TP 3 - RESOLUTION D'EQUATIONS

OBJECTIFS :

- *Programmer un solveur permettant de trouver la solution x d'un système linéaire Ax = b par différentes méthodes.*
- *Résoudre par la méthode du pivot de gauss*
- *Résoudre par une méthode Numpy.linalg*
- *Exploitation d'une base de données.*

1. REVISIONS : EQUATIONS SIMPLES F(X)=0

1.1 Dichotomie

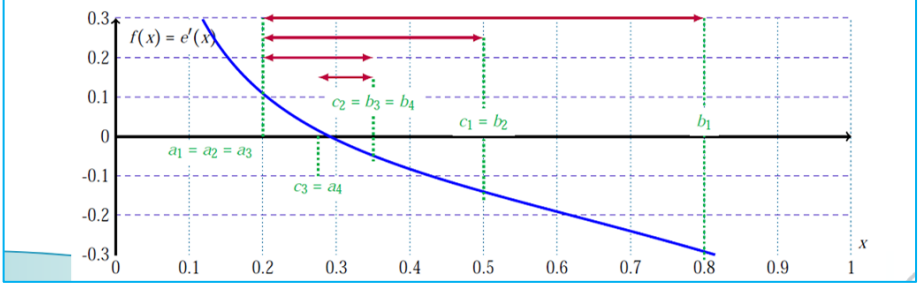
Soit n un entier naturel non nul. On étudie l'équation suivante :

$$x + \ln x - k = 0 \qquad k \in R^{+*} \qquad (1)$$

Le tableau de variation de

$f : x \to x + ln\, x$ permet d'affirmer que cette fonction réalise une bijection croissante *de* R^{+*} *vers* R. L'équation (1) possède alors pour solution unique $x_n = f^{-1}(k)$.

- *Calculer le point c milieu de l'intervalle:* $\boldsymbol{c = \frac{a+b}{2}}$
- *Évaluer* $p = f(a).f(c)$ *puis nous testons :*
  - *si p > 0, il n'y a pas de racine dans l'intervalle [a; c]. La racine est donc dans l'intervalle [c; b]. On donne alors à a la valeur de c*
  - *si p < 0, la racine est dans [a; c]. On donne alors à b la valeur de c*
  - *si p = 0, alors la racine est c. C'est fini !*
- *test d'arrêt : Évaluer le critère de convergence* $|f(c)| < \epsilon$ *ou* $|b - a| < \epsilon$
- *Recommencer si le critère de convergence n'est pas satisfait.*



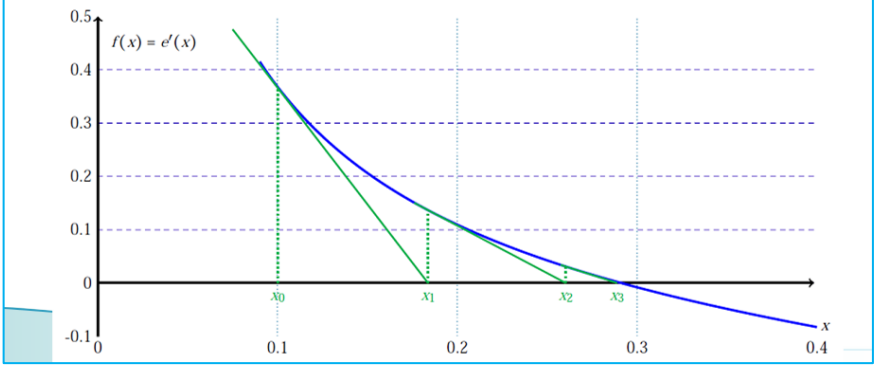
Q1. Définir la fonction qui pour les paramètres x et k renvoie $x + ln\, x - k$. *Indiquer la valeur de la fonction pour k = 1 et x = 1.*

Q2. Appliquer l'algorithme de dichotomie pour donner les valeurs de x_1, x_2, x_3, *pour k = 10*

Q3. Donner les instructions pour tracer les 100 premiers termes, indiquer la valeur de la limite

A partir d'un intervalle donné [a, b] :

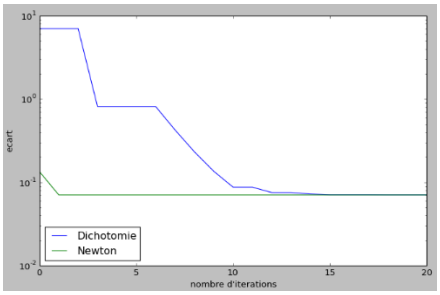
- *Choisir un premier candidat :* $\boldsymbol{x_0}$
- *test d'arrêt : Évaluer le critère de convergence* $|f(x_0)| < \epsilon$
- *Recommencer si le critère de convergence n'est pas satisfait avec un nouveau candidat* $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$



1.2 Newton

Q4. Donner les instructions pour effectuer la même méthode avec Newton

Q5. Observer l'écart entre la méthode de Newton et la dichotomie et commenter



1.3 Un peu de chimie !

Dans une réaction chimique on souhaite modéliser l'évolution de la concentration d'un réactif en fonction du temps. On a mesuré expérimentalement :

Temps(s)	0	7	18	27	37	56	102
Concentration (mol.L^{-1})	34,83	32,14	28,47	25,74	23,14	18,54	11,04

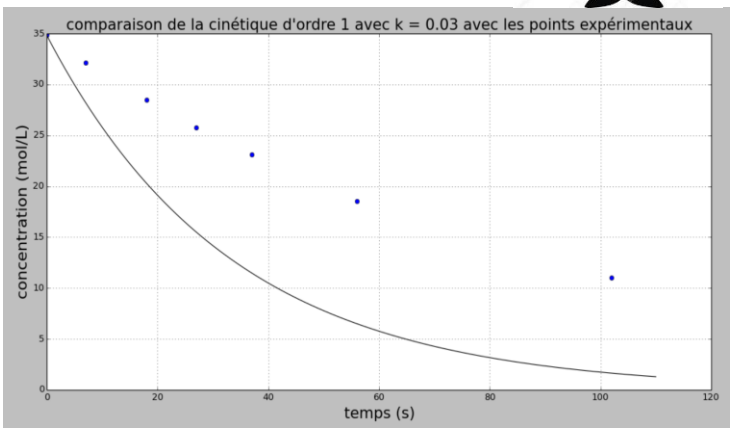
Dans la suite on notera $(T_i)_{0 \le i \le 6}$ la suite des temps considérés et $(C_i)_{0 \le i \le 6}$ la suite des concentrations mesurées.

On tente de trouver une modélisation par une réaction chimique à l'ordre 1, c'est à dire :

$$\frac{d}{dt}\big(C(t)\big) = -kC(t) \text{ qui admet pour solution } C(t) = C_0 e^{-kt}$$

Pour k = 0,03 s^{-1}. On obtient le tracé suivant :

```
import numpy as np
import matplotlib.pyplot as plt
T = np.array([0,7,18,27,37,56,102])
C =
np.array([34.83,32.14,28.47,25.74,23.14,18.54,11.
04])
Temps = np.linspace(0,110,100)
g = lambda t: 34.83*np.exp(-0.03*t)
Y = g(Temps)
plt.title('comparaison de la cinétique d\'ordre 1
avec k = 0.03 avec les points expérimentaux ',
fontsize=20)
plt.plot(T,C,'o')
plt.plot(Temps,Y,'black')
plt.grid()
```



On souhaite utiliser la méthode de Newton pour trouver l'équation de la concentration au sens des moindres carrés

Une fois choisie la constante C_0 on souhaite déterminer, s'il existe, le réel k pour laquelle la solution trouvée approche le mieux le nuage de points $(T_i, C_i)_{0 \le i \le 6}$ **au sens des moindres carrés**, c'est à dire le minimum de l'application :

$$f : k \mapsto \sum_{i=0}^{6} (C_i - C_0 e^{-kt_i})^2$$

$$f' : k \mapsto 2C_0 \sum_{i=0}^{6} t_i . (C_i - C_0 e^{-kt_i}) . e^{-kt_i}$$

$$f'' : k \mapsto 2C_0 \sum_{i=0}^{6} {t_i}^2 (2C_0 e^{-kt_i} - C_i) . e^{-kt_i}$$

```
import numpy as np
T = np.array([0,7,18,27,37,56,102])
C = np.array([34.83,32.14,28.47,25.74,23.14,18.54,11.04])
f = lambda x : sum((C-C[0]*np.exp(-T*x))**2)
df = lambda x : 2 * C[0] * sum(T * (C-C[0]*np.exp(-x*T)) * np.exp(-x*T))
ddf = lambda x : 2*C[0] * sum(T**2 * (2*C[0]*np.exp(-x*T) - C) * np.exp(-x*T))
```

*Q6. Ecrire la méthode de Newton pour le cas général **newton(f,g,u,n)** avec g la dérivée de la fonction f, u la variable de départ (x0) et n le nombre coup souhaité*

Q7. Ecrire alors les instructions qui permettent de déterminer le minimum de f sur l'intervalle I=[0,0.1]

En appelant la fonction solutionNewton (premier argument = le premier terme de la suite u_0, second argument = le nombre d'itérations) qui retourne la valeur de f'(u) et u, on obtient les résultats suivants :

solutionNewton (0.1,10) : 0.868069492978, 1.40015167377

solutionNewton (0.1,50) : 3.6877395658e-18, 7.11445082309

solutionNewton (0.1,100) : 7.11272712686e-40, 14.2573079659

solutionNewton (0.02,10) : 1.29000010674e-11, 0.0112140333598

solutionNewton (0.02,50) : 1.29000010674e-11, 0.0112140333598

solutionNewton (0.02,100) 1.29000010674e-11, 0.0112140333598

Q8. Commenter. Quelle valeur prendre pour k_{min} qui minimise f ?
Q9. Si on recherche ce minimum par dichotomie, proposer un intervalle de recherche en justifiant.

2. PROBLEME DISCRET MULTIDIMENSIONNEL LINEAIRE

Le treillis est un ensemble de barres connectées entre elles "en triangles". Ce type de structure est par exemple utilisé dans les grues et certains ponts :

Le contact entre les barres est souvent modélisé par des liaisons pivots. Par conséquent, Lorsque la structure est soumise à un chargement (par ex. une masse accrochée au bout de la grue) les barres travaillent en traction/compression.

Penser à importer les bibliothèques au début de votre programme …

```
########################################
import pylab as plt
import numpy as np
import time
###################
```

1.1 Construction de l'ensemble des points dans un graphique et des barres

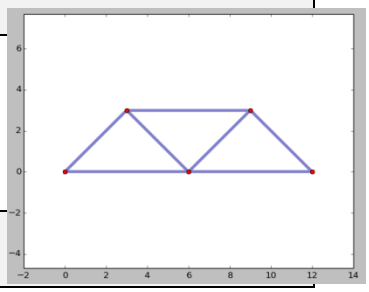
Q10. On peut choisir (par exemple) d'utiliser une liste `pivots = [(0, 0), (3, 3), (6, 0), (9, 3), (12, 0)]` *contenant la position (dans un plan) des pivots. Construire grâce à une boucle for et l'instruction plot, l'ensemble des points dans un graphique. Mettre les limites en x à (-2 et 14).*

Q11. Définir grâce à une instruction python le nombre de pivots, on appellera cette variable `N_piv`

On crée ensuite la liste des barres. On choisira les 1er point et 2ème point du treillis comme point d'accroche.

Q12. Construire comme pour les pivots la liste des barres de notre treillis, on en dénombre 7

Définir grâce à une instruction python le nombre de barres, on appellera cette variable `N_bar`.
Tracer ces barres dans un graphique.



1.2 Construction des matrices A et b

Les inconnues sont les forces de traction qui s'exercent sur chaque barre.

On parle de "Traction" pour indiquer le choix (arbitraire) de convention de signe :

- $f_j > 0$ *si la barre j est "étirée"*
- $fj < 0$ *si la barre j est "comprimée"*

Le vecteur des inconnues est $x = (f_1, \dots, f_{Nbar})$. On veut construire les matrices ***A et b*** telles que ***A.x=b***

Les équations se basent de l'application du principe de la statique à chaque pivot.

$$\sum_j \vec{F}_{(barre_j \to pivot_i)} = \vec{0} \ , \qquad pour\ chaque\ pivot\ i$$

1.3 Matrice d'incidence ayant *N_piv* lignes et *N_bar* colonnes

Il s'agit de considérer le treillis comme un graphe orienté :

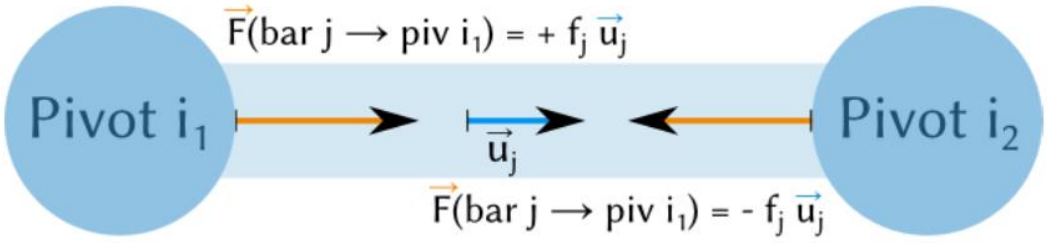
- *les sommets sont les pivots*
- *les arcs sont les barres. Chaque barre (Piv1, Piv2) sera considérée orientée de Piv1 vers Piv2.*

Cette orientation dirigera leur vecteur directeur.

On va construire la matrice d'incidence ***M*** qui contient l'information :

$m_{ij} = +1$ si la barre j "arrive" pivot i

$m_{ij} = -1$ si la barre j "quitte" pivot i



(cette barre est en traction : $f_j > 0$)

Grâce à la matrice d'incidence, on peut connaître la force exercée par une barre sur n'importe quel pivot. $\vec{F}_{(barre_j \rightarrow pivot_i)} = -M[i,j].f_j;\vec{u_j}$

Grâce à cette formule, on peut construire les équations d'équilibre génériques. Il reste à :

- *construire la matrice d'incidence*
- *calculer les vecteurs directeurs $\vec{u_j}$*

On va remplir cette matrice colonne par colonne (i.e barre par barre).

Pour trouver dans quelle ligne placer les "+1" et les "-1", on a besoin d'une méthode pour retrouver l'indice d'un élément dans une liste.

On crée alors la matrice d'incidence. Vous pourrez utiliser la fonction ***enumerate() :*** *elle permet de récupérer sur une liste ou tout autre objet itérable en même temps l'élément et son indice à chaque itération. En effet on obtient une liste de tuples (indice, valeur) en fonction du contenu de la liste, et d'une manière très pratique.*

```
#Test
for j, bar_j in enumerate(barres):
    P1, P2 = bar_j
    i1 = pivots.index(P1)
    print ( i1 )
# pivots.index((9,9)) # la recherche d'un élément inexistant génère une ValueError
```

Q13. Tester les instructions ci-dessus. Construire ensuite la matrice ayant N_piv lignes et N_bar colonnes remplie de -1 et +1 suivant la méthode décrite ci-dessus. Elle sera initialisée au préalable avec des zéros.

1.4 Chargement

Pour décrire une force extérieure (2 composantes) s'appliquant au niveau de chaque liaison, on utilise un tableau NumPy de taille `(Npiv,2)`.

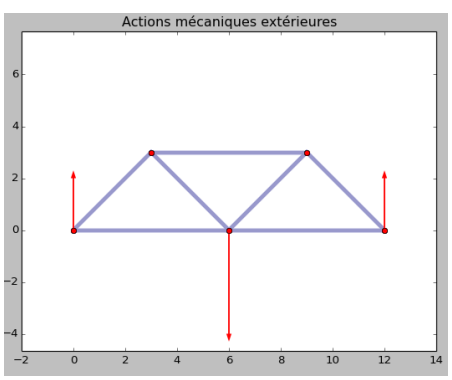
*Q14. Créer `F_ext`, un tableau de zéros de dimension N_piv * 2*

Soit une charge de 1,5 tonne au centre du pont (15KN sur y)

Q15. Définir les valeurs adéquates pour `F_ext` pour que le pont soit à l'équilibre

Pour représenter les forces extérieures avec des flèches vous pourrez vous inspirer des instructions suivantes

```
color=(0.6, 0.6, 0.8)
# Échelle pour le tracé des forces
F_scale = F_ext.max()*0.5
# Couleur des efforts:
F_color = (1., 0, 0) # rouge
dx, dy = 1,2
plt.arrow(piv[0], piv[1], F_ext[0,2]/F_scale,
F_ext[1,2]/F_scale, zorder=2, head_width=0.2*dx, lw=0,
width=0.06*dx, color=F_color)
plt.show()
```



Le résultat final du chargement (quand vous aurez adapté les instructions) doit être le suivant :

1.5 Vecteurs directeurs des barres

Pour projeter les équations de la statique selon les axes *(x,y)*, on a besoin du vecteur directeur de chaque barre.

Pour calculer le vecteur directeur, on calcule d'abord $\overrightarrow{P_1P_2} = \overrightarrow{OP_2} - \overrightarrow{OP_1}$. Il faut donc d'abord récupérer $\overrightarrow{OP_1}$ et $\overrightarrow{OP_2}$ pour chaque barre.

*Q16. Convertir les barres en tableau **(array(barres))** puis récupérer la valeur dans le tableau.*
Q17. Quelle est la forme du tableau ? Ecrire le calcul du vecteur $\overrightarrow{P_1P_2}$ Et construire le tableau `barres_P1P2` *qui contient les distances P_1P_2 pour chacune des barres.*

On détermine ensuite le vecteur normalisé intitulé *barre_dir* pour chacune des barres en utilisant les 2 étapes suivantes :

$$\vec{u} = \frac{\overrightarrow{P_1P_2}}{\left\|\overrightarrow{P_1P_2}\right\|}$$

```
# Étape 1 : calcul de la longueur des barres :
barre_l = np.sqrt(barres_P1P2[:,0]**2+barres_P1P2[:,1]**2)
# Étape 2 : Transformation en vecteur colonne :
barre_l = barre_l . reshape ( -1 ,1)
```

Q18. Calculer alors la matrice `barre_dir` *qui doit donner le résultat suivant*

```
Matrice barre_dir:
[[ 0.70710678  0.70710678]
 [ 1.          0.        ]
 [ 0.70710678 -0.70710678]
 [ 1.          0.        ]
 [ 0.70710678  0.70710678]
 [ 1.          0.        ]
 [ 0.70710678 -0.70710678]]
```

1.6 Construction de la matrice A et du vecteur b

La matrice A est fabriquée par superposition de 2 blocs :

- *A_x contient les équations d'équilibre projetées sur l'axe horizontal*
- *A_y contient les équations d'équilibre projetées sur l'axe vertical*

Nous associons ensuite les deux matrices avec ***A = np. vstack ((Ax , Ay))***

```
Inc_mat = np.zeros((N_piv, len(barres)), dtype=int)

for j, bar_j in enumerate(barres):
P1, P2 = bar_j
# Remarquons la convention de signe:
i1=pivots.index((P1))
Inc_mat[i1,j] = -1 # la barre j "quitte" P1
i2=pivots.index((P2))
Inc_mat[i2,j] = +1 # la barre j "arrive à" P2

print('Matrice d\'incidence:')
print(str(Inc_mat).replace('0','.')) # Méthode d'affichage cosmétique

#Construction de la Matrice A
Ax = Inc_mat*barre_dir[:,0]
Ay = Inc_mat*barre_dir[:,1]

#La superposition (concaténation) des blocs se fait avec np.vstack (il existe np.hstack et
np.concatenate)
A = np.vstack((Ax, Ay))
```

Le vecteur b est construit selon le même principe de concaténation que la matrice d'incidence.

```
b= np.concatenate((  ???  , ???   ))
```

Q19. Déterminer le second membre b.

3. RESOLUTION

On utilise la fonction pivot de gauss pour résoudre le système

Q20. Ecrire la fonction `pivot_gauss(A,b)` *qui renvoie la solution du problème (penser à utiliser un corrigé récemment donné !)*

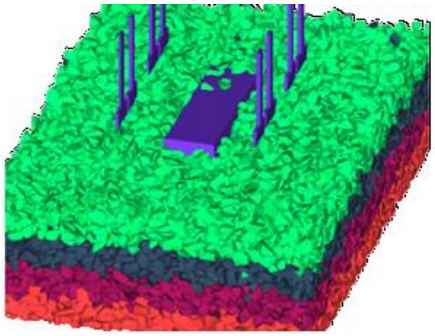
Utiliser votre fonction pour déterminer les efforts dans chaque pivot que vous stockerez dans le vecteur `solution`. Utiliser alors le programme suivant pour tracer votre résultat.

```
#affichage du résultat
#-------------------------------------------------------
#Représentation des efforts
color=(0.8, 0.8, 0.8)
# Échelle pour le tracé des forces
F_scale = 0.3*barre_l.mean()/solution.max()
# Couleur des efforts:
F_color = (0., 0, 1) # bleu
dx, dy  = 1,2

# Tracé des barres et des efforts
for j, bj in enumerate(barres):
    # Coordonnées des 2 pivots d'accroche:
    (x1,y1), (x2, y2) = bj
    # direction :
    uj = barre_dir[j]
    trac = solution[0,j]

    plt.plot((x1, x2), (y1, y2), '-', color = color, lw=4, zorder=1)

    # Tracé des efforts barre -> pivot1 et pivot 2
    plt.arrow(x1, y1, +trac*uj[0]*F_scale, +trac*uj[1]*F_scale,
              zorder=2, head_width=0.08*dx, lw=0, width=0.02*dx, color=F_color)
    plt.arrow(x2, y2, -trac*uj[0]*F_scale, -trac*uj[1]*F_scale,
              zorder=2, head_width=0.08*dx, lw=0, width=0.02*dx, color=F_color)
```

4. STOCKAGE ET COUT

Le coût de résolution d'un problème fait naturellement intervenir le temps mis pour effectuer l'ensemble des opérations, mais aussi la taille mémoire requise.

Simulation numérique d'un essai de choc sur une voiture : les cellules utilisées pour le maillage sont visibles sur la surface du véhicule



Simulation numérique du bourrage de ballast (une difficulté en plus ici : les contacts sont Non linéaires)

On rappelle que le temps peut être mesuré de deux façons :

- *On peut utiliser le temps horloge, qui est une mesure globale du temps extérieur écoulé pendant la résolution, dépendante de la machine utilisée, du problème traité et de la compilation du programme. Le résultat est alors une estimation du temps pour effectuer un programme dans sa globalité, il n'est pas uniquement lié à l'algorithme employé.* → ***Bibliothèque TIME***
- *Une deuxième mesure est le temps CPU (Central Processing Unit), cette fois lié aux opérations effectuées.* → ***Bibliothèque PROFILE***

Pour avoir un indicateur sur l'algorithme uniquement on utilise le nombre d'opérations en virgule flottante nécessaire à la résolution, nommé complexité de l'algorithme. Cependant elle ne fait pas intervenir les temps d'écriture, de transferts en mémoire...

Simple précision (32 bits)	**Exposant codé sur 8 bits, mantisse 23 bits plus 1 pour le signe.**
Double précision (64 bits)	**Exposant sur 11 bits, mantisse 52 bits plus 1 pour le signe.**

On commence par utiliser le module time. Celui-ci permet d'afficher le nombre de secondes écoulées depuis le 1er janvier 1970 à 00 :00 :00 (date de l'Unix Epoch : L'année 1970 a été considérée comme un bon départ, compte tenu de l'essor qu'a pris l'informatique à partir de cette époque.).

On place cette instruction au début et à la fin du programme à mesurer (il faut penser à enlever les affichages qui sont très chronophages que ce soit print ou plot).

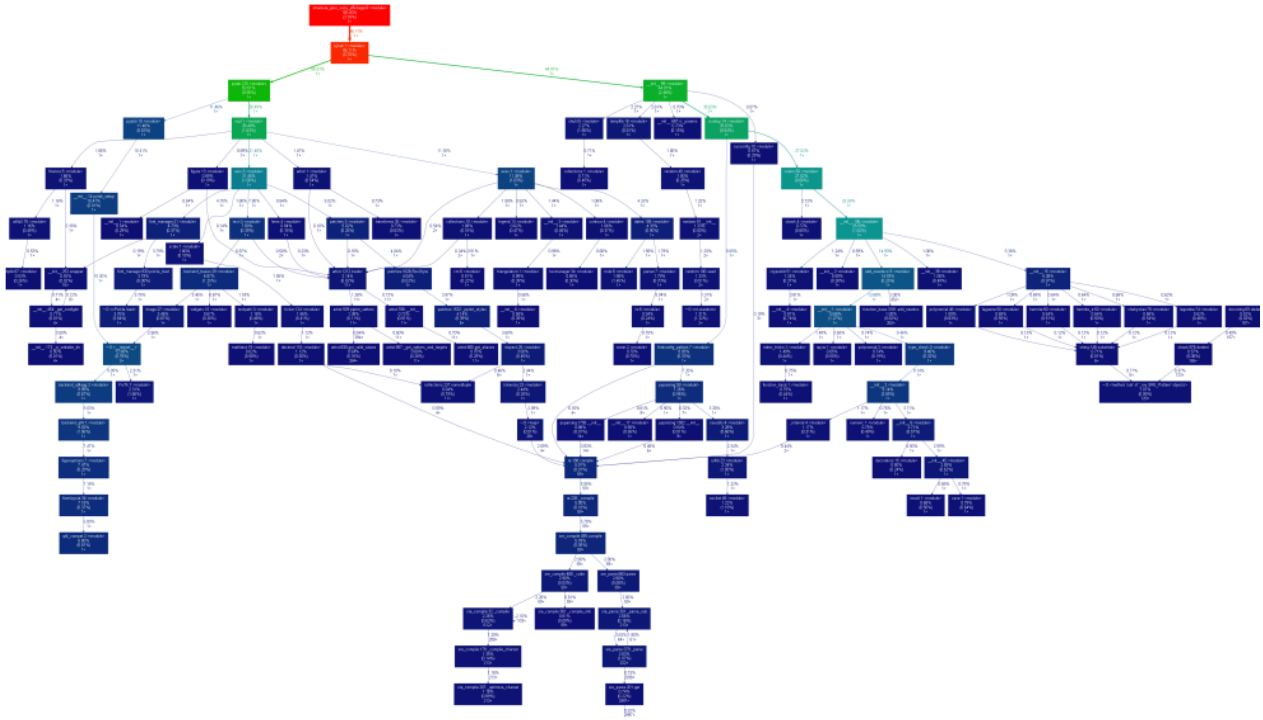
```
import time
debut = time . time ()

#Programme
fin = time . time ()

print ('temps ',fin - debut)
```

Q21. Utiliser la fonction Time pour évaluer votre temps de résolution.

Quels sont les changements à effectuer pour augmenter la taille du pont et le nombre de pivots. Quelle est l'influence de l'augmentation de la taille du problème sur le cout de résolution ?

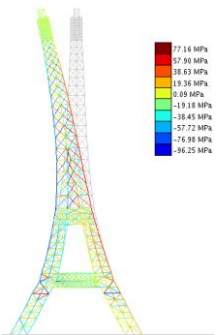
Il est aussi possible de conduire une analyse plus fine des différents temps d'utilisation CPU. On utilise pour cela ***cProfile*** qui renvoie le temps consacré à chacune des instructions utilisées dans un programme.

Le problème de cette méthode est qu'elle est limitée par un temps de rafraichissement de .001 secondes. Donc si tout se passe en moins de temps on n'est pas capable d'identifier la procédure la plus longue. On peut alors réaliser un affichage graphique des différents temps :



Graphique de l'analyse cProfile du programme

5. BASE DE DONNEES



On s'intéresse maintenant à des systèmes de treilles plus conséquent (exemple la Tour Eiffel). L'ensemble des nœuds, barres et efforts est alors stocké dans une base de données afin de déterminer les contraintes dans les barres. Une version simplifiée, réduite à deux tables, de la base de données est donnée sur la figure suivante.

NOEUDS			
id_noeuds	x_ini	y_ini	z_ini
⋮	⋮	⋮	⋮

ETAT							
id_noeuds	datem	x	y	z	Fx	Fy	Fz
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

La table NOEUDS répertorie les positions étudiées, elle contient les colonnes

- *id_noeuds (clé primaire) entier identifiant chaque noeuds ;*
- *x_ini, de type flottant, désigne la position initiale en x ;*
- *y_ini de type flottant, désigne la position initiale en x ;*
- *z_ini, de type flottant désigne la position initiale en x ;*

La table ETAT rassemble l'historique des états successifs (positions et forces subies) des noeuds étudiés. Elle est constituée de huit colonnes :

- *id_noeuds de type entier, identifie le corps concerné ;*
- *datem est la date de la mesure, sous forme d'un entier donnant le nombre de secondes écoulées depuis un instant d'origine ;*
- *trois colonnes de type flottant pour les composantes de la position x, y, z ;*
- *trois colonnes de type flottant pour les composantes des forces Fx, Fy, Fz.*

Q22. Écrire une requête SQL qui renvoie la liste des positions initiales de tous les nœuds.

Les états des différents nœuds ne sont pas forcément tous déterminés exactement au même instant.

Nous allons assimiler l'état initial (à la date t_{min}) de chaque nœud à son dernier état connu antérieur à t_{min}.

Dans toute la suite, on supposera que la valeur de t_{min}, sous le format utilisé dans la table ETAT, est accessible à toute requête SQL via l'expression $t_{min}()$.

4.1 Comptage des nœuds

On souhaite d'abord vérifier que tous les nœuds étudiés disposent d'un état connu antérieur à tmin().

Q23. Écrire une requête SQL qui renvoie le nombre de nœuds qui ont au moins un état connu antérieur à tmin().

Écrire une requête SQL qui renvoie, pour chaque nœuds, son identifiant et la date de son dernier état antérieur à tmin().

4.2 Simplification du problème pour la simulation

Le résultat de la requête précédente est stocké dans une nouvelle table date_mesure à deux colonnes :

- *id_noeuds de type entier, contient l'identifiant du corps considéré ;*
- *date_der de type entier, correspond à la date du dernier état connu du corps considéré, antérieur à tmin().*

Pour simplifier la simulation, on décide de négliger l'influence des nœuds directement reliés par une barre ayant une force dont la norme est strictement inférieure à une valeur fixée force_min().

Q24. Écrire une requête SQL qui renvoie la position, la force et la position initial (sous la forme x, y, z, Fx, Fy, Fz) de chaque nœuds retenu pour participer à la simulation. Classez les corps dans l'ordre croissant par rapport à la norme de leur force. (les fonctions SQRT et POWER(x,nb) sont disponibles en SQL)

RAPPELS EULER : RESOLUTION D'EQUA DIFF –VOIR TP1 D'INFO