à 10
for i in range(1,11):
    print(i)
```

I.8. Les listes.

```
# création d'une liste vide et définition de trois cases
l = []
l.append(2)
l.append(3.14)
l.append('Génie-civil')

# liste fournie par range (entiers de 1 à 100)
premiers = range(1,101)

# une liste de notes
notes = [5,12,8,20,10] # on définit une liste
notes[2] = 9           # on modifie la 2ème case
```

Le langage Python propose plusieurs syntaxes pour parcourir une liste. Notons l'expression `len(l)` qui fournit la taille d'une liste `l` et la boucle *for in* dédiée au parcours de listes.

```
# affichage des éléments de la liste...

# ... en utilisant les indices et la longueur de la liste
for i in range(0,len(notes)) :
    print(notes[i])

# ... une deuxième manière :
for note in notes:
    print(note)

# ... une troisième manière:
print(notes)
```

Méthodes Python dédiées aux listes :

- list permet de copier une liste existante pour en créer une nouvelle,
- remove efface la première occurrence d'un élément dans une liste,
- del supprime une case,
- insert ajoute un élément à une position donnée d'une liste,
- append ajoute un élément en fin de liste,
- pop supprime et fournit le dernier élément d'une liste.

```
notes = [5,12,9,20,12,10] # définition d'une liste
l = list(notes)           # copier la liste
l.append(12)              # ajout de 12 en fin de liste
l.insert(3,14)            # ajout de 14 en position 3
del(l[2])                 # suppression de la case 2
l.remove(12)              # suppression du premier 12
l.pop()                   # suppression du dernier élément
print notes               # donne [5, 12, 9, 20, 12, 10]
print l                   # donne [5, 14, 20, 12, 10]
```

Retour sur les chaînes de caractères : il est parfois commode de voir les chaînes de caractères comme des listes de caractères, et effectivement Python permet d'appliquer aux chaînes les opérations dédiées aux listes.

```
# définition d'une chaîne de caractères
phrase = "Génie-civil"

# parcours d'une chaîne caractère par caractère
for l in phrase
    print(l)
print()

# autre parcours caractère par caractère
for i in range(0,len(phrase)):
  print(phrase[i],end="-")
print()
```

I.9. Les librairies du langage Python

Plusieurs syntaxes sont possibles pour préciser les librairies et les fonctions en provenance de ces librairies qu'il est possible d'utiliser.

```
import librairie
...
from librairie import fonction
...
from librairie import f1,f2
...

from librairie import *
```

I.9.1. La librairie Time.

```
import time

t1 = time.process_time()
...
# ici les instructions à chronométrer
...
t2 = time.process_time()
print((t2 - t1),'sec.')
```

I.9.2. La librairie NumPy

NumPy est disponible sur le lien http://www.numpy.org. C'est est un outil performant pour la manipulation de tableaux à plusieurs dimensions. Il ajoute en effet le type array, qui est similaire à une liste, mais dont tous les éléments sont du même type : des entiers, des flottants ou des booléens.

La librairie NumPy possède des fonctions basiques en algèbre linéaire, ainsi que pour les transformées de Fourier.

Au préalable, il est nécessaire d'importer le package **numpy** avec l'instruction suivante :

```
import numpy as np
```

I.9.2.1. Les variables prédéfinies

NumPy définit par défaut la valeur de pi.

```
np.pi
3.141592653589793
```

I.9.2.2. Les tableaux

Les tableaux peuvent être créés avec numpy.array(). On utilise des crochets pour délimiter les listes d'éléments dans les tableaux.

```
a = np.array([[1, 2, 3], [4, 5, 6]])
```

Affichage

```
a
array([[1, 2, 3],
       [4, 5, 6]])
type(a)
numpy.ndarray
```

On constate qu'un objet de type **numpy.ndarray** a été obtenu

- Accès aux éléments d'un tableau

Comme pour les listes, les indices des éléments commencent à zéro.

```
a[0,1]
2
a[1,2]
6
```

- **La fonction numpy.arange()**

```
m = np.arange(3, 15, 2)
m
array([ 3,  5,  7,  9, 11, 13])
type(m)
numpy.ndarray
```

Il est à noter la différence entre numpy.arange() et range() :

- numpy.arange() retourne un objet de type **numpy.ndarray**.
- range() retourne un objet de type **range**.

```
n = range(3, 15, 2)
n
range(3, 15, 2)
type(n)
range
```

Ceci est également à distinguer d'une liste.

```
u = [3, 7, 10]
type(u)
list
```

Il est possible d'obtenir des listes en combinant list et range().

```
list(range(3, 15, 2))
[3, 5, 7, 9, 11, 13]
```

numpy.arange() accepte des arguments qui ne sont pas entiers.

```
np.arange(0, 11*np.pi, np.pi)
array([  0.        ,   3.14159265,   6.28318531,   9.42477796,
        12.56637061,  15.70796327,  18.84955592,  21.99114858,
        25.13274123,  28.27433388,  31.41592654])
```

- **La fonction numpy.linspace()**

numpy.linspace() permet d'obtenir un tableau 1D allant d'une valeur de départ à une valeur de fin avec un nombre donné d'éléments.

```
np.linspace(3, 9, 10)
array([ 3. ,  3.66666667,  4.33333333,  5.  ,  5.66666667,
        6.33333333,  7.  ,  7.66666667,  8.33333333,  9. ])
```

I.9.2.3. Action d'une fonction mathématique sur un tableau.

NumPy dispose d'un grand nombre de fonctions mathématiques qui peuvent être appliquées directement à un tableau. Dans ce cas, la fonction est appliquée à chacun des éléments du tableau.

```
x = np.linspace(-np.pi/2, np.pi/2, 3)
x
array([-1.57079633,  0. ,  1.57079633])
y = np.sin(x)
y
array([-1.,  0.,  1.])
```

I.9.3. La librairie matplotlib.

Matplotlib a vu le jour pour permettre de générer directement des graphiques à partir de Python. Au fil des années, Matplotlib est devenu une librairie puissante, compatible avec beaucoup de plateformes, et capable de générer des graphiques dans beaucoup de formats différents. Elle peut être combinée avec les bibliothèques python de calcul scientifique NumPy et SciPy.

Tout d'abord, il s'agit de mettre en place l'environnement de travail.

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

I.9.3.1. La réalisation de graphiques simples

Pour tracer la courbe d'une fonction, il est nécessaire de créer son conteneur sous forme de figure vide.

```
fig = plt.figure()
ax = plt.axes()
```

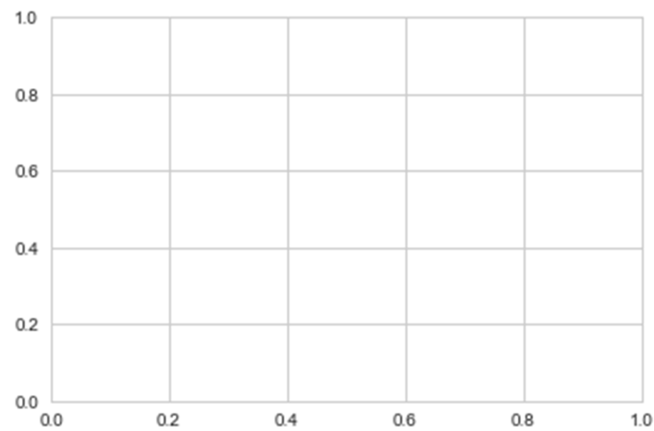


Figure I.1 : Conteneur sous forme d’une figure vide

La variable fig correspond à un conteneur qui contient tous les objets (axes, labels, données, …). Les axes correspondent au carré que l'on voit au-dessus, et qui contiendra par la suite les données du graphe.

```
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
```

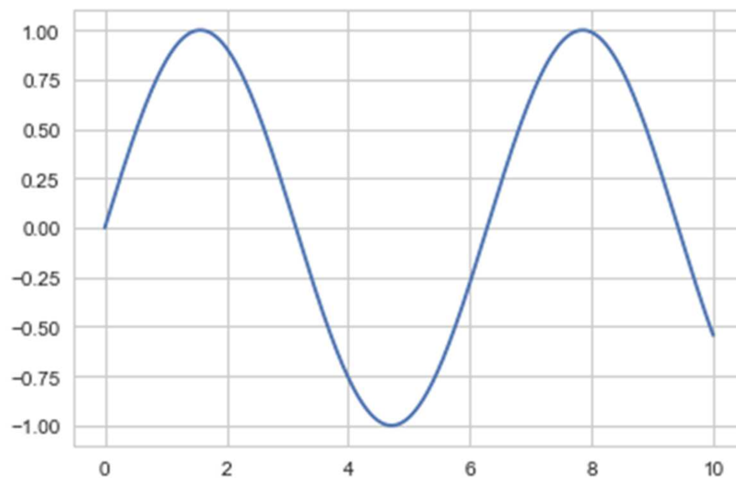


Figure I.2 : Courbe de sinus

Il est également possible d'utiliser `plt.plot(x, np.sin(x))`.

I.10. Les fichiers.

Les instructions Python de base pour créer un fichier texte :

```
# ouverture du fichier en écriture
f = open('monfichier.dat','w')
# écritures dans le fichier
f.write("Génie-civil") # on écrit Génie-civil dans le fichier
f.write(str(12))     # on convertit 12 en texte et on l'écrit
f.write("\n")        # on passe à la ligne dans le fichier
f.close() # fermeture du fichier
```

`write` ne traite que des chaînes de caractères, il est nécessaire de convertir tout autre type de données en chaîne à l'aide de la fonction `str`.

Chapitre II

Méthodes classiques de résolution des systèmes linéaires

II.1 Introduction

Beaucoup de problèmes se réduisent à la résolution numérique d'un système d'équations linéaires. Par exemple la méthode matricielle des déplacements amène dans la plupart des cas à la résolution d'un système de n équations à n inconnues. Les outils de calcul de structures font très souvent appel à des méthodes plus pertinentes telles que celle par élimination de Gauss.

Il existe deux grandes classes de méthodes pour résoudre ce type de systèmes :

1. les méthodes directes qui déterminent explicitement la solution après un nombre fini d'opérations arithmétiques,
2. les méthodes itératives qui consistent à générer une suite qui converge vers la solution du système.

Leur efficacité sera directement liée aux performances du ou des processeurs de l'ordinateur utilisé, de la vitesse d'accès au disque dur mais surtout de la quantité de mémoire vive (RAM) disponible.

Dans le cas des méthodes directes appliquées aux problèmes linéaires élastiques, seul le premier cas de charge nécessite une inversion de la matrice de rigidité. Les résultats sont obtenus par linéarité après stockage de matrice inverse en mémoire (uniquement si les conditions d'appui ne varient pas).

Les méthodes itératives se révèlent moins gourmandes en terme de mémoire. Ceci étant et spécifiquement pour les problèmes linéaires élastiques comportant plusieurs cas de charges.

II.2. Procédures directes de résolution.

Dans cette partie, nous allons considérer la méthode de Gauss

II.2. 1. Méthode du pivot de Gauss (méthode directe)

La méthode du pivot de Gauss est une méthode directe de résolution de système linéaire qui permet de transformer un système en un autre système équivalent échelonné. On résout le système ainsi obtenu à l'aide d'un algorithme de remontée.

Principe

On cherche à résoudre le système suivant de **n** équations à **n** inconnues $\mathbf{x_1, x_2, \ldots, x_n}$:

$$\begin{cases} a_{11}\,x_1 + a_{12}\,x_2 + \ldots + a_{1n}\,x_n = b_1 \\ a_{21}\,x_1 + a_{22}\,x_2 + \ldots + a_{2n}\,x_n = b_2 \\ . \\ . \\ a_{n1}\,x_1 + a_{n2}\,x_1 + \ldots + a_{nn}\,x_n = b_n \end{cases} \qquad \text{(II.1)}$$

Du point de vue matriciel, on a $Ax=b$

Avec

$$\begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \ldots \\ b_n \end{bmatrix} \qquad \text{(II.2)}$$

Les données du système linéaire sont :

- les coefficients réels ou complexes a_{ij} pour i=1,…,n et j=1,…,p (n et p sont deux entiers connus),
- le second membre du système constitué par les nombres réels ou complexes b_i (i=1,…,n),

et L_i (i=1,…,n) désigne la $i^{ième}$ ligne du système (S).

Les inconnues à déterminer sont les x_j (j=1,…,p).

Le système est dit carré lorsque n=p. C'est le cas où il y a autant d'équations que d'inconnues.

On dira que le système est homogène lorsque le second membre est nul (b_i=0, i=1,…,n). On peut remarquer qu'un système linéaire homogène admet au moins la solution nulle x_i=0, ∀i=1,…,n (qui n'est pas nécessairement la seule).

Lorsque tous les coefficients « sous la diagonale » d'un système linéaire sont nuls, i.e. :

$$i > j \Rightarrow a_{ij} = 0$$

On dit que le système est échelonné.

a- la méthode du pivot de Gauss : Triangularisation

$$k = 1, \ldots, n-1 \left\{ \begin{array}{ll|ll} a_{ij}^{(k+1)} = a_{ij}^{(k)} & & i = 1, \ldots, k & j = 1, \ldots, n \\ a_{ij}^{(k+1)} = 0 & & i = k+1, \ldots, n & j = 1, \ldots, k \\ a_{ij}^{(k+1)} = a_{ij}^{(k)} - \dfrac{a_{ik}^{(k)} a_{kj}^{(k)}}{a_{kk}^{(k)}} & & i = k+1, \ldots, n & j = k+1, \ldots, n \\ b_i^{(k+1)} = b_i^{(k)} & & i = 1, \ldots, k & \\ b_i^{(k+1)} = b_i^{(k)} - \dfrac{a_{ik}^{(k)} b_k^{(k)}}{a_{kk}^{(k)}} & & i = k+1, \ldots, n & \end{array} \right. \quad \text{(II.3)}$$

Soit **U** la matrice échelonnée du système, on a alors

$$U = (u_{ij})_{1 \leq i,j \leq n} = (a_{ij}^{(n)})_{1 \leq i,j \leq n}$$

b- Remontée et résolution

À présent la matrice **A** du système linéaire est échelonnée, on doit alors résoudre le système triangulaire :

$$\mathbf{Ux = b(n)}$$

Puisque **b(n)** rappelons-le, est le second membre échelonné, il a subi les mêmes opérations que la matrice échelonnée **U**.

On utilise alors un algorithme de remontée pour le système **Ux = b(n)** :

$$\begin{cases} x_n = \dfrac{y_n}{u_{nn}} = \dfrac{y_n}{a_{nn}^{(n)}}; \\ x_i = \dfrac{1}{u_{ii}}(y_i - \sum_{j=i+1}^{n} u_{ij}x_j) = \dfrac{1}{a_{ii}^{(n)}}(y_i - \sum_{j=i+1}^{n} a_{ij}^{(n)}x_j) \quad \forall i = n-1, n-2, \ldots, 1. \end{cases} \qquad \text{(II.4)}$$

c- Exemple de résolution

Résoudre le système d'équations suivant en utilisant la méthode directe de Gauss

$$\begin{cases} x_1 + 3x_2 + 4x_3 = 1 & L_1 \\ 2x_1 + x_2 + 3x_3 = 2 & L_2 \\ 4x_1 + 7x_2 + 2x_3 = 3 & L_3 \end{cases}$$

Avec

$$A = \begin{pmatrix} 1 & 3 & 4 \\ 2 & 1 & 3 \\ 4 & 7 & 2 \end{pmatrix}, x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Première étape du pivot de Gauss pour éliminer les variables **x_1** dans les lignes **L_2**et **L_3** :

$$\begin{cases} x_1 + 3x_2 + 4x_3 = 1 & L_1 \\ \quad - 5x_2 - 5x_3 = 0 & L_2 \leftarrow 2L_1 - L_2 \\ \quad - 5x_2 - 14x_3 = -1 & L_3 \leftarrow 4L_1 - L_3 \end{cases}$$

Seconde étape du pivot de Gauss pour éliminer les variables **x_2** dans la ligne **L_3** :

$$\begin{cases} x_1 + 3x_2 + 4x_3 = 1 & L_1 \\ \quad - 5x_2 - 5x_3 = 0 & L_2 \\ \quad\quad - 9x_3 = -1 & L_3 \leftarrow L_3 - L_2 \end{cases}$$

En remontant le système, on obtient aisément la solution **x** du système :

$$x = \begin{pmatrix} 8/9 \\ -1/9 \\ 1/9 \end{pmatrix}$$

Résolution par Python :

```
import numpy as np
import sys

# Lecture du nombre d'inconnues
n = int(input('Entrer le nombre des inconnues:'))

# Création d'un tableau numpy de taille n x n+1 et initialisation

a = np.zeros((n,n+1))

# Création d'un tableau numpy de taille n et initialisation

x = np.zeros(n)

# Lecture des coefficients de matrice
print('Saisir les coefficients de matrice :')
for i in range(n):
   for j in range(n+1):
      a[i][j] = float(input( 'a['+str(i)+']['+ str(j)+']='))

# Application de l'élimination de Gauss
for i in range(n):
   if a[i][i] == 0.0:
      sys.exit('Division par zéro détectée !')

   for j in range(i+1, n):
      ratio = a[j][i]/a[i][i]

      for k in range(n+1):
         a[j][k] = a[j][k] - ratio * a[i][k]


x[n-1] = a[n-1][n]/a[n-1][n-1]

for i in range(n-2,-1,-1):
   x[i] = a[i][n]

   for j in range(i+1,n):
      x[i] = x[i] - a[i][j]*x[j]

   x[i] = x[i]/a[i][i]

# Affichage de la solution
print('\nLa solution requise est : ')
for i in range(n):
   print('X%d = %0.2f' %(i,x[i]), end = '\t')
```

Après exécution, on obtient :

```
Entrer le nombre des inconnues:3
Saisir les coefficients de matrice :
a[0][0]=1
a[0][1]=3
a[0][2]=4
a[0][3]=1
a[1][0]=2
a[1][1]=1
a[1][2]=3
a[1][3]=2
a[2][0]=4
a[2][1]=7
a[2][2]=2
a[2][3]=3

La solution requise est :
X0 = 0.89       X1 = -0.11      X2 = 0.11
```

II.3. Procédures itératives de résolution

Principe

On cherche à obtenir une suite de vecteur $(X_n)_n$ convergentes vers $\overline{X}$ telleque $A\overline{X} = b$.

En posant $A = M - N$ où M est une matrice inversible, on propose la formule de récurrence $MX_{n+1} = NX_n + b$ ou encore $X_{n+1} = M^{-1}NX_n + M^{-1}b$. Si $\overline{X}$ est solution du problème, il vient que $(\overline{X} - X_n) = (M^{-1}N)^n \times (\overline{X} - X_0)$. $B = (M^{-1}N)$ est appelée la matrice d'itération.

Ainsi pour que $X_n \to \overline{X}$, il est nécessaire que $(M^{-1}N)^n \to 0$.

Remarque : Le cadre normal de convergence des méthodes itératives classiques est : « A matrice symétrique définie positive ».

Vitesse de convergence :

Théorème : $(B^n)_n \to 0$ ssi $\rho(B) < 1$. Où $\rho(B)$ est le rayon spectral de la matrice B.

Vitesse de convergence : Considérons la matrice d'itération B symétrique, alors $\|B\|_2 = \rho(B)$ et l'on obtient le résultat suivant : $\|x_k - x\|_2 \leq \rho(B)^k . \|x_0 - x\|_2$.

En conclusion, le rayon spectral de la matrice d'itération mesure la vitesse de convergence. La convergence est d'autant plus rapide que le rayon spectral est plus petit. On admettra que ce résultat se généralise aux matrices quelconques.

Dans cette partie, nous allons considérer deux méthodes : les méthodes de Jacobi et de Gauss-Seidel.

II.3.1. Méthode Jacobi

La méthode de Jacobi en algèbre linéaire numérique est une méthode itérative pour calculer la solution d'un système d'équations linéaires strictement dominant en diagonale. Le processus adopté par cette méthode pour résoudre un ensemble d'équations linéaires est le suivant. Supposons qu'un ensemble d'équations linéaires sous forme matricielle soit le suivant :

AX = B

Où

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}$$

La matrice A peut-être décomposée en une composante diagonale (D), une partie triangulaire strictement inférieure (L) et une composante triangulaire strictement supérieure (U) comme :

A=D + L + U

Où

$$D = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{nn} \end{bmatrix} L = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & 0 \end{bmatrix} U = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix} \quad \text{(II.5)}$$

De plus, la solution peut être obtenue itérativement en utilisant la relation suivante :

$$X^{(k+1)}=D^{-1}(B-(L+U)\,X^{(k)}) \quad \text{(II.6)}$$

Où $X^{(k)}$ et $X^{(k+1)}$ sont la $k^{ième}$ et la $(k+1)^{ième}$ itération de X. Les éléments de peuvent être calculés à l'aide de la formule basée sur les éléments :

$$X_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j\neq i}^{n} a_{ij} X_j^{(k+1)}\right] \quad ,i=1,2,\dots\dots,n \quad \text{(II.7)}$$

Ainsi, la nouvelle valeur de X est calculée après avoir mis la valeur itérative précédente de X dans l'équation ci-dessus jusqu'à ce que la précision requise soit atteinte. La condition suffisante mais non nécessaire pour que la méthode converge est que la matrice soit strictement diagonale dominante. Cela signifie que pour chaque ligne, la valeur absolue du terme diagonal est supérieure à la somme des valeurs absolues des autres termes :

$$|a_{ii}| > \sum_{i \neq j} |a_{ij}| \qquad \text{(II.8)}$$

La méthode est simple et numériquement robuste et chaque itération est assez rapide. Cependant, la méthode calcule les variables indépendantes des équations linéaires de manière parallèle et indépendante. Par conséquent, la méthode peut nécessiter beaucoup d'itérations ainsi qu'une bonne puissance de mémoire du système de calcul.

Exemple de résolution

Résoudre le système d'équations suivant en utilisant la méthode itérative de Jacobi

$$\begin{cases} \mathbf{2x_1 + x_2 = 11} \\ \mathbf{5x_1 + 7x_2 = 13} \end{cases}$$

$A = \begin{bmatrix} 2 & 1 \\ 5 & 7 \end{bmatrix}$ B=$\begin{bmatrix} 11 \\ 13 \end{bmatrix}$ X=$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

$D = \begin{bmatrix} 2 & 0 \\ 0 & 7 \end{bmatrix}$ $L = \begin{bmatrix} 0 & 0 \\ 5 & 0 \end{bmatrix}$ $U = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$

$$D^{-1} = \begin{bmatrix} 1/2 & 0 \\ 0 & 1/7 \end{bmatrix}$$

$$-D^{-1}(L+U) = \begin{bmatrix} 0 & -1/2 \\ -5/7 & 0 \end{bmatrix}$$

$$D^{-1}B = \begin{bmatrix} 11/2 \\ 13/7 \end{bmatrix}$$

Ainsi selon la méthode d'itération de Jacobi : $X^{(k+1)}=D^{-1}(B-(L+U)\,X^{(k)})$

Avec: $X^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Résolution par Python :

```
import numpy as np
from scipy.linalg import solve
def Jacobi(A, B, x, n):
# calcul de la matrice diagonale
  D = np.diag(A)
#calcul de  la somme des valeurs supérieures et de la matrice triangulaire inférieure (R=U+L=A-D)
  R = A - np.diagflat(D)
# calcul d'une nouvelle solution à partir d'une ancienne solution
  for i in range(n):
     x = (B - np.dot(R,x))/ D
     print(x)
  return x
'''___Main___'''
A = eval(input('Entrez la matrice A:'))
# as np.array([[a11,a12],[a21,a22]])
B = eval(input('Entrez la matrice B:'))# as [b1,b2]
x = eval(input('Entrez  estimation de x:')) # as [x1,x2]
n = eval(input('Entrez le nombre d'itérations:'))
x = Jacobi(A, B, x, n)
print ('Solution :', solve(A, B))
```

Après exécution, on obtient :

```
Entrez la matrice A:np.array([[2,1],[5,7]])
Entrez la matrice B:[11,13]
Entrez d estimation de x:[0,0]
Entrez le nombre d itérations:25
[5.5        1.85714286]
[ 4.57142857 -2.07142857]
[ 6.53571429 -1.40816327]
[ 6.20408163 -2.81122449]
[ 6.90561224 -2.57434402]
[ 6.78717201 -3.07543732]
[ 7.03771866 -2.99083715]
[ 6.99541858 -3.16979904]
[ 7.08489952 -3.1395847 ]
[ 7.06979235 -3.20349966]
[ 7.10174983 -3.19270882]
[ 7.09635441 -3.21553559]
[ 7.1077678  -3.21168172]
[ 7.10584086 -3.21983414]
[ 7.10991707 -3.21845776]
[ 7.10922888 -3.22136934]
[ 7.11068467 -3.22087777]
[ 7.11043889 -3.22191762]
[ 7.11095881 -3.22174206]
[ 7.11087103 -3.22211344]
[ 7.11105672 -3.22205074]
[ 7.11102537 -3.22218337]
[ 7.11109168 -3.22216098]
[ 7.11108049 -3.22220835]
[ 7.11110417 -3.22220035]
Solution : [ 7.11111111 -3.22222222]
```

II.3.2. Méthode Gauss-Seidel

La méthode de Gauss-Seidel est une méthode itérative pour résoudre un ensemble d'équations linéaires et très similaire à la méthode de Jacobi. Cette méthode est également connue sous le nom de méthode des déplacements successifs. Le nom déplacement successif est dû au fait que la deuxième inconnue est déterminée à partir de la première inconnue dans l'itération en cours, la troisième inconnue est déterminée à partir des première et deuxième inconnues. La méthode peut être appliquée à n'importe quelle matrice avec des éléments diagonaux non nuls. Cependant, la convergence n'est possible que si la matrice est soit diagonale dominante, soit symétrique et définie positivement. Le processus adopté par cette méthode pour résoudre un ensemble d'équations linéaires est le suivant.

Supposons qu'un ensemble d'équations linéaires sous forme matricielle soit le suivant :

AX = B

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}$$

De plus, la matrice A peut-être décomposée en une partie triangulaire inférieure (L*) et une composante triangulaire supérieure stricte (U) comme :

A= L* + U

$$L^* = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} U = \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 \end{bmatrix} \quad (II.9)$$

Et l'équation AX=B peut être écrite comme suit

$$X^{(k+1)} = L_*^{-1}(B - U\,X^{(k)}) \qquad (II.10)$$

Par conséquent, la solution peut être obtenue itérativement en utilisant la relation suivante :

Où $X^{(k)}$ et $X^{(k+1)}$ sont les $k^{ième}$ et $(k+1)^{ième}$ itérations de X.

Les éléments de $X^{(k+1)}$ peuvent être calculés séquentiellement à l'aide de la formule de substitution directe basée sur les éléments :

$$X_i^{(k+1)} = \frac{1}{a_{ii}}\left[b_i - \sum_{j=1}^{i-1} a_{ij} X_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} X_j^{(k)}\right] \quad ,i=1,2,\ldots\ldots,n \qquad \text{(II.11)}$$

Ainsi, le calcul de $X^{(k+1)}$ utilise la valeur de $X^{(k+1)}$ qui a déjà été calculée et uniquement les valeurs de $X^{(k)}$ qui n'ont pas été calculées dans l'itération (k+1). Par conséquent, un seul vecteur de stockage est requis car les éléments peuvent être écrasés lors de leur calcul, ce qui est opposé à la méthode de Jacobi.

Exemple de résolution

Résoudre le système d'équations suivant en utilisant la méthode itérative de Gauss-Seidel

$$\begin{cases} \mathbf{16x_1 + 3x_2 = 11} \\ \mathbf{7x_1 - 11x_2 = 13} \end{cases}$$

$A = \begin{bmatrix} 16 & 3 \\ 7 & -11 \end{bmatrix}$ B= $\begin{bmatrix} 11 \\ 13 \end{bmatrix}$ X= $\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

L *= $\begin{bmatrix} 16 & 0 \\ 7 & -11 \end{bmatrix}$ U = $\begin{bmatrix} 0 & 3 \\ 0 & 0 \end{bmatrix}$

L *= $\begin{bmatrix} 16 & 0 \\ 7 & -11 \end{bmatrix}$ I = $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

R1→ 7R1
R2→ 16R2

$\begin{bmatrix} 112 & 0 \\ 112 & -176 \end{bmatrix}$ $\begin{bmatrix} 7 & 0 \\ 0 & 16 \end{bmatrix}$

R2→ R2- R1
$\begin{bmatrix} 112 & 0 \\ 0 & -176 \end{bmatrix}$ $\begin{bmatrix} 7 & 0 \\ -7 & 16 \end{bmatrix}$

R1→ R1/112
R2→ R2/-176

$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ $\begin{bmatrix} 0.0625 & 0 \\ -0.0398 & -0.0909 \end{bmatrix}$

Par conséquent L_*^{-1}= $\begin{bmatrix} 0.0625 & 0 \\ -0.0398 & -0.0909 \end{bmatrix}$

L_*^{-1}B= $\begin{bmatrix} 0.6875 \\ -0.7439 \end{bmatrix}$ -L_*^{-1}B= $\begin{bmatrix} 0 & -0.1875 \\ 0 & -0.1194 \end{bmatrix}$

Ainsi selon la méthode d'itération de Gauss-Seidel : $X^{(k+1)}=L_*^{-1}(B- U X^{(k)})$

Avec $X^{(0)} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

Résolution par Python :

```
import numpy as np
from scipy.linalg import solve
def GaussSeidel(A, B, x, n):
    L=np.tril(A)
    U=A-L
    for i in range(n):
        x = np.dot(np.linalg.inv(L), B - np.dot(U, x))
        print(x)
    return x
'''___Main___'''
A = eval(input('Entrez la matrice A:'))
B = eval(input('Entrez la matrice B:'))
x = eval(input('Enter estimation de x:'))
n = eval(input('Entrez le nombre d’itérations:'))
x = GaussSeidel(A, B, x, n)

print ('Solution :', solve(A, B))
```

Après exécution, on obtient :

```
Entrez la matrice A:([[2,1],[3,7]])
Entrez la matrice B:[4,3]
Enter estimation de x:[0,0]
Entrez le nombre d itérations:25
[ 2.         -0.42857143]
[ 2.21428571 -0.52040816]
[ 2.26020408 -0.54008746]
[ 2.27004373 -0.54430446]
[ 2.27215223 -0.5452081 ]
[ 2.27260405 -0.54540174]
[ 2.27270087 -0.54544323]
[ 2.27272161 -0.54545212]
[ 2.27272606 -0.54545403]
[ 2.27272701 -0.54545443]
[ 2.27272722 -0.54545452]
[ 2.27272726 -0.54545454]
[ 2.27272727 -0.54545454]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
[ 2.27272727 -0.54545455]
Solution : [ 2.27272727 -0.54545455]
```

Chapitre III

Méthode matricielle des déplacements (Structure en barres et poutres)

III.1. Introduction

La méthode des déplacements est une méthode de résolution systématique des systèmes (iso) hyperstatiques. Elle débouche sur les techniques de résolution matricielle et la résolution numérique des problèmes.

III.2. Degré de liberté

-Dans la méthode des déplacements, les inconnues choisies sont des déplacements ou des degrés de liberté.
-le caractère physique du déplacement se prête plus facilement à une technique de détermination automatique contrairement à une force qui est un concept dans la mesure est déduite) à partir d'une mesure d'un déplacement.
Dans le plan une section donnée possède trois déplacements possibles ou trois degrés de liberté :
-Les deux translations (u) et (v)
-La rotation (θ)

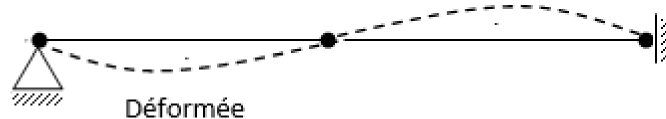


Figure III.1 : Représentation d'une poutre

Une poutre constituée de plusieurs sections, possède théoriquement une infinité de degré de liberté. On peut représenter une poutre par les deux nœuds extrêmes (origine-extrémité). La connaissance des déplacements de ces deux nœuds est suffisante pour connaitre complétement le comportement du milieu.

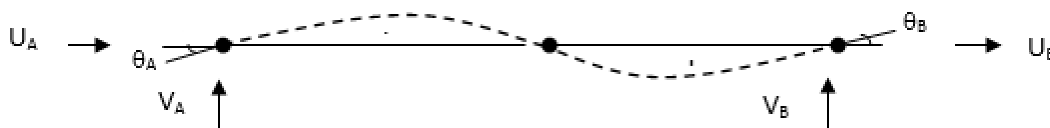
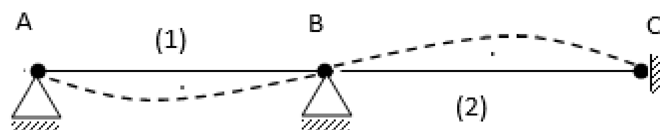


Figure III.2 : Déplacements des nœuds au niveau d'une poutre

Ainsi une poutre dans le plan possède six déplacements possibles ou six degrés de liberté.
Nombre de degrés de liberté =Nombre d'indéterminations cinématiques de la structure compatibles avec les liaisons existantes.

Exemple



L'élément de poutre (1) posséderait

De même l'élément (2)

Ceci représente un bilan exhaustif de tous les degrés de liberté en déplacement possible de la structure.

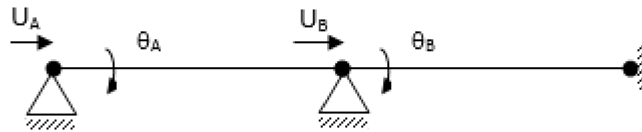
Cependant la continuité au nœud B impose :

$U^1_B = U^2_B = U_B$; $V^1_B = V^2_B = V_B$; $\theta^1_B = \theta^2_B = \theta_B$;

Par ailleurs les conditions de liaison de la structure permettant la détermination de certains degrés de liberté soit :

$V_A = V_B = V_C=0$; $\theta_C=0$ et $U_C=0$

Finalement la structure possède les quatre degrés de liberté restants.



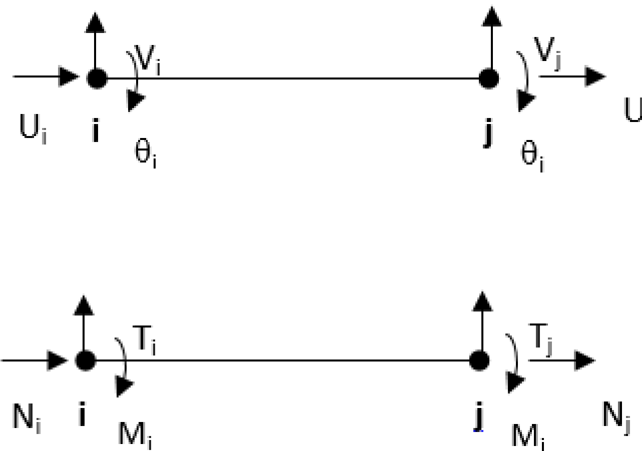
De plus si nous retenons l'hypothèse de négliger les déformations axiales.

Du fait que U_C=0 alors U_B= U_A=0

Avec cette hypothèse ; la structure ne possède que deux degrés de liberté : les rotations θ_A et θ_B.

III.3. Matrice de rigidité des éléments

La déformation d'un élément de la structure est représentée par les déplacements des nœuds qui le déterminent.



On remarquera que les déplacements (D.D.L) et les efforts aux nœuds sont représentées dans ce cas dans un système local ; lié à la barre ou la poutre (du nœud i vers le nœud j).

Le problème qui est posé est de lié le vecteur $[u_i\ \ vi\ \ \theta_i\ \ u_j\ \ v_j\ \theta_j\ ]^T$ représentant les déplacements au vecteur$[N_i\ \ Ti\ \ M_i\ \ N_j\ \ T_j\ M_j\ ]^T$ représentant les efforts aux nœuds.

Par analogie à la relation f=k.x liant l'effort dans un ressort du à un déplacement (x) en fonction de la rigidité k.

K : est un facteur de rigidité qui est donc l'effort produit par un déplacement unitaire x=1.

La relation précédente lie un seul effort (f) à un seul degré de liberté (x). si l'on généralise ; on peut donc écrire la relation [F]=[k].[δ]

Avec $[f]=[\,[N_i\ \ Ti\ \ M_i\ \ N_j\ \ T_j\ \ M_j\,]^T$ et $[\delta]=[u_i\ \ vi\ \ \theta_i\ \ u_j\ \ v_j\ \ \theta_j\,]^T$

[f] et [δ] étant des vecteurs de dimension (6x1) ; la matrice [k] devra être de dimension (6x6) et chacun de ces coefficients k_{ij} représente un facteur de rigidité ; c'est-à-dire l'effort produit par un déplacement unitaire.

Pour déterminer les coefficients de rigidité k ; il faudrait donc calculer les efforts induits par chaque déplacement unitaire.

Si on écrit :

$$\begin{bmatrix} N_i \\ T_i \\ M_i \\ N_j \\ T_j \\ M_j \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & k_{11} & k_{11} \\ k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} \\ k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} \\ k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} \\ k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\ k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix} \times \begin{bmatrix} u_i \\ v_i \\ \theta_i \\ u_j \\ v_j \\ \theta_j \end{bmatrix}$$

Le coefficient k_{11} représentera l'effort normal induit par u_1=1.

Pour déterminer tous ces coefficients, on donnera successivement aux déplacements u_i , vi ,θ_i , u_j ,v_j , θ_j la valeur unitaire alors que tous les autres déplacements seront bloqués.

Tous les autres coefficients seront obtenus en considérant tous les modes de déformation de la poutre.

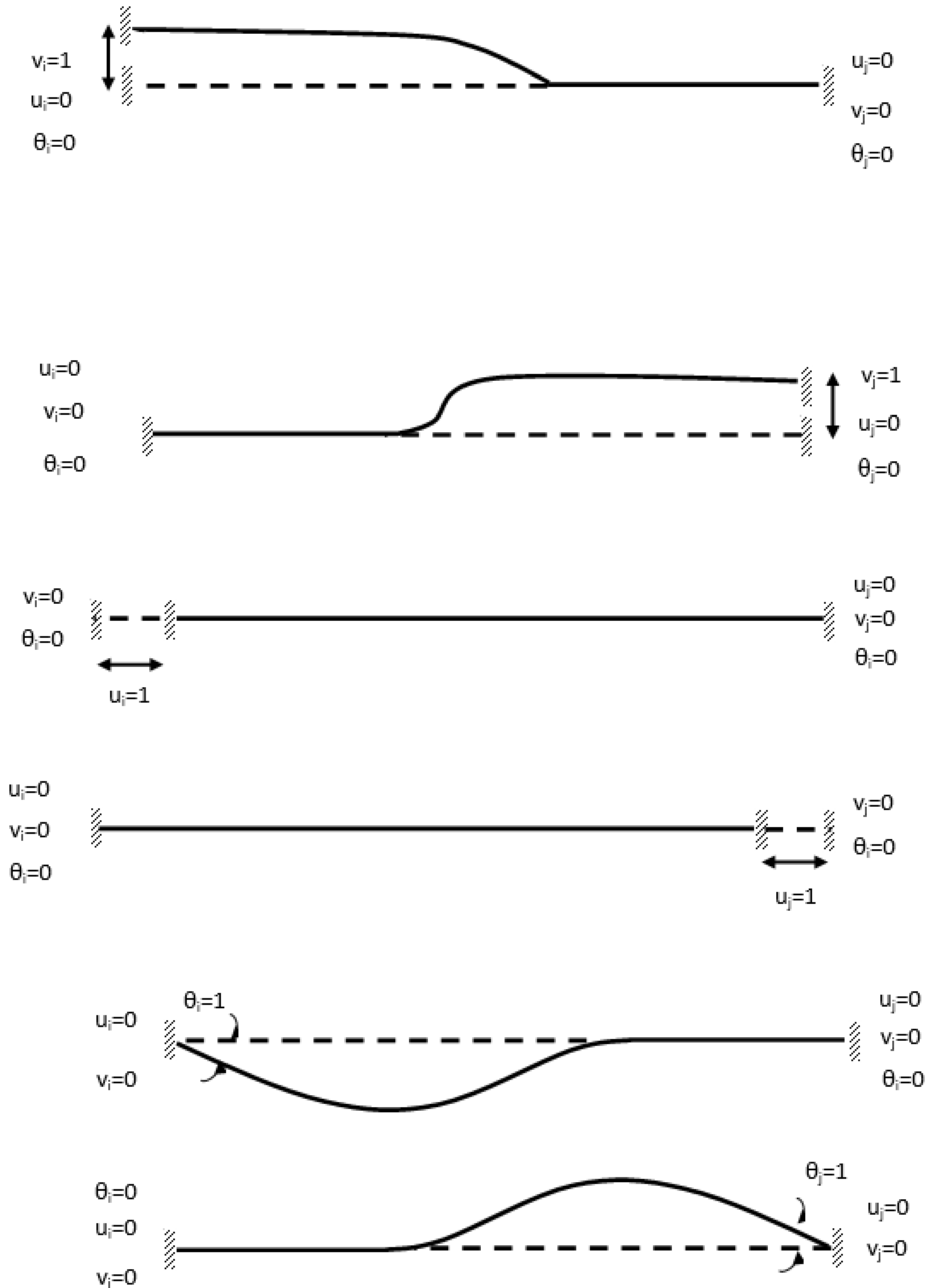


Figure III.3 : Efforts induits par chaque déplacement unitaire.

On obtient le système d'équations suivant :

$$\begin{bmatrix} N_i \\ T_i \\ M_i \\ N_j \\ T_j \\ M_j \end{bmatrix} = \begin{bmatrix} \frac{EA}{L} & 0 & 0 & -\frac{EA}{L} & 0 & 0 \\ 0 & \frac{12EI}{L^3} & \frac{6EI}{L^2} & 0 & -\frac{12EI}{L^3} & \frac{6EI}{L^2} \\ 0 & \frac{6EI}{L^2} & \frac{4EI}{L} & 0 & -\frac{6EI}{L^2} & \frac{2EI}{L} \\ -\frac{EA}{L} & 0 & 0 & \frac{EA}{L} & 0 & 0 \\ 0 & -\frac{12EI}{L^3} & -\frac{6EI}{L^2} & 0 & \frac{12EI}{L^3} & -\frac{6EI}{L^2} \\ 0 & \frac{6EI}{L^2} & \frac{2EI}{L} & 0 & -\frac{6EI}{L^2} & \frac{4EI}{L} \end{bmatrix} \times \begin{bmatrix} u_i \\ v_i \\ \theta_i \\ u_j \\ v_j \\ \theta_j \end{bmatrix}$$

Cette matrice de rigidité peut être spécialisée pour une barre ou un élément de treillis. Il suffit d'éliminer les degrés de liberté qui ne sont pas actifs et de retenir simplement u_i et u_j.

III.4. Elément Barre

L'élément barre linéaire est un élément unidimensionnel où les coordonnées globale et locale coïncident. Il schématise un composant [IJ] d'une structure qui travaille uniquement en traction ou compression. L'élément linéaire possède un module d'élasticité E, une aire de section transversale A, et la longueur L.

Chaque élément de barre linéaire a deux nœuds comme le montre la figure ci-dessous. Dans ce cas, la matrice de rigidité de l'élément est donnée par :

$$K = \begin{bmatrix} \frac{EA}{L} & -\frac{EA}{L} \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix}$$

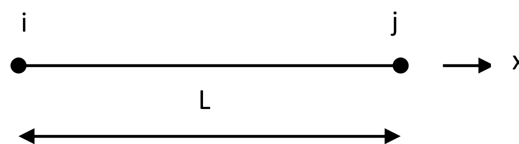


Figure III.4 : Elément Barre

La matrice de rigidité pour l'élément de barre linéaire est similaire à celle de l'élément de ressort avec la rigidité remplacée par EA / L. L'élément barre linéaire a seulement deux degrés de liberté, une à chaque nœud. En conséquence, pour une structure à n nœuds, la matrice de rigidité globale K sera de taille $n \times n$ (puisque nous avons un degré de liberté à chaque nœud).

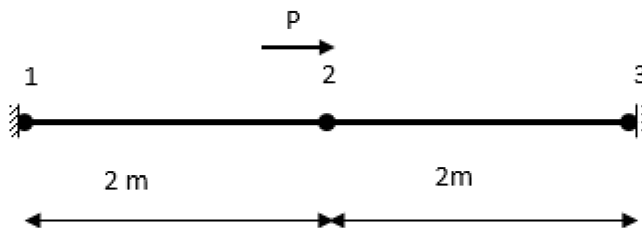
Une fois la matrice de rigidité globale K est obtenue, nous avons l'équation de la structure suivante :

$$[K]\{U\} = \{F\}$$

Où U est le vecteur de déplacement nodal global et F est le vecteur de force nodale globale. À cette étape, les conditions aux limites sont appliquées pour les vecteurs U et F ; ce système d'équation est résolu par la séparation et l'élimination de Gauss.

Application

Considérons l'ensemble, constitué de deux barres linéaires comme le montre la figure ci-dessous.



Données :

E =200 GPa, A =0.003 m^2, P =20 kN.

Déterminer :

1.la matrice de rigidité globale de la structure.

2. le déplacement au nœud 2.

3. les réactions aux nœuds 1 et 3.

Solution :

Étape 1 -La discrétisation du domaine :

Ce problème est déjà discrétisé. Le domaine est divisé en deux éléments et trois nœuds. Le Tableau ci-dessous montre la connexion de l'élément pour cet exemple.

Element	**Noeud 1**	**Noeud 2**
1	**1**	**2**
2	**2**	**3**

Étape 2 –calcul des matrices de rigidité

Les deux matrices de rigidité k_1 et k_2 sont de taille 2×2.

Elles sont calculées comme suit :

$$k_1 = k_2 = \begin{bmatrix} \frac{\mathrm{E} * \mathrm{A}}{\mathrm{L}} & -\frac{\mathrm{E} * \mathrm{A}}{\mathrm{L}} \\ -\frac{\mathrm{E} * \mathrm{A}}{\mathrm{L}} & \frac{\mathrm{E} * \mathrm{A}}{\mathrm{L}} \end{bmatrix} = \begin{bmatrix} 300000 & -300000 \\ -300000 & 300000 \end{bmatrix}$$

Étape 3 –Assemblage de la matrice de rigidité globale:

Étant donné que la structure comporte trois nœuds, la taille de la matrice de rigidité globale est de 3 x3.

$$K = \begin{bmatrix} 300000 & -300000 & 0 \\ -300000 & 300000 + 300000 & -300000 \\ 0 & -300000 & 300000 \end{bmatrix}$$

$$K = \begin{bmatrix} 300000 & -300000 & 0 \\ -300000 & 600000 & -300000 \\ 0 & -300000 & 300000 \end{bmatrix}$$

Étape 4 –Application des conditions aux limites :

En utilisant la matrice de rigidité globale obtenue à l'étape précédente, on obtient le système d’équation suivant :

$$\begin{bmatrix} 300000 & -300000 & 0 \\ -300000 & 600000 & -300000 \\ 0 & -300000 & 300000 \end{bmatrix} \begin{Bmatrix} \mathrm{U}_1 \\ \mathrm{U}_2 \\ \mathrm{U}_3 \end{Bmatrix} = \begin{Bmatrix} \mathrm{F}_1 \\ \mathrm{F}_2 \\ \mathrm{F}_3 \end{Bmatrix}$$

Les conditions aux limites pour ce problème sont données à titre :
U1 =0,
F2 =20,
U3=0.

Nous obtenons :

$$\begin{bmatrix} 300000 & -300000 & 0 \\ -300000 & 600000 & -300000 \\ 0 & -300000 & 300000 \end{bmatrix} \begin{Bmatrix} 0 \\ \mathrm{U}_2 \\ 0 \end{Bmatrix} = \begin{Bmatrix} \mathrm{F}_1 \\ 20 \\ \mathrm{F}_3 \end{Bmatrix}$$

Étape 5 -la résolution des équations :

La résolution du système d'équations obtenu sera effectuée par partitionnement (manuellement) et l'élimination de Gauss.

Nous obtenons :

600000 U2= {20}

Le déplacement au nœud 2 est 3.33* 10^{-5} m.

Étape 6 -les réactions aux nœuds 1 et 3**:**

$$\begin{bmatrix} 300000 & -300000 & 0 \\ -300000 & 600000 & -300000 \\ 0 & -300000 & 300000 \end{bmatrix} \begin{Bmatrix} 0 \\ 3.33 * 10^{-5} \\ 0 \end{Bmatrix} = \begin{Bmatrix} F_1 \\ 20 \\ F_3 \end{Bmatrix}$$

-300000*3.33* 10^{-5}=F_1

F_1= F_3=-10 kN

Ainsi, les réactions au niveau des nœuds 1 et 3 sont des forces de 10 kN (dirigé vers la gauche)

Résolution par Python

Le code Python utilisé pour le calcul des matrices de rigidité élémentaires ainsi que la matrice de rigidité globale des éléments barre est le suivant :

```
import math
import numpy

numpy.set_printoptions(3, suppress=True)

tn = int(input('Entrez le nombre total de nœuds : ')) #Nbre de nœuds
te = int(input('Entrez le nombre total d'éléments : ')) #Nbre d'éléments
xco = []

for i in range(tn):
   x = float(input('Entrez la coordonnée du nœud '+str(i+1)+' in m : '))
   xco.append(x)


A = float(input('Entrez aire de la section transversale en m2: '))
E = float(input('Entrez le module d'élasticité  : '))

snofel = [] #nœud de départ des éléments
enofel = [] #nœud d'extrémité des éléments
lenofel = [] #longueur de l'élément
elcon = [] #constant_élément


for i in range(te):
   a = int(input('Entrez le nœud de départ d'élément '+str(i+1)+' : '))
   b = int(input('Entrez le  nœud d extrémité de l'élément '+str(i+1)+' : '))
   x1 = float(xco[a-1])
   x2 = float(xco[b-1])
   l =x2-x1
   con = A*E/l
   snofel.append(a)
   enofel.append(b)
   lenofel.append(l)
   elcon.append(con)

elstmat = [] #matrice de rigidité des éléments
for i in range(te):
   mat = elcon[i]*numpy.array([[1, -1],[-1, 1]])

   elstmat.append(mat)
```

Après exécution, on obtient :

```
Entrez le nombre total de nœuds : 3
Entrez le nombre total d éléments : 2
Entrez la coordonnée du nœud 1 in m : 0
Entrez la coordonnée du nœud 2 in m : 2
Entrez la coordonnée du nœud 3 in m : 4
Entrez aire de la section transversale en m2: 0.003
Entrez le module d élasticité  : 200e6
Entrez le nœud de départ d élément 1 : 1
Entrez le  noeud d extrémité de l element 1 : 2
Entrez le nœud de départ d élément 2 : 2
Entrez le  noeud d extrémité de l element 2 : 3
[array([[ 300000., -300000.],
       [-300000.,  300000.]]), array([[ 300000., -300000.],
       [-300000.,  300000.]])]
```

Le code Python utilisé pour le calcul de la matrice de rigidité globale des éléments barre est le suivant :

```
gstmatmap = []      ## Maillage_la matrice de rigidité globale, gst matmap sera la matrice carrée de tn*
for i in range(te):  ## faire ceci pour chaque élément
   m = snofel[i]     ## prendre le nœud de départ de l'élément (i)
   n = enofel[i]    ## prendre le nœud d'extrémité de l'élément (i)
   add = [m, n]

   gmat = numpy.zeros((tn, tn))
   elmat = elstmat[i]
   for j in range(2):
      for k in range(2):
         a = add[j]-1
         b = add[k]-1
         gmat[a,b] = elmat[j,k]
   gstmatmap.append(gmat)


GSM = numpy.zeros((tn, tn))
for mat in gstmatmap:
   GSM = GSM+mat


print('Matrice de rigidité globale de la barre')
print(numpy.around(GSM, 3))
```

Après exécution, on obtient :

```
Matrice de rigidité globale de la barre
[[ 300000. -300000.       0.]
 [-300000.  600000. -300000.]
 [      0. -300000.  300000.]]
```

Le code Python utilisé pour l'application des conditions aux limites et le chargement est le suivant :

```
#----------------------Condition aux limites et chargement --------------------#
displist = []
forcelist = []
for i in range(tn):
   a = str('u')+str(i+1)
   displist.append(a)
   c = str('fx')+str(i+1)
   forcelist.append(c)

   print('\n\n________________Condition aux limites______________\n')

dispmat = numpy.ones((tn,1))
tsupn = int(input('Entrez le nombre de nœuds bloqués : '))
supcondition = ['H = Horizontalement bloqué ']
for i in range(tsupn):
   supn = int(input('Entrez le numéros de nœuds bloqué : '))
   for a in supcondition:
      print(a)
   condition = str(input('Entrez la condition du nœuds bloqué : '))
   if condition in['H', 'H']:
      dispmat[supn-1, 0] = 0
   else:
      print('Veuillez saisir des entrées valides')

print('\n_________________chargement_____________________\n')
forcemat = numpy.zeros((tn,1))
tlon = int(input('Entrez le nombre de nœuds chargés : '))

for i in range(tlon):
   lon = int(input('Entrez le numéros de nœuds chargé : '))
   fx = float(input('Entrez la valeur de force : '))
   forcemat[lon-1, 0] = fx

   print(i,"   ",fx)
```

Après exécution, on obtient :

```
________________Condition aux limites______________

Entrez le nombre de nœuds bloqués : 2
Entrez le numéros de nœuds bloqué : 1
H = Horizontalement bloqué
Entrez la condition du nœuds bloqué : H
Entrez le numéros de nœuds bloqué : 3
H = Horizontalement bloqué
Entrez la condition du nœuds bloqué : H

_________________chargement____________________

Entrez le nombre de nœuds chargés : 1
Entrez le numéros de nœuds chargé : 2
Entrez la valeur de force : 20
```

Le code Python utilisé pour la résolution du système d'équation linéaire ainsi que le calcul des réactions est le suivant :

```
###____________________Réduction de matrice____________________###

rcdlist = []
for i in range(tn):
    if dispmat[i,0] == 0:
        rcdlist.append(i)
    elif dispmat[i,0] == 0.002:
        rcdlist.append(i)

rrgsm = numpy.delete(GSM, rcdlist, 0)
crgsm = numpy.delete(rrgsm, rcdlist, 1)
rgsm = crgsm
rforcemat = numpy.delete(forcemat, rcdlist, 0)

print(rforcemat)
rdispmat = numpy.delete(dispmat, rcdlist, 0)

print(rdispmat)
###__________________Résolution________________________###

dispresult = numpy.matmul(numpy.linalg.inv(rgsm), rforcemat)
rin = 0
for i in range(tn):
    if dispmat[i,0] == 1:
        dispmat[i,0] = dispresult[rin,0]
        rin = rin+1


forceresult = numpy.matmul(GSM, dispmat)


print('vecteur des r&actions')
print(forceresult)
```

Après exécution, on obtient :

```
vecteur des r&actions
[[-10.]
 [ 20.]
 [-10.]]
```

III.5. Structures planes à treillis

L'élément plan de treillis est un élément à deux dimensions avec les deux coordonnées locales et globales.

Il est caractérisé par des fonctions de forme linéaire. Chaque élément de treillis a deux nœuds est incliné avec un angle θ mesuré dans le sens antihoraire à partir de l'axe x.

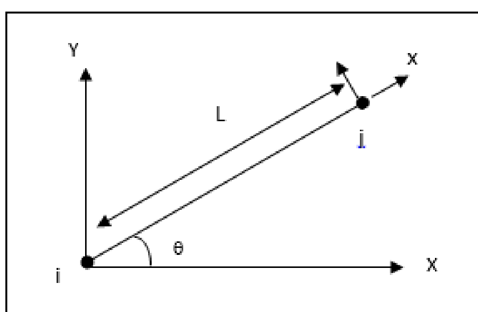


Figure III.5 : Elément d'un treillis

Les matrices de rigidité (barre de treillis) sont exprimées dans un repère local à l'élément.

Dans un calcul de structure constituée par un ensemble d'éléments, les équations d'équilibre sont écrites dans un repère global, dans lequel on repère la structure.

Aussi, il est nécessaire de pouvoir exprimer ces matrices de rigidité dans un système global.

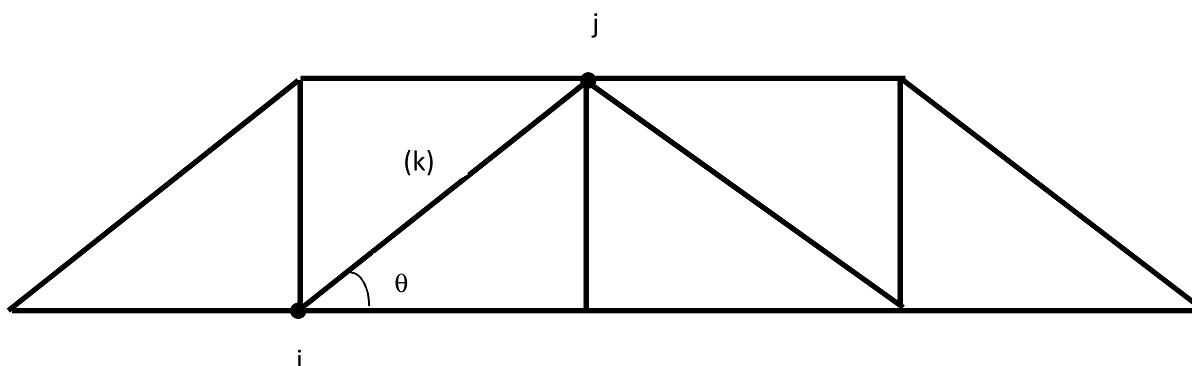


Figure III.6 : Exemple de structure en treillis

Si l'on s'intéresse à la barre (k) du treillis ci-dessus, on peut repérer les degrés de liberté dans les deux repères local et global.

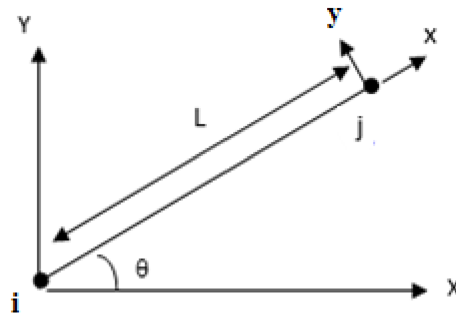


Figure III.7 : Elément d'un treillis dans le repère local et global

On pourra noter :

$[\delta]^k = [u_i \quad u_j]^{k\,T}$ D.D.L (système local)

Et $[\Delta]^k = [\Delta x_i \quad \Delta y_i \quad \Delta x_j \quad \Delta y_j]^{k\,T}$ D.D.L (système global)

De même

$[f]^k = [N_i \quad N_j]^{k\,T}$ D.D.L (système local)

Et $[F]^k = [Fx_i \quad Fy_i \quad Fx_j \quad Fy_j]^{k\,T}$ D.D.L (système global)

Nous avons établi $[f]^k=[k]^k.[\delta]^k$ ou $[f]=[k].[\delta]$

Nous voulons établir $[F]=[K_G].[\Delta]$

On peut toujours déterminer $[\delta]=[R].\ [\Delta]$ Où [R] est **une matrice de transformation**.

Par exemple dans le cas considéré

$Ui=\Delta x_i \cos\theta + \Delta y_i \sin\theta$

$Uj=\Delta x_j \cos\theta + \Delta y_j \sin\theta$

Que l'on peut écrire sous forme matricielle.

$$\begin{bmatrix} u_i \\ u_j \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ 0 & 0 & \cos\theta & \sin\theta \end{bmatrix} \times \begin{bmatrix} \Delta x_i \\ \Delta y_i \\ \Delta x_j \\ \Delta y_j \end{bmatrix}$$

De même, on peut écrire $[f]=[R].\ [F]$ ou $[F]=[R^{-1}].\ [F]$

$F_{xi}=N_i \cos\theta$

$F_{yi}=N_i \sin\theta$

$F_{xj}=N_j \cos\theta$

$F_{yj}=N_j \sin\theta$

Qui peut s'écrire

$$\begin{bmatrix} Fx_i \\ Fy_i \\ Fx_j \\ Fy_j \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & \cos\theta \\ 0 & \sin\theta \end{bmatrix} \times \begin{bmatrix} N_i \\ N_j \end{bmatrix}$$

On a donc

$$[R] = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ 0 & 0 & \cos\theta & \sin\theta \end{bmatrix} \text{ et } [R^{-1}] = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & \cos\theta \\ 0 & \sin\theta \end{bmatrix}$$

On remarque que $[R^{-1}] = [R]^T$

$[f]=[k].[\delta] \rightarrow [R]\ [F]=[k].[R]\ .[\Delta]$

$[R]^{-1}\ [R]\ [F]= [R]^{-1}\ [k].[R]\ .[\Delta]$

$[F]= [R^T]\ [k].[R]\ .[\Delta]$

Soit $[K_G]= [R^T]\ [k].[R]$

$[K_G]$ est encore une matrice symétrique. Le développement pour le cas d'une barre de treillis donne :

$$[K_G] = \frac{EA}{L} \begin{bmatrix} \cos^2\theta & \cos\theta\sin\theta & -\cos^2\theta & -\cos\theta\sin\theta \\ \cos\theta\sin\theta & \sin^2\theta & -\cos\theta\sin\theta & -\sin^2\theta \\ \cos^2\theta & -\cos\theta\sin\theta & \cos^2\theta & \cos\theta\sin\theta \\ -\cos\theta\sin\theta & -\sin^2\theta & \cos\theta\sin\theta & \sin^2\theta \end{bmatrix}$$

L'élément plan de treillis a quatre degrés de liberté - deux à chaque nœud. En conséquence, pour une structure à n nœuds, la matrice de rigidité globale K sera de taille $2n \times 2n$ (puisque nous avons deux degrés de liberté à chaque nœud). La matrice de rigidité K globale est assemblée.

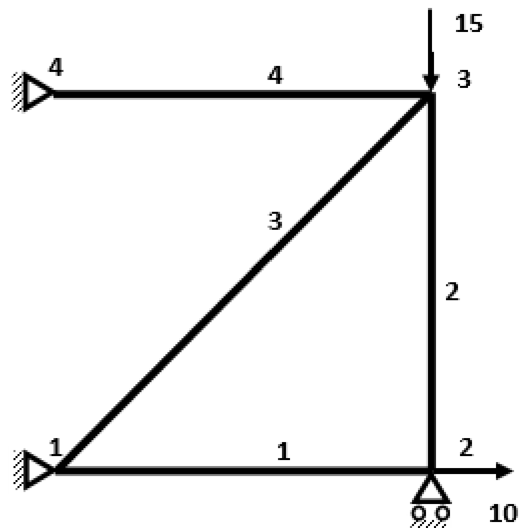
Une fois la matrice de rigidité globale K est obtenue, nous avons l'équation de la structure suivante:

$$[K]\{U\} = \{F\}$$

Où U est le vecteur de déplacement global nodal et F est vecteur de force global nodale. A cette étape, les conditions aux limites sont appliquées pour les vecteurs U et F. Puis le système d'équations linéaires obtenues est résolu par la séparation et l'élimination de Gauss. Enfin, une fois les déplacements et les réactions inconnues sont calculés, la force est obtenue pour chaque élément.

Application

Considérons le treillis plan représenté sur la figure ci-dessous.



E= 200 GPa, A=5.10^{-4} m^2, L1= L4= 10 m, L2=8 m

1. Déterminer :

- Les matrices de rigidité élémentaires.
- La matrice de rigidité globale de la structure.

2. Déterminer :

-Le déplacement horizontal du nœud 2.

-Les déplacements horizontal et vertical du nœud 3.

3. Déterminer les réactions aux nœuds 1 ,2 et 3.

Solution :

Étape 1 -La discrétisation du domaine :

Le Tableau suivant montre la connexion des éléments pour cet exemple.

Element	Noeud 1	Noeud 2
1	1	2
2	2	3
3	1	3
4	3	4

Étape 2 -Calcul des matrices de rigidité :

Les matrices de rigidité élémentaires (k1, k2, k3 et k4) sont obtenue sen faisant appel à la matrice de l'élément treillis. Chaque matrice a une taille4×4.

Elément 1

θ_1=0°

$$k_1 = \begin{bmatrix} 10000 & 0 & -10000 & 0 \\ 0 & 0 & 0 & 0 \\ -10000 & 0 & 10000 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Elément 2

θ_2=90°

$$k_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 12500 & 0 & -12500 \\ 0 & 0 & 0 & 0 \\ 0 & -12500 & 0 & 12500 \end{bmatrix}$$

Elément 3

$tg\theta_3 = 8/10 \rightarrow \theta_3 = \text{Atang}\ (8/10) = 38.6598°$

$L_3^2 = 10^2 + 8^2 \rightarrow L_3 = \sqrt{164}$

$\rightarrow L_3 = 12.80\ \text{m}$

$$k_3 = 10^3 \times \begin{bmatrix} 4.7614 & 3.8091 & -4.7614 & -3.8091 \\ 3.8091 & 3.0473 & -3.8091 & -3.0473 \\ -4.7614 & -3.8091 & 4.7614 & 3.8091 \\ -3.8091 & -3.0473 & 3.8091 & 3.0473 \end{bmatrix}$$

Elément 4

$\theta_4 = 0°$

$$k_4 = \begin{bmatrix} 10000 & 0 & -10000 & 0 \\ 0 & 0 & 0 & 0 \\ -10000 & 0 & 10000 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Étape 3 -assemblage de la matrice de rigidité globale :

Étant donné que la structure comporte quatre nœuds, la taille de la matrice de rigidité globale est de 8 x8.

$$K_G = 10^4 \times \begin{bmatrix} 1.4761 & 0.3809 & -1 & 0 & -0.4761 & -0.3089 & 0 & 0 \\ 0.3089 & 0.3047 & 0 & 0 & -0.3809 & -0.3047 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.25 & 0 & -1.25 & 0 & 0 \\ -0.4761 & 0.3089 & 0 & 0 & 1.4761 & 0.3809 & -1 & 0 \\ 0.3089 & -0.3047 & 0 & -1.25 & 0.3809 & 1.5547 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Étape 4 -Application des conditions aux limites :

La matrice pour cette structure est obtenue de la manière suivante en utilisant la matrice de rigidité globale obtenue à l'étape précédente :

$$K_G = 10^4 \times \begin{bmatrix} 1.4761 & 0.3809 & -1 & 0 & -0.4761 & -0.3089 & 0 & 0 \\ 0.3089 & 0.3047 & 0 & 0 & -0.3809 & -0.3047 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.25 & 0 & -1.25 & 0 & 0 \\ -0.4761 & 0.3089 & 0 & 0 & 1.4761 & 0.3809 & -1 & 0 \\ 0.3089 & -0.3047 & 0 & -1.25 & 0.3809 & 1.5547 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} u_{1x} \\ u_{1y} \\ u_{2x} \\ u_{2y} \\ u_{3x} \\ u_{3y} \\ u_{4x} \\ u_{4y} \end{bmatrix} = \begin{bmatrix} F_{1x} \\ F_{1y} \\ F_{2X} \\ F_{2y} \\ F_{3X} \\ F_{3y} \\ F_{4x} \\ F_{4y} \end{bmatrix}$$

Insertion des conditions aux limites

$u_{1x} = u_{1y} = 0$

$u_{4x} = u_{4y} = 0$

$u_{2y} = 0$

F_{2x}=10 kN

F_{3x}=0 kN

F3y= -15 kN

Nous obtenons :

$$K_G = 10^4 \times \begin{bmatrix} 1.4761 & 0.3809 & -1 & 0 & -0.4761 & -0.3089 & 0 & 0 \\ 0.3089 & 0.3047 & 0 & 0 & -0.3809 & -0.3047 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.25 & 0 & -1.25 & 0 & 0 \\ -0.4761 & 0.3089 & 0 & 0 & 1.4761 & 0.3809 & -1 & 0 \\ 0.3089 & -0.3047 & 0 & -1.25 & 0.3809 & 1.5547 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ u_{2x} \\ 0 \\ u_{3x} \\ u_{3y} \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{1x} \\ F_{1y} \\ 10 \\ F_{2y} \\ 0 \\ -15 \\ F_{4x} \\ F_{4y} \end{bmatrix}$$

Étape 5 -la résolution des équations :

La résolution du système d'équations sera effectuée par partitionnement et l'élimination de Gauss. Nous avons d'abord partagé le système d'équations par l'extraction des sous-matrices en ligne 3 et de la colonne 3, ligne 3et descolonnes5-6, lignes 5-6 et de la colonne 3 et des lignes 5-6 et des colonnes 5-6.

$$10^4 \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1.4761 & 0.3809 \\ 0 & 0.3809 & 1.5547 \end{bmatrix} \begin{Bmatrix} U_{2x} \\ U_{3x} \\ U_{3y} \end{Bmatrix} = \begin{Bmatrix} 10 \\ 0 \\ -15 \end{Bmatrix}$$

Par élimination de Gauss, on obtient :

u_{2x} = 0.001 m

u_{3x} = 0.0003 m

u_{3y} =- 0.001 m

Il est maintenant clair que le déplacement horizontal au nœud 2 est 0.001m (vers le droite), tandis que le déplacement horizontal au nœud 3 est 0.0003 m (vers la droite) déplacement vertical au nœud 3 est -0.001m(vers le bas).

Étape 6 -Post-traitement:

Dans cette étape, nous obtenons les réactions au niveau des nœuds1,2 et 4, dans chaque élément du treillis. Nous avons d'abord mis en place le vecteur de déplacement nodal U global, puis nous calculons le vecteur global de force nodale F.

$$F = K \times U = \begin{bmatrix} -7.34 \\ 2.12 \\ 10 \\ 12.87 \\ 0 \\ -15 \\ -2.65 \\ 0 \end{bmatrix}$$

Résolution par Python : Le code Python utilisé pour le calcul des matrices de rigidité élémentaires des éléments treillis est le suivant :

```
import math
import numpy

numpy.set_printoptions(3, suppress=True)

tn = int(input('Entrez le nombre total de nœuds : ')) #Nbre de nœuds
te = int(input('Entrez le nombre total d éléments : ')) #Nbre d'élements
xco = [] #coordonée x du noeuds
yco = [] #coordonée y du noeuds
for i in range(tn):
   x = float(input('Entrez la coordonnée x du nœud'+str(i+1)+' in m : '))
   y = float(input('Entrez la coordonnée y du nœud '+str(i+1)+' in m : '))
   xco.append(x)
   yco.append(y)


A = float(input('Entrez aire de la section transversale en m2: '))
E = float(input('Entrez le module d élasticité  : '))

snofel = [] #nœud de départ des éléments
enofel = [] #nœud d'extrémité des éléments
lenofel = [] #longueur de l'élément
elcon = [] #constant_élément
cosofel = [] #cos d element
sinofel = [] #sin d element

for i in range(te):
   a = int(input('Entrez le nœud de départ d élément '+str(i+1)+' : '))
   b = int(input('Entrez le nœud  fin d élément  '+str(i+1)+' : '))
   x1 = float(xco[a-1])
   y1 = float(yco[a-1])
   x2 = float(xco[b-1])
   y2 = float(yco[b-1])
   l = math.sqrt((x2-x1)**2+(y2-y1)**2)
   con = A*E/l
   cos = (x2-x1)/l
   sin = (y2-y1)/l

   snofel.append(a)
   enofel.append(b)
   lenofel.append(l)
   elcon.append(con)
   cosofel.append(cos)
   sinofel.append(sin)
```

```
elstmat = [] #matrice de rigidité des éléments

for i in range(te):
   cc = float(cosofel[i])**2
   ss = float(sinofel[i])**2
   cs = float(cosofel[i])*float(sinofel[i])

   mat = elcon[i]*numpy.array([[cc, cs, -cc, -cs],
                 [cs, ss, -cs, -ss],
                 [-cc, -cs, cc, cs],
                 [-cs, -ss, cs, ss]])

   elstmat.append(mat)

print('elstmat ')
```

Après exécution, on obtient :

```
Entrez le nombre total de nœuds : 4
Entrez le nombre total d éléments : 4
Entrez la coordonnée x du nœud1 in m : 0
Entrez la coordonnée y du nœud 1 in m : 0
Entrez la coordonnée x du nœud2 in m : 10
Entrez la coordonnée y du nœud 2 in m : 0
Entrez la coordonnée x du nœud3 in m : 10
Entrez la coordonnée y du nœud 3 in m : 8
Entrez la coordonnée x du nœud4 in m : 0
Entrez la coordonnée y du nœud 4 in m : 8
Entrez aire de la section transversale en m2: 5e-4
Entrez le module d élasticité  : 200e6
Entrez le nœud de départ d élément 1 : 1
Entrez le nœud  fin d élément  1 : 2
Entrez le nœud de départ d élément 2 : 2
Entrez le nœud  fin d élément  2 : 3
Entrez le nœud de départ d élément 3 : 1
Entrez le nœud  fin d élément  3 : 3
Entrez le nœud de départ d élément 4 : 3
Entrez le nœud  fin d élément  4 : 4

[array([[ 10000.,      0., -10000.,     -0.],
       [     0.,      0.,     -0.,     -0.],
       [-10000.,     -0.,  10000.,      0.],
       [    -0.,     -0.,      0.,      0.]]), array([[     0.,      0.,     -0.,     -0.],
       [     0.,  12500.,     -0., -12500.],
       [    -0.,     -0.,      0.,      0.],
       [    -0., -12500.,      0.,  12500.]]), array([[ 4761.395,  3809.116, -4761.395, -3809.116],
       [ 3809.116,  3047.293, -3809.116, -3047.293],
       [-4761.395, -3809.116,  4761.395,  3809.116],
       [-3809.116, -3047.293,  3809.116,  3047.293]]), array([[ 10000.,     -0., -10000.,      0.],
       [    -0.,      0.,      0.,     -0.],
       [-10000.,      0.,  10000.,     -0.],
       [     0.,     -0.,     -0.,      0.]])]
```

Le code Python utilisé pour le calcul de la matrice de rigidité globale des éléments treillis est le suivant :

```
gstmatmap = []                    ## Maillage_la matrice de rigidité globale, gst matmap
sera la matrice carrée de tn*
for i in range(te):              ## faire ceci pour chaque élément
   m = snofel[i]*2                ## prendre le nœud de départ de l'élément (i) et
multiplier par 2
   n = enofel[i]*2               ## prendre le nœud d'extrémité de l'élément (i) et
multiplier par 2
   add = [m-1, m, n-1, n]

   gmat = numpy.zeros((tn*2, tn*2))
   elmat = elstmat[i]
   for j in range(4):
      for k in range(4):
         a = add[j]-1
         b = add[k]-1
         gmat[a,b] = elmat[j,k]
   gstmatmap.append(gmat)
##    print(numpy.around(gmat, 3))

GSM = numpy.zeros((tn*2, tn*2))
for mat in gstmatmap:
   GSM = GSM+mat

print('Matrice de rigidité globale du treillis ')
print(numpy.around(GSM, 3))
```

Après exécution, on obtient :

```
Matrice de rigidité globale du treillis
[[ 14761.395   3809.116 -10000.         0.     -4761.395  -3809.116
      0.         0.   ]
 [  3809.116   3047.293      0.         0.     -3809.116  -3047.293
      0.         0.   ]
 [-10000.         0.     10000.         0.         0.         0.
      0.         0.   ]
 [     0.         0.         0.     12500.         0.    -12500.
      0.         0.   ]
 [ -4761.395  -3809.116      0.         0.     14761.395   3809.116
  -10000.         0.   ]
 [ -3809.116  -3047.293      0.    -12500.      3809.116  15547.293
      0.         0.   ]
 [     0.         0.         0.         0.    -10000.         0.
  10000.         0.   ]
 [     0.         0.         0.         0.         0.         0.
      0.         0.   ]]
```

Le code Python utilisé pour l'application des conditions aux limites et le chargement est le suivant :

```
#----------------------Condition aux limites et chargement --------------------#

displist = []
forcelist = []
for i in range(tn):
   a = str('u')+str(i+1)
   displist.append(a)
   b = str('v')+str(i+1)
   displist.append(b)
   c = str('fx')+str(i+1)
   forcelist.append(c)
   d = str('fy')+str(i+1)
   forcelist.append(d)


print('\n\n________________Condition aux limites______________\n')

dispmat = numpy.ones((tn*2,1))
tsupn = int(input('Entrez le nombre de nœuds bloqués : '))
supcondition = ['P = encastré',
             'H = Horizontalement bloqué',
             'V = Verticalement bloqué']

for i in range(tsupn):
   supn = int(input('Entrez le numéros du nœuds bloqué: '))
   for a in supcondition:
      print(a)
   condition = str(input('Entrez la condition du nœuds bloqué : '))
   if condition in['P', 'p']:
      dispmat[supn*2-2, 0] = 0
      dispmat[supn*2-1, 0] = 0
   elif condition in['H', 'h']:
      dispmat[supn*2-2, 0] = 0
   elif condition in['V', 'v']:
      dispmat[supn*2-1, 0] = 0
   else:
      print('Veuillez saisir des entrées valides')

print('\n_________________chargement____________________\n')
forcemat = numpy.zeros((tn*2,1))
tlon = int(input('Entrez le nombre de nœuds chargés : '))

for i in range(tlon):
   lon = int(input('Entrez le numéros du nœuds chargé: '))
   fx = float(input('Entrez la valeur de force horizontale: '))
   fy = float(input('Entrez la valeur de force verticale : '))
   forcemat[lon*2-2, 0] = fx
   forcemat[lon*2-1, 0] = fy
```

Après exécution, on obtient :

```
________________Condition aux limites______________

Entrez le nombre de nœuds bloqués : 3
Entrez le numéros du nœuds bloqué: 1
P = encastré
H = Horizontalement bloqué
V = Verticalement bloqué
Entrez la condition du nœuds bloqué : p
Entrez le numéros du nœuds bloqué: 2
P = encastré
H = Horizontalement bloqué
V = Verticalement bloqué
Entrez la condition du nœuds bloqué : v
Entrez le numéros du nœuds bloqué: 4
P = encastré
H = Horizontalement bloqué
V = Verticalement bloqué
Entrez la condition du nœuds bloqué : p

_________________chargement____________________

Entrez le nombre de nœuds chargés : 2
Entrez le numéros du nœuds chargé: 2
Entrez la valeur de force horizontale: 10
Entrez la valeur de force verticale : 0
Entrez le numéros du nœuds chargé: 3
Entrez la valeur de force horizontale: 0
Entrez la valeur de force verticale : -15
```

Le code Python utilisé pour la résolution du système d'équation linéaire ainsi que le calcul des réactions est le suivant :

```
###___________________Matrix Réduction___________________###


rcdlist = []
for i in range(tn*2):
    if dispmat[i,0] == 0:
        rcdlist.append(i)

rrgsm = numpy.delete(GSM, rcdlist, 0)
crgsm = numpy.delete(rrgsm, rcdlist, 1)
rgsm = crgsm
rforcemat = numpy.delete(forcemat, rcdlist, 0)
rdispmat = numpy.delete(dispmat, rcdlist, 0)

###_________________Résolution______________________###

dispresult = numpy.matmul(numpy.linalg.inv(rgsm), rforcemat)
rin = 0
for i in range(tn*2):
    if dispmat[i,0] == 1:
        dispmat[i,0] = dispresult[rin,0]
        rin = rin+1


forceresult = numpy.matmul(GSM, dispmat)



print('Vecteur des réactions')
print(forceresult)
```

Après exécution, on obtient :

```
Vecteur des réactions
[[ -7.342]
 [  2.126]
 [ 10.   ]
 [ 12.874]
 [ -0.   ]
 [-15.   ]
 [ -2.658]
 [  0.   ]]
```

III.6. Elément Poutre

C'est un élément unidimensionnel [IJ] qui reprend toutes les hypothèses des poutres longues. Il intègre les énergies d'effort normal, d'effort tranchant, de flexion et de torsion.

L'élément poutre est un élément à deux dimensions où les coordonnées locale et globale coïncident. Il est caractérisé par des fonctions de forme linéaire. L'élément de poutre a un module d'élasticité E, moment d'inertie I, et la longueur L. Chaque élément de poutre comporte deux nœuds et est supposé être horizontal comme représenté sur la figure ci-dessous. Dans ce cas, la matrice de rigidité de l'élément est donnée par la matrice suivante, en supposant que la déformation axiale est négligée :

$$k=\frac{EI}{L^3}\begin{bmatrix} 12 & 6L & -12 & -6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix}$$

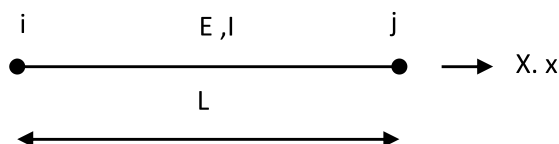
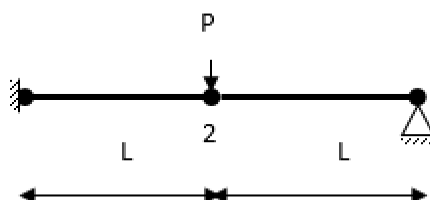


Figure III.8 : Elément poutre

L'élément poutre a quatre degrés de liberté - deux à chaque nœud (un déplacement transversal et une rotation). La convention de signe utilisée est que le déplacement est positif si il pointe vers le haut et la rotation est positive si elle est dans le sens antihoraire. En conséquence, pour une structure avec n nœuds, la matrice de rigidité globale K sera de taille ($2n \times 2n$) (puisque nous avons deux degrés de liberté à chaque nœud).

Application

Considérons la poutre représentée sur la figure ci-dessous.



Données :

$E = 210$ GPa, $I = 60 \times 10^{-6}$ m^4, $P = 20$ kN, et $L = 2$m,

Déterminer :

1. la matrice de rigidité globale de la structure.
2. le déplacement vertical au nœud 2.
3. les rotations aux nœuds 2 et 3.
4. les réactions aux nœuds 1 et 3.

Solution:

Étape 1 - La discrétisation du domaine :

Nous allons mettre un nœud (nœud 2) à l'emplacement de la force concentrée afin que nous puissions déterminer les déplacements, les rotations, les efforts tranchants et les moments) en ce point. Alternativement, nous pouvons considérer la structure comme composée d'un élément de poutre et seulement deux nœuds et utiliser les forces nodales équivalentes pour prendre en compte la charge concentrée.

Cependant, il est nécessaire de trouver le déplacement vertical sous la charge concentrée.

Par conséquent, le domaine est divisé en deux éléments, et trois nœuds. Le Tableau suivant montre la connexion de l'élément pour cet exemple.

Element	Noeud 1	Noeud 2
1	1	2
2	2	3

Étape 2 -Calcul des matrices de rigidité :

Les deux matrices de rigidité élément (k1 et k2) sont obtenues en faisant appel à la matrice de l'élément poutre. Chaque matrice a une taille 4×4.

$$k_1 = \begin{bmatrix} 18900 & 18900 & -18900 & 18900 \\ 18900 & 25200 & -18900 & 12600 \\ -18900 & -18900 & 18900 & -18900 \\ 18900 & 12600 & -18900 & 25200 \end{bmatrix}$$

$$k_2 = \begin{bmatrix} 18900 & 18900 & -18900 & 18900 \\ 18900 & 25200 & -18900 & 12600 \\ -18900 & -18900 & 18900 & -18900 \\ 18900 & 12600 & -18900 & 25200 \end{bmatrix}$$

Étape 3 - assemblage de la matrice de rigidité globale:

Étant donné que la structure comporte trois nœuds, la taille de la matrice de rigidité globale est de 6 x 6.

$$K_G = \begin{bmatrix} 18900 & 18900 & -18900 & 18900 & 0 & 0 \\ 18900 & 25200 & -18900 & 12600 & 0 & 0 \\ -18900 & -18900 & 18900+18900 & -18900+18900 & -18900 & 18900 \\ 18900 & 12600 & -18900+18900 & 25200+25200 & -18900 & 12600 \\ 0 & 0 & -18900 & -18900 & 18900 & -18900 \\ 0 & 0 & 18900 & 12600 & -18900 & 25200 \end{bmatrix}$$

$$K_G = 10^3 \times \begin{bmatrix} 18.9 & 18.9 & -18.9 & 18.9 & 0 & 0 \\ 18.9 & 25.2 & -18.9 & 12.6 & 0 & 0 \\ -18.9 & -18.9 & 37.8 & 0 & -18.9 & 18.9 \\ 18.9 & 12.6 & 0 & 50.4 & -18.9 & 12.6 \\ 0 & 0 & -18.9 & -18.9 & 18.9 & -18.9 \\ 0 & 0 & 18.9 & 12.6 & -18.9 & 25.2 \end{bmatrix}$$

Étape 4 - Application des conditions aux limites :

La matrice pour cette structure est obtenue de la manière suivante en utilisant la matrice de rigidité globale obtenue à l'étape précédente :

$$10^3 \begin{bmatrix} 18.9 & 18.9 & -18.9 & 18.9 & 0 & 0 \\ 18.9 & 25.2 & -18.9 & 12.6 & 0 & 0 \\ -18.9 & -18.9 & 37.8 & 0 & -18.9 & 18.9 \\ 18.9 & 12.6 & 0 & 50.4 & -18.9 & 12.6 \\ 0 & 0 & -18.9 & -18.9 & 18.9 & -18.9 \\ 0 & 0 & 18.9 & 12.6 & -18.9 & 25.2 \end{bmatrix} \begin{Bmatrix} U_{1y} \\ \Phi_1 \\ U_{2y} \\ \Phi_2 \\ U_{3y} \\ \Phi_3 \end{Bmatrix} = \begin{Bmatrix} F_{1y} \\ M_1 \\ F_{2y} \\ M_2 \\ F_{3y} \\ M_3 \end{Bmatrix}$$

Insertion des conditions ci-dessous nous obtenons :

U1y = φ1 = 0, F2y = –20, M2 = 0, U3y = 0, M3 = 0

$$10^3 \begin{bmatrix} 18.9 & 18.9 & -18.9 & 18.9 & 0 & 0 \\ 18.9 & 25.2 & -18.9 & 12.6 & 0 & 0 \\ -18.9 & -18.9 & 37.8 & 0 & -18.9 & 18.9 \\ 18.9 & 12.6 & 0 & 50.4 & -18.9 & 12.6 \\ 0 & 0 & -18.9 & -18.9 & 18.9 & -18.9 \\ 0 & 0 & 18.9 & 12.6 & -18.9 & 25.2 \end{bmatrix} \begin{Bmatrix} 0 \\ 0 \\ \mathrm{U}_{2y} \\ \Phi_2 \\ 0 \\ \Phi_3 \end{Bmatrix} = \begin{Bmatrix} \mathrm{F}_{1y} \\ \mathrm{M}_1 \\ -20 \\ 0 \\ \mathrm{F}_{3y} \\ 0 \end{Bmatrix}$$

Étape 5 - la résolution des équations :

La résolution du système d'équations sera effectuée par partitionnement et l'élimination de Gauss. Nous avons d'abord partager le système d’équations par l'extraction des sous-matrices en lignes 3-4 et 3-4 colonnes, lignes 3-4 et de colonne 6, ligne 6 et colonnes 3-4, et de la ligne et de la colonne 6. On obtient donc:

$$\begin{bmatrix} 37.8 & 0 & 18.9 \\ 0 & 50.4 & 12.6 \\ 18.9 & 12.6 & 25.2 \end{bmatrix} \begin{Bmatrix} \mathrm{U}_{2y} \\ \Phi_2 \\ \Phi_3 \end{Bmatrix} = \begin{Bmatrix} -20 \\ 0 \\ 0 \end{Bmatrix}$$

Par élimination de Gauss, on obtient :

$$u = 10^{-3} \times \begin{bmatrix} -0.9259 \\ -0.1984 \\ 0.7937 \end{bmatrix}$$

Il est maintenant clair que le déplacement vertical au noeud 2 est 0.9259m (vers le bas), tandis que les rotations aux nœuds 2 et 3 sont 0,1984 rad (sens horaire) et 0,7937 rad (sens anti-horaire), respectivement.

Étape 6 - Post-traitement :

Dans cette étape, nous obtenons les réactions au niveau des nœuds 1 et 3, les efforts tranchants et les moments dans chaque élément de poutre. Nous avons d'abord mis en place le vecteur de déplacement nodal U global, puis nous calculons le vecteur global de force nodale F.

$$F = K \times U = \begin{bmatrix} 13.75 \\ 15 \\ -20 \\ 0 \\ 6.25 \\ 0 \end{bmatrix}$$

Ainsi, les réactions au noeud 1 sont une force verticale de 13,75 kN (vers le haut) et un moment de 15 kN.m (sens anti-horaire) tandis que la réaction au nœud 3 est une force verticale de 6,25 kN (vers le haut).

Résolution par Python

Le code Python utilisé pour le calcul des matrices de rigidité élémentaires ainsi que la matrice globale des éléments poutre est le suivant :

```
import math
import numpy

numpy.set_printoptions(3, suppress=True)

tn = int(input('Entrez le nombre total de nœuds : ')) #Nbre de nœuds
te = int(input('Entrez le nombre total d éléments : ')) #Nbre d'élements
xco = [] #coordonée x du noeuds

for i in range(tn):
   x = float(input('Entrez la coordonnée du nœud '+str(i+1)+' in m : '))
   xco.append(x)


I = float(input('Entrez inertie de la section transversale en m3: '))
E = float(input('Entrez le module d élasticité  : '))

snofel = [] #nœud de départ des éléments
enofel = [] #nœud d'extrémité des éléments
lenofel = [] #longueur de l'élément



for i in range(te):
   a = int(input('Entrez le nœud de départ d élément '+str(i+1)+' : '))
   b = int(input('Entrer le  noeud d extrémité de l element '+str(i+1)+' :
'))
   x1 = float(xco[a-1])
   x2 = float(xco[b-1])
   l =x2-x1
   snofel.append(a)
   enofel.append(b)
   lenofel.append(l)


elstmat = [] #matrice de rigidité des éléments
for i in range(te):
   mat = ((E*I)/(l**3))*numpy.array([[12,6*l,-12,6*l],[6*l,4*l*l,-6*l,2*l*l],[-
12,-6*l,12,-6*l],[6*l,2*l*l,-6*l,4*l*l]])
   elstmat.append(mat)


print(elstmat)
```

Après exécution, on obtient :

```
Entrez le nombre total de nœuds : 3
Entrez le nombre total d éléments : 2
Entrez la coordonnée du nœud 1 in m : 0
Entrez la coordonnée du nœud 2 in m : 2
Entrez la coordonnée du nœud 3 in m : 4
Entrez inertie de la section transversale en m3: 60e-6
Entrez le module d élasticité  : 210e6
Entrez le nœud de départ d élément 1 : 1
Entrer le  noeud d extrémité de l element 1 : 2
Entrez le nœud de départ d élément 2 : 2
Entrer le  noeud d extrémité de l element 2 : 3
[array([[ 18900.,  18900., -18900.,  18900.],
       [ 18900.,  25200., -18900.,  12600.],
       [-18900., -18900.,  18900., -18900.],
       [ 18900.,  12600., -18900.,  25200.]]), array([[ 18900.,  18900., -18900.,  18900.],
       [ 18900.,  25200., -18900.,  12600.],
       [-18900., -18900.,  18900., -18900.],
       [ 18900.,  12600., -18900.,  25200.]])]
```

Le code Python utilisé pour le calcul de la matrice de rigidité globale des éléments poutres est le suivant :

```
gstmatmap = []                    ## Maillage_la matrice de rigidité globale, gst matmap sera la
matrice carrée de tn*
for i in range(te):               ## faire ceci pour chaque élément
   m = snofel[i]*2                 ## prendre le nœud de départ de l'élément (i) et multiplier par 2
   n = enofel[i]*2                ## prendre le nœud d'extrémité de l'élément (i) et multiplier par 2
   add = [m-1, m, n-1, n]

   gmat = numpy.zeros((tn*2, tn*2))
   elmat = elstmat[i]
   for j in range(4):
      for k in range(4):
         a = add[j]-1
         b = add[k]-1
         gmat[a,b] = elmat[j,k]
   gstmatmap.append(gmat)

GSM = numpy.zeros((tn*2, tn*2))
for mat in gstmatmap:
   GSM = GSM+mat


print('Matrice de rigidité globale de la poutre ')
print(numpy.around(GSM, 3))
```

Après exécution, on obtient :

```
Matrice de rigidité globale de la poutre
[[ 18900.  18900. -18900.  18900.      0.      0.]
 [ 18900.  25200. -18900.  12600.      0.      0.]
 [-18900. -18900.  37800.      0. -18900.  18900.]
 [ 18900.  12600.      0.  50400. -18900.  12600.]
 [     0.      0. -18900. -18900.  18900. -18900.]
 [     0.      0.  18900.  12600. -18900.  25200.]]
```

Le code Python utilisé pour l'application des conditions aux limites et le chargement est le suivant :

```
#----------------------Condition aux limites et chargement --------------------#

displist = []
forcelist = []
for i in range(tn):
   a = str('u')+str(i+1)
   displist.append(a)
   b = str('v')+str(i+1)
   displist.append(b)
   c = str('fx')+str(i+1)
   forcelist.append(c)
   d = str('fy')+str(i+1)
   forcelist.append(d)


print('\n\n_________________Condition aux limites_______________\n')

dispmat = numpy.ones((tn*2,1))
tsupn = int(input('Entrez le nombre de nœuds bloqués : '))
supcondition = ['P = Encastré',
            'V = Verticalement bloqué',
            'théta = Rotation bloqué']

for i in range(tsupn):
   supn = int(input('Entrez le numéros du nœuds bloqué : '))
   for a in supcondition:
      print(a)
   condition = str(input('Entrez la condition du nœuds bloqué : '))
   if condition in['P', 'p']:
      dispmat[supn*2-2, 0] = 0
      dispmat[supn*2-1, 0] = 0
   elif condition in['V', 'V']:
      dispmat[supn*2-2, 0] = 0
   elif condition in['THETA', 'théta']:
      dispmat[supn*2-1, 0] = 0
   else:
      print('Veuillez saisir des entrées valides')
print('\n__________________chargement_____________________\n')
forcemat = numpy.zeros((tn*2,1))
tlon = int(input('Entrez le nombre de nœuds chargés : '))

for i in range(tlon):
   lon = int(input('Entrez le numéros du nœuds chargé : '))
   f = float(input('Entrez la valeur de force: '))
   M = float(input('Entrez la valeur du moment : '))
   forcemat[lon*2-2, 0] = f
   forcemat[lon*2-1, 0] = M
```

Après exécution, on obtient :

```
________________Condition aux limites_____________

Entrez le nombre de nœuds bloqués : 2
Entrez le numéros du nœuds bloqué : 1
P = Encastré
V = Verticalement bloqué
théta = Rotation bloqué
Entrez la condition du nœuds bloqué : p
Entrez le numéros du nœuds bloqué : 3
P = Encastré
V = Verticalement bloqué
théta = Rotation bloqué
Entrez la condition du nœuds bloqué : V

_________________chargement____________________

Entrez le nombre de nœuds chargés : 1
Entrez le numéros du nœuds chargé : 2
Entrez la valeur de force: -20
Entrez la valeur du moment : 0
```

Le code Python utilisé pour la résolution du système d'équation linéaire ainsi que le calcul des réactions est le suivant :

```
###____________________ Reduction de matrice____________________###


rcdlist = []
for i in range(tn*2):
   if dispmat[i,0] == 0:
      rcdlist.append(i)

rrgsm = numpy.delete(GSM, rcdlist, 0)
crgsm = numpy.delete(rrgsm, rcdlist, 1)
rgsm = crgsm
rforcemat = numpy.delete(forcemat, rcdlist, 0)
rdispmat = numpy.delete(dispmat, rcdlist, 0)

###_________________Résolution_______________________###

dispresult = numpy.matmul(numpy.linalg.inv(rgsm), rforcemat)
rin = 0
for i in range(tn*2):
   if dispmat[i,0] == 1:
      dispmat[i,0] = dispresult[rin,0]
      rin = rin+1


forceresult = numpy.matmul(GSM, dispmat)


print('Matrice de rigidité globale de la poutre')
print(GSM)
print('Vecteur des déplacements')
print(dispmat)
print('Vecteur des réactions')
print(forceresult)
```

Après exécution, on obtient :

```
Vecteur des réactions
[[ 13.75]
 [ 15.  ]
 [-20.  ]
 [  0.  ]
 [  6.25]
 [  0.  ]]
```

Chapitre IV

Méthode des différences finies

IV.1. Introduction :

La méthode des différences finies est une méthode de résolutions d'équation différentielle partielle couramment pratiquée, car elle est la plus facile d'accès, puisqu'elle repose sur deux notions bien connus :

- la discrétisation des opérateurs de dérivation/différentiation (assez intuitive) d'une part,
- la convergence du schéma numérique ainsi obtenu d'autre part.

Les avantages de cette méthode sont :

- simplicité de mise en œuvre
- efficacité
- possibilité de construire des approximations d'ordre élevé
- analyse locale (simple) de la précision et de la convergence
- les autres méthodes (E.F. et V.F.) peuvent souvent s'interpréter comme des schémas différences finies dans le cas de maillage régulier.

Mais elle possède un certain nombre de limitations

- domaine de calcul simple (maillage régulier)
- transformation géométrique possible, mais plus complexe
- traitement des conditions aux limites

La méthode des différences finies est une méthode d’approximation d’équation, cela signifie que l’équation différentielle ordinaire ou aux dérivées partielles à résoudre est approchée par une équation plus facile à mettre en œuvre numériquement.

IV.2. Théorème de Taylor

Soit y(x) a n dérivés continues sur l'intervalle [a,b] et a < x, x+h < b

$$y(x+h) = y(x) + h\frac{y'(x)}{1!} + \frac{h^2}{2!}y''(x) + \cdots + \frac{h^{n-1}}{(n-1)!}y^{(n-1)}(x) + \frac{h^n}{n!}y^{(n)}(x+\theta h) \qquad \text{(IV.1)}$$

IV.3. Application du théorème de Taylor à la méthode des différences finies (cas monodimensionnel)

La première étape consiste à diviser (discrétisé) l'espace monodimensionnel [a,b] en un nombre fini d'intervalles de dimensions connues. C'est le maillage.

On remplace ensuite les dérivées apparaissant dans l'équation par des quotients aux différences obtenues à partir du développement de Taylor.

D'après ((IV.1))

$$y'(x) = \frac{y(x+h)-y(x)}{h} - \frac{h^2}{2!}y''(x) + o(h^2) \qquad \text{(IV.2)}$$

$$y'(x) = \frac{y(x+h)-y(x)}{h} - o(h^2) \qquad \text{(IV.3)}$$

$$y'(x) \cong \frac{y(x+h)-y(x)}{h} \qquad \text{(IV.4)}$$

Pour obtenir une approximation de la dérivée seconde, on procède comme suit :

$$y(x+h) = y(x) + +\frac{h^2}{2!}y''(x) + o(h^2) \qquad \text{(IV.5)}$$

$$y(x-h) = y(x) - h\,y'(x) + \frac{h^2}{2!}y''(x) + o(h^2) \qquad \text{(IV.6)}$$

On additionne les deux relations ; ce qui conduit à :

$$y(x+h) + y(x-h) = 2y(x) + h^2\,y''(x) + o(h^2). \qquad \text{(IV.7)}$$

$$y''^{(x)} = \frac{y(x+h)-2y(x)+y(x+h)}{h^2} + o(h^2) \quad \text{(IV.8)}$$

$$y''(x) \cong \frac{y(x+h)-2y(x)+y(x+h)}{h^2} \quad \text{(IV.9)}$$

L'approximation de y'' (x) est qualifiée de différence finie centrée à l'ordre 2.

IV.4. Applications :

IV.4.1 Application 1

Utiliser la méthode des différences centrales pour résoudre $\frac{d^2y}{dx^2} = -5$ avec un pas de Δx=1 ; $0 \leq x \leq 5$; y(0) = 0 et y(5)=50.

$\frac{y_{i+1}-2y_i+y_{i-1}}{\Delta x^2} = -5$

$y_{i+1} - 2y_i + y_{i-1} = -5 * \Delta x^2$

i=0 → y(0) =0

i=1 →$y_2 - 2y_1 + y_0 = -5 * \Delta x^2$

→$y_2 - 2y_1 + 0 = -5 * 1^2$

$y_2 - 2y_1 = -5$

i=2→

$y_3 - 2y_2 + y_1 = -5 * \Delta x^2$

$y_3 - 2y_2 + y_1 = -5 * 1^2$

$y_3 - 2y_2 + y_1 = -5$

i=3→$y_4 - 2y_3 + y_2 = -5 * \Delta x^2$

$y_4 - 2y_3 + y_2 = -5 * 1^2$

$y_4 - 2y_3 + y_2 = -5$

i=4→$y_5 - 2y_4 + y_3 = -5 * \Delta x^2$

$y_5 - 2y_4 + y_3 = -5 * 1^2$

$y_5 - 2y_4 + y_3 = -5$

i=5→ y(5) =50

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 0 \\ -5 \\ -5 \\ -5 \\ -5 \\ -50 \end{bmatrix}$$

y_1=20

y_2=35

y_3=45

y_4=50

Résolution par python

$$y_{i+1} - 2y_i + y_{i-1} = -5 * \Delta x^2$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 0 \\ -5\times\Delta x^2 \\ -5\times\Delta x^2 \\ -5\times\Delta x^2 \\ -5\times\Delta x^2 \\ -50 \end{bmatrix}$$

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

n = 5
h = (5-0) / n

# MATRICE An
A = np.zeros((n+1, n+1))
A[0, 0] = 1
A[n, n] = 1
for i in range(1, n):
    A[i, i-1] = 1
    A[i, i] = -2
    A[i, i+1] = 1

print("A = ",A)

# VECTEUR bn
b = np.zeros(n+1)
b[1:-1] = -5*h**2
b[-1] = 50
print("b = " ,b)

# Résolution du systéme linéaire
y = np.linalg.solve(A, b)
print("y = ",y)

x = np.linspace(0, 5, 6)

plt.figure(figsize=(10,8))
plt.plot(x, y)
plt.plot(5, 50)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Après exécution, on obtient :

```
A =  [[ 1.  0.  0.  0.  0.  0.]
 [ 1. -2.  1.  0.  0.  0.]
 [ 0.  1. -2.  1.  0.  0.]
 [ 0.  0.  1. -2.  1.  0.]
 [ 0.  0.  0.  1. -2.  1.]
 [ 0.  0.  0.  0.  0.  1.]]
b =  [ 0. -5. -5. -5. -5. 50.]
y =  [ 0. 20. 35. 45. 50. 50.]
```

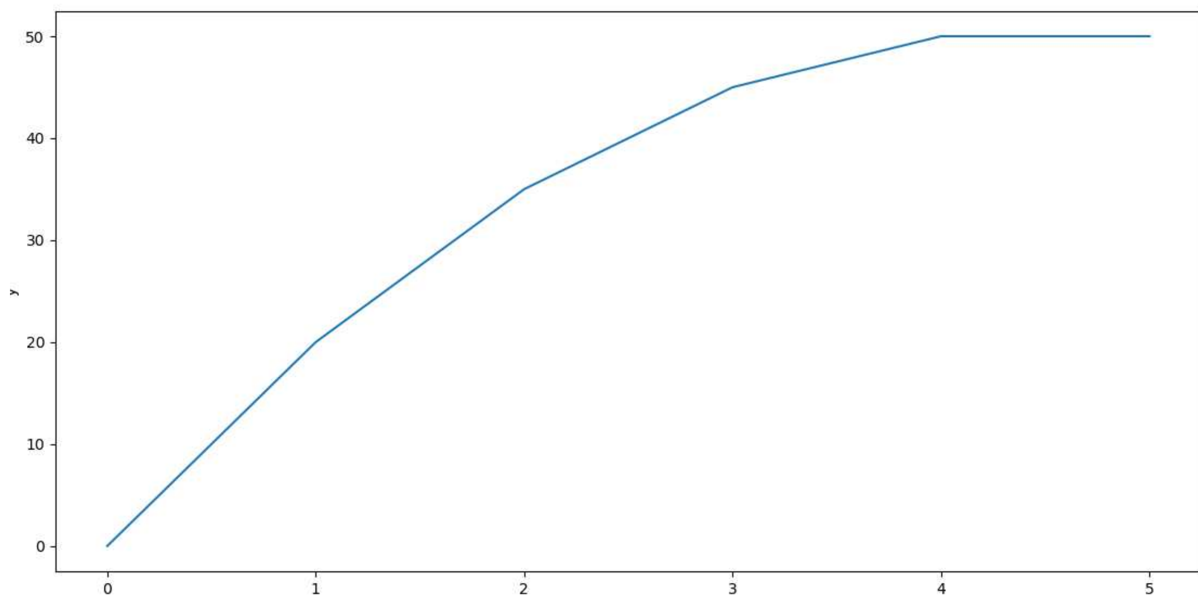


Figure IV.1 : Courbe de la solution de l'équation

IV.4.2 Application 2

Utiliser la méthode des différences centrales pour résoudre $\frac{d^2y}{dx^2} = y + x(x-4)$ avec un pas de $\Delta x=1$;

$0 \leq x \leq 4$; $y(0) = y(4)=0$.

$$\frac{y_{i+1}-2y_i+y_{i-1}}{\Delta x^2} = y_i + x_i(x_i - 4)$$

i=0 → $y(0) = 0$

i=1 →

$$\frac{y_2 - 2y_1 + y_0}{\Delta x^2} = y_1 + x_1(x_1 - 4)$$

$$\frac{y_2 - 2y_1 + 0}{1^2} = y_1 + 1(1 - 4)$$

$$y_2 - 2y_1 = y_1 - 3$$

$-3y_1+y_2=-3$

i=2→

$$\frac{y_3 - 2y_2 + y_1}{\Delta x^2} = y_2 + x_2(x_2 - 4)$$

$$\frac{y_3 - 2y_2 + y_1}{1^2} = y_1 + 2(2 - 4)$$

$y_1-3y_2+y_3=-4$

i=3

$$\frac{y_4-2y_3+y_2}{\Delta x^2}=y_3+x_3(x_3-4)$$

$$\frac{y_4-2y_3+y_2}{1^2}=y_3+3(3-4)$$

$$-2y_3+y_2=y_3+3(-1)$$

$y_2-3y_3=-3$

i=4 → y(4) =0

$$\begin{bmatrix}1&0&0&0&0\\1&-3&1&0&0\\0&1&-3&1&0\\0&0&1&-3&1\\0&0&0&0&1\end{bmatrix}\begin{bmatrix}y_0\\y_1\\y_2\\y_3\\y_4\end{bmatrix}=\begin{bmatrix}0\\-3\\-4\\-3\\0\end{bmatrix}$$

$y_1=13/7$

$y_2=18/7$

$y_3=13/7$

Résolution par Python

$\Delta x=1$;

$0 \leq x \leq 4$; $y(0) = y(4)=0$.

$$\frac{y_{i+1}-2y_i+y_{i-1}}{\Delta x^2}=y_i+x_i(x_i-4)$$

$$y_{i+1}-2y_i+y_{i-1}=y_i\Delta x^2+x_i(x_i-4)\Delta x^2$$

$$y_{i+1}-2y_i-y_i\Delta x^2+y_{i-1}=x_i(x_i-4)\Delta x^2$$

$$y_{i+1}-(2+\Delta x^2)y_i+y_{i-1}=x_i(x_i-4)\Delta x^2$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & -2-\Delta x^2 & 1 & 0 & 0 \\ 0 & 1 & -2-\Delta x^2 & 1 & 0 \\ 0 & 0 & 1 & -2-\Delta x^2 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ x_1 \times (x_1 - 4) \times \Delta \mathrm{x}^2 \\ x_2 \times (x_2 - 4) \times \Delta \mathrm{x}^2 \\ x_3 \times (x_3 - 4) \times \Delta \mathrm{x}^2 \\ 0 \end{bmatrix}$$

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

n = 4
h = (4-0) / n

x = np.linspace(0, 4, 5)
# MATRICE An
A = np.zeros((n+1, n+1))
A[0, 0] = 1
A[n, n] = 1
for i in range(1, n):
   A[i, i-1] = 1
   A[i, i] = -2-h**2
   A[i, i+1] = 1

print("A = ",A)


# VECTEUR bn
b = np.zeros(n+1)
for i in range(1, n+1):
   b[i]= x[i] * (x[i]- 4)*h**2


print("b = " ,b)

# Résolution du systéme linéaire
y = np.linalg.solve(A, b)
print("y = ",y)
x = np.linspace(0, 4, 5)

plt.figure(figsize=(10,8))
plt.plot(x, y)
plt.plot(4, 3)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Après exécution, on obtient :

```
A =  [[ 1.  0.  0.  0.  0.]
 [ 1. -3.  1.  0.  0.]
 [ 0.  1. -3.  1.  0.]
 [ 0.  0.  1. -3.  1.]
 [ 0.  0.  0.  0.  1.]]
b =  [ 0. -3. -4. -3.  0.]
```

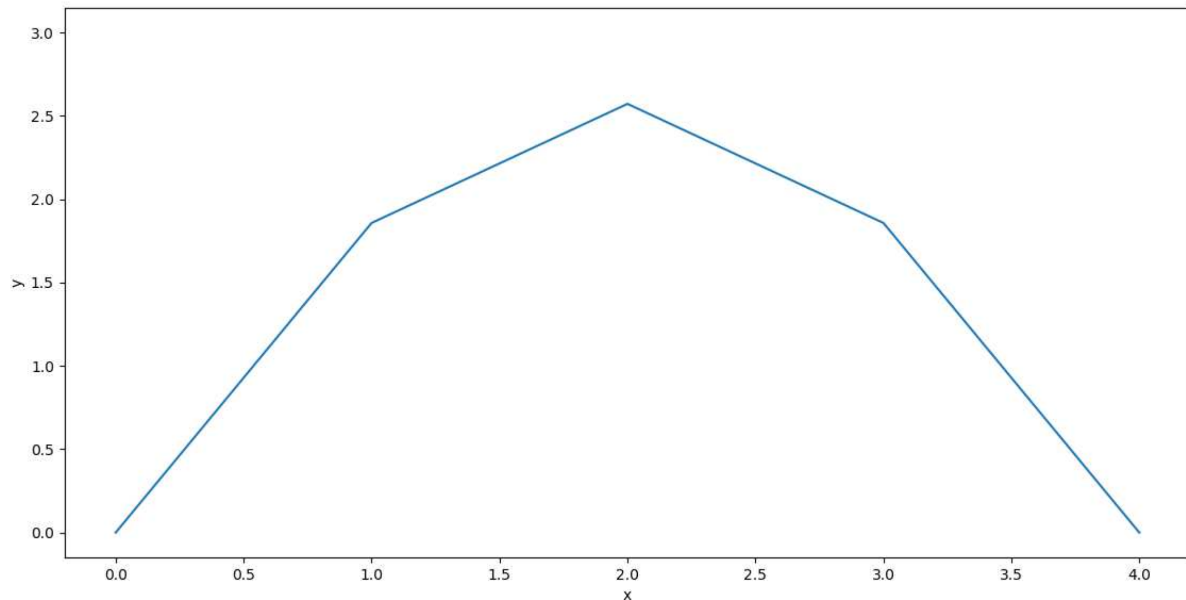
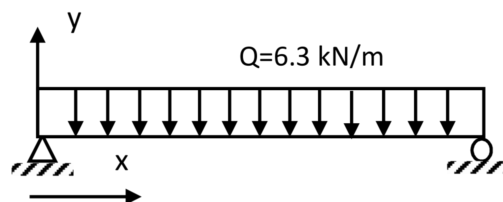


Figure IV. 2: Courbe de la solution de l'équation

IV.4.3 Application 3

Calculer la déformée pour x=5m en utilisant la méthode des différences centrales avec un pas de x=2.5 m.

E=210 GPa, I=5.10^{-6} m^4, L=7.5m ,Q=6.3 kN/m



Solution:

$$\frac{d^2y}{dx^2} = \frac{qx(l-x)}{2EI}$$

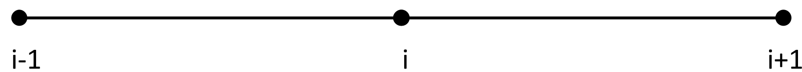
Conditions aux limites y(x=0)=0

y(x=L)=0

$$\frac{d^2y}{dx^2} = \frac{6.3x_i \times (7.5 - x)}{2 \times\ 210 \times 10^6 \times 5 \times 10^{-6}}$$

$$\frac{d^2y}{dx^2} = 0.003x_i \times (7.5 - x_i)$$

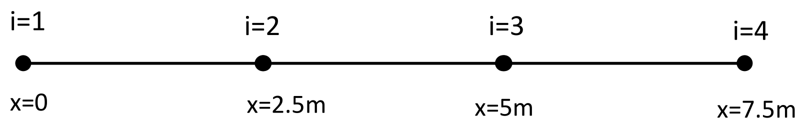
Approximation de la dérivée $\frac{d^2y}{dx^2}$ au nœud i par l'application des différences centrales



$$\frac{d^2y}{dx^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2}$$

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{(\Delta x)^2} = 0.003x_i \times (7.5 - x_i)$$

Pour un pas Δx=2.5 m , nous allons obtenir 4 nœuds :



Nœud 1 x=0→ y_1=0

Nœud 2

$$\frac{y_3 - 2y_2 + y_1}{(2.5)^2} = 0.003x_2 \times (7.5 - x_2)$$

$$\frac{y_3 - 2y_2 + y_1}{(2.5)^2} = 0.003 \times 2.5 \times (7.5 - 2.5)$$

$$0.16y_3 - 0.32y_2 + 0.16y_1 = 0.0375$$

$0.16y_{31} - 0.32y_2 + 0.16y_3 = 0.0375$

Nœud 3

$$\frac{y_4 - 2y_3 + y_2}{(2.5)^2} = 0.003x_3 \times (7.5 - x_3)$$

$$\frac{y_4 - 2y_3 + y_2}{(2.5)^2} = 0.003 \times 5 \times (7.5 - 5)$$

$0.16y_4 - 0.32y_3 + 0.16y_2 = 0.0375$

Nœud 4 $x=7.5 \rightarrow y_4=0$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.16 & -0.32 & 0.16 & 0 \\ 0 & 0.16 & -0.32 & 0.16 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0.0375 \\ 0.0375 \\ 0 \end{bmatrix}$$

$$\begin{cases} -0.32y_2+0.16y_3= 0.0375 & \mathbf{L_1} \\ 0.16y_2-0.32\ y_3=0.0375 & \mathbf{L_2} \end{cases}$$

$$\begin{cases} -0.32y_2+0.16y_3= 0.0375 & \\ 0\ -0.48\ y_3=0.0375 & \mathbf{L_1+2L_2} \end{cases}$$

$$y_3 = -\frac{0.1125}{0.48} = -0.234$$

$$y_2 = -\frac{0.0375 - 0.16(-0.234)}{0.32} = -0.234$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ -0.234 \\ -0.234 \\ 0 \end{bmatrix} \rightarrow y(5)=y(x_3) \rightarrow y_3 \approx -0.234 \text{ m}$$

La déformée pour x=5m est de -0.234 m.

Résolution par Python

$\Delta x=1$;

$0 \leq x \leq 4$; $y(0) = y(4)=0$.

$$\frac{d^2y}{dx^2} = \frac{qx(l-x)}{2EI}$$

$$\frac{y_{i+1}-2y_i+y_{i-1}}{\Delta x^2} = \frac{qx_i(l-x_i)}{2EI}$$

$$y_{i+1} - 2y_i + y_{i-1} = \frac{qx_i(l-x_i)}{2EI} \Delta x^2$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 0 \\ \frac{qx_1 \times (l-x_1)}{2EI} \times \Delta x^2 \\ \frac{qx_2 \times (l-x_2)}{2EI} \times \Delta x^2 \\ 0 \end{bmatrix}$$

```
import numpy as np
import matplotlib.pyplot as plt
E=210
I=5
q=6.3
l=7.5
n =3
h = (7.5-0) / 3
x = np.linspace(0, 7.5, 4)
# MATRICE An
A = np.zeros((n+1, n+1))
A[0, 0] = 1
A[n, n] = 1
for i in range(1, n):
   A[i, i-1] = 1
   A[i, i] = -2
   A[i, i+1] = 1

print("A = ",A)

# VECTEUR bn
b = np.zeros(n+1)
for i in range(1, n+1):
   b[i]= (q*x[i]*(l-x[i])*h**2)/(2*E*I)

print("b = " ,b)

# Résolution du systéme linéaire
y = np.linalg.solve(A, b)
print("y = ",y)
x = np.linspace(0, 7.5, 4)

plt.figure(figsize=(10,8))
plt.plot(x, y)
plt.plot(7.5, 0)
plt.xlabel('x(m)')
plt.ylabel('y(m)')
plt.show()
```

Après exécution, on obtient :

```
A =  [[ 1.  0.  0.  0.]
 [ 1. -2.  1.  0.]
 [ 0.  1. -2.  1.]
 [ 0.  0.  0.  1.]]
b =  [0.       0.234375 0.234375 0.       ]
y =  [ 0.       -0.234375 -0.234375  0.       ]
```

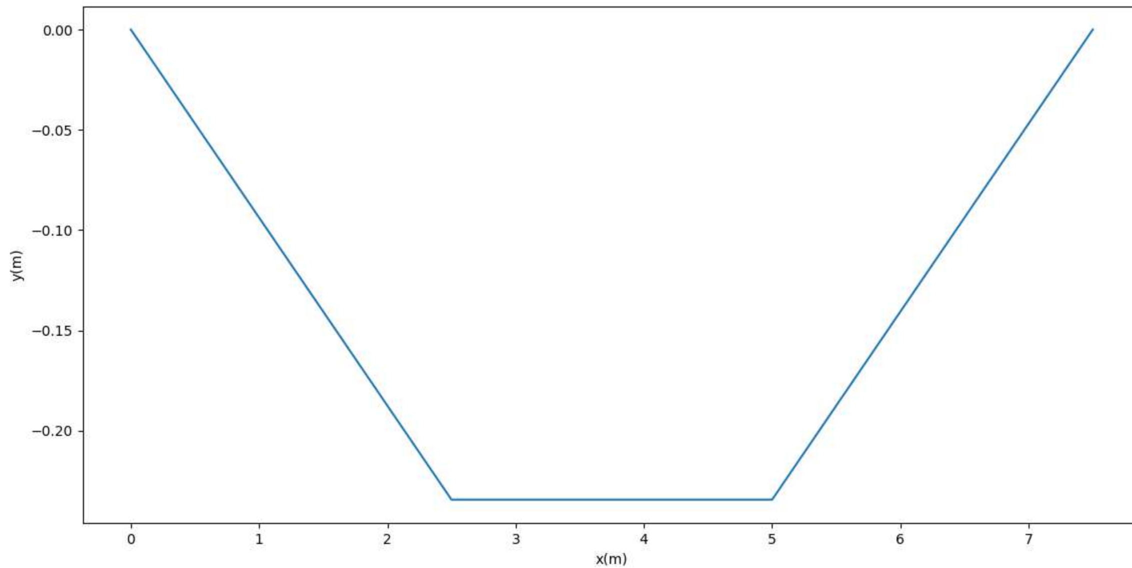


Figure IV.3 : Déformée de la poutre

IV.4.4 Application 4

Calculer la réponse temporelle libre d'un système masse-ressort en utilisant la méthode des différences centrales.

où k= 50 N/m; m = 1.5 kg, excité par une vitesse initiale de 1m/s.

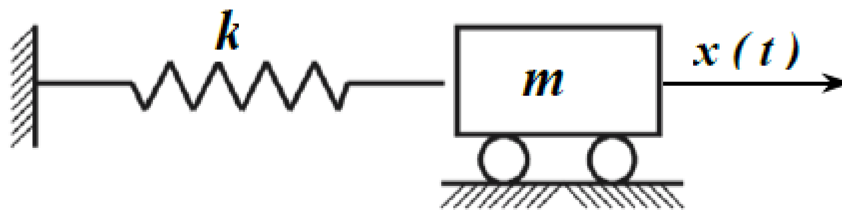


Figure IV. 4: Système non amorti

$$m\frac{d^2u}{dx^2} + ku = 0$$

$$v0 = 1$$

$$\frac{d^2u}{dx^2} = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta t^2}$$

$$m\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta t^2} + ku_i = 0$$

$$\frac{du}{dx} = \frac{u_{i+1} - u_{i-1}}{2\Delta t}$$

$$v0 = \frac{u_1 - u_{-1}}{2\Delta t}$$

$$u_1 - u_{-1} = 2\Delta t * v0$$

$$u_{-1} = u_1 - 2\Delta t * v0$$

$$u_{i+1} - 2u_i + u_{i-1} + \frac{k}{m}\Delta t^2 u_i = 0$$

$$u_1 - 2u_0 + u_{-1} + \frac{k}{m}\Delta t^2 u_0 = 0$$

$$u_1 - 2u_0 + u_1 - 2\Delta t * v0 + \frac{k}{m}\Delta t^2 u_0 = 0$$

$$2u_1 - 2u_0 - 2\Delta t * v0 + \frac{k}{m}\Delta t^2 u_0 = 0$$

$$u_1 = u_0 + \Delta t * v0 - \frac{k}{2m}\Delta t^2 u_0$$

$$u_1 = u_0 + \Delta t * v0 - \frac{k}{2m}\Delta t^2 u_0$$

$$u_{i+1} = 2u_i - u_{i-1} - \frac{k}{m}\Delta t^2 u_i$$

Résolution par Python

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

K = 50                          # Rigidité
M = 1.5                         # Masse
dt= 0.001                       # pas de temps
x0=0
v0=1
T=5
dt = float(dt)
Nt = int(round(T/dt))

x= np.zeros(Nt+1)
t = np.linspace(0, Nt*dt, Nt+1)

x[0]=x0                         # Conditions initiales

A = K / M

x[1]=x[0]*dt+dt*v0-(A/2*(dt**2)*x[0])

for k in range(1, Nt):
   x[k+1] = 2 * x[k] - x[k - 1] - dt**2 *A* x[k ]

t = np.linspace(0, Nt*dt, Nt+1)
umin = 1.2*x.min(); umax = -umin

plt.axis([t[0], t[-1], umin, umax])
plt.figure(1)
plt.plot(t, x,linewidth=2.0)
plt.xlabel('Temps (s)')
plt.ylabel('Déplacement (m)')
plt.show()
```

Après exécution, on obtient :

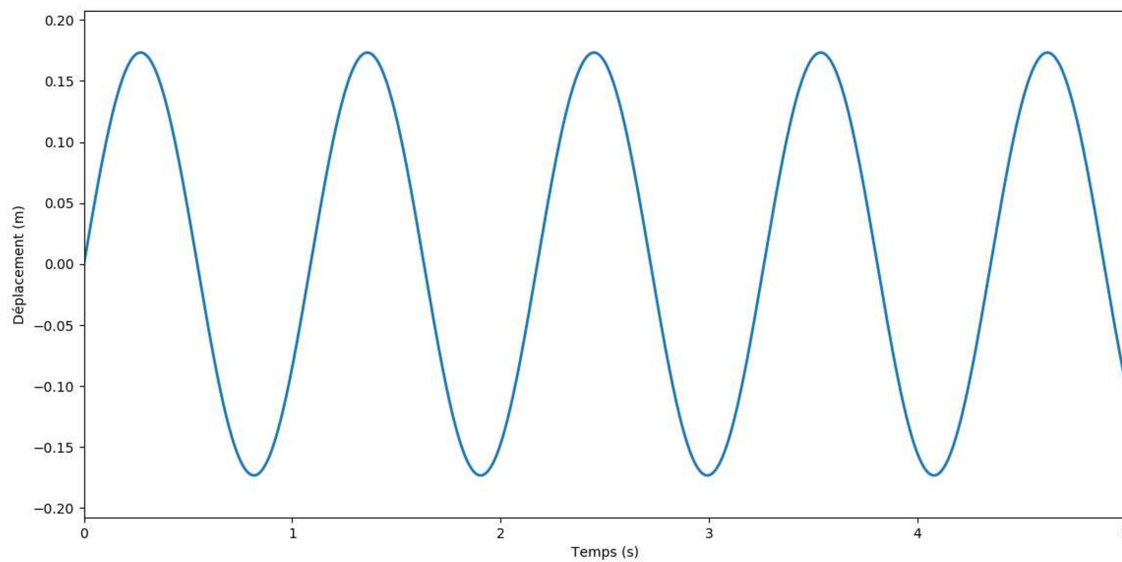


Figure IV.5 : Réponse temporelle libre

Références bibliographiques.

Python programming fundamentals.

LEE, Kent D.

London : Springer, 2011.

An Introduction to Python and computer programming.

ZHANG, Yue.

Springer, Singapore, 2015. p. 1-11.

Practical programming: an introduction to computer science using Python 3.6.

GRIES, Paul, CAMPBELL, Jennifer, et MONTOJO, Jason.

Pragmatic Bookshelf, 2017.

Méthodes numériques. Algorithmes, analyse et applications.

ALFIO, Q., RICCARDO, S., et FAUSTO, S. 2007.

Introduction aux méthodes numériques.

JEDRZEJEWSKI, Franck..
Springer Science & Business Media, 2005.

Calcul des structures par les méthodes numériques et matricielles.

WANG, Ping-Chun.
Dunod, 1969.

Structural Analysis: using classical and matrix methods.

MCCORMAC, Jack C.
John Wiley & Sons, 2006.

Aide-mémoire de mécanique des structures : Résistance des matériaux.
DELAPLACE, Arnaud, GATUINGT, Fabrice, et RAGUENEAU, Frédéric.
Hachette, 2008.

Méthodes numériques en mécanique des solides.
CURNIER, Alain.
PPUR presses polytechniques, 1993.

Finite difference computing with PDEs: a modern software approach.
PETTER LANGTANGEN, Hans et LINGE, Svein.
Springer Nature, 2017.