Méthodes numériques

Objectifs

A la fin de la séquence d'enseignement l'élève doit pouvoir écrire et analyser les algorithmes suivants :

- créer des listes de `flot`, `boolean`, `int`
- récupérer un fichier de points
- résoudre une équation du type $f(x) = 0$.
- intégrer ou dériver une fonction ou l'image d'une fonction via une liste de points.
- intégrer une équation différentielle d'ordre n, pouvant se ramener à une équation vectorielle du type $\mathbf{Y}' = \mathbf{F}(\mathbf{Y}, t)$.
- résoudre un système linéaire inversible.

Table des matières

Introduction

Si le cours d'informatique commune permet d'établir ou détailler des méthodes numériques, en Tp de Physique, Chimie, SI ou en TIPE, il est souhaitable que les élèves puissent utiliser des méthodes numériques performantes et facilement programmables. Ce document détaille les méthodes disponibles dans les modules `matplotlib.pyplot`, `numpy` et `scipy` pour résoudre les problèmes numériques.

Afin de bien définir quelle méthode appartient à quel module, le document utilisera les alias suivants :

```
import math as m
import matplotlib.pyplot as plt
import numpy as np
import scipy as sci
import numpy.linalg as nalg
import scipy.optimize as sciop
import scipy.integrate as scint
```

- `m` pour `math`
- `plt` pour `matplotlib.pyplot`
- `np` pour `numpy`
- `sci` pour `scipy`
- `nalg` pour `numpy.linalg`
- `sciop` pour `scipy.optimize`
- `scint` pour `scipy.scint`

Remarque : il est bien sûr possible d'utiliser `from numpy import *`. Cependant il faut être vigilant ; les deux cas suivants ne conduisent pas au même résultat :

```
from math import *
from numpy import *
```

```
from numpy import *
from math import *
```

Le premier cas permet, par exemple, d'utiliser `cos` sur un tableau ; le second ne le permet pas (voir 1.1.2).

Les méthodes numériques sans l'utilisation des modules précédemment listés sont détaillées dans la partie 2.

1 Méthodes numériques en utilisant des modules

1.1 Tableaux de `floats`

1.1.1 Répartitions linéaires ou logarithmiques

Objectif : obtenir un tableau avec des valeurs réparties de façons linéaire ou logarithmique.

- `np.linspace(deb, fin, nbp)` avec `deb` la première valeur, `fin` la dernière valeur (incluse) et `nbp` le nombre d'éléments.

```
>>> np.linspace(10, 50, 5)
array([10., 20., 30., 40., 50.])
```

```
>>> np.linspace(0.1, 1.1, 5)
array([0.1 , 0.35, 0.6 , 0.85, 1.1 ])
```

- `np.arange(deb, fin, pas)` avec `deb` la première valeur, `fin` la dernière valeur (exclues) et `pas` le pas entre deux éléments.

```
>>> np.arange(10, 60, 10)
array([10, 20, 30, 40, 50])
```

```
>>> np.arange(0.1, 1.2, 0.25)
array([0.1 , 0.35, 0.6 , 0.85, 1.1 ])
```

- `np.logspace(deb, fin, nbp)` avec `10**deb` la première valeur, `10**fin` la dernière valeur (incluse) et `nbp` le nombre d'éléments.

```
>>> np.logspace(-1, 2, 4)
array([  0.1,   1. ,  10. , 100. ])
```

```
>>> np.logspace(0, 1, 3)
array([ 1.        ,  3.16227766, 10.        ])
```

1.1.2 Vectorisation

Contrairement aux listes (cf 2.1), il est possible d'appliquer des opérations directement au tableau.

```
>>> T = np.linspace(0, np.pi/2, 7)
>>> np.cos(T)
array([1.00000000e+00, 9.65925826e-01, 8.66025404e-01, 7.07106781e-01,
       5.00000000e-01, 2.58819045e-01, 6.12323400e-17])
```

1.1.3 Charger un fichier de points

Le module `numpy` propose la méthode `loadtxt` pour lire un fichier de points. Dans un fichier `.csv`, les colonnes sont séparées par des `;`. Ainsi `donnees = np.loadtxt('donnees.csv', delimiter = ';')` permet d'obtenir un tableau `donnees` avec les valeurs du fichier `donnees.csv`.

Si on connait le nombre de colonnes, il est possible d'associer une variable à chaque colonne avec l'argument optionnel `unpack`. Si les colonnes sont séparées par des tabulations, il faut utiliser `\t` comme valeur de `delimiter`.

```
t, x, v = np.loadtxt('donnees.txt', delimiter = '\t', unpack = True)
```

1.2 Résoudre $f(x) = 0$

Objectif : résoudre un problème du type $f(x) = 0$.

Pour résoudre un problème du type $f(x) = 0$, le plus simple est d'utiliser le module `scipy.optimize` avec la méthode `newton`. Ainsi, `sciop.newton(f, x0)` permet de déterminer un zéro de la fonction `f` en partant de `x0`. Il n'est pas nécessaire de donner la dérivée `fp` de `f` mais il est possible de le faire `sciop.newton(f, x0, fp)`.

```
def f(x):
    return x**2/2 - 1

def fp(x):
    return x

sol = sciop.newton(f, 3, fp)
```

Rappel il n'est pas nécessaire de déclarer les fonctions avant d'utiliser `newton`. La fonction `lambda` permet de faire lors de l'appel.

```
>>> sciop.newton(lambda x: x**2/2 - 1, 3)
1.4142135623730971
```

1.3 Intégrer une fonction ou une liste de points

Objectif : obtenir une approximation de $\int_a^b f(x).dx$

Pour obtenir une approximation d'une intégrale simple, double ou triple d'une fonction, le module `scipy.intergrate` propose les fonctions `quad`, `dblquad` et `tplquad`.

Ainsi `quad(f, a, b)` renvoie un tuple contenant une approximation de $\int_a^b f(x).dx$ et une estimation de l'erreur commise.

```
>>> scint.quad(lambda x : x**3, 0, 2)
(4.0, 4.440892098500626e-14)
```

$\int_{yg}^{yd}\left(\int_{xg}^{xd} f(y,x).dx\right).dy \approx$ `scint.dblquad(f, xg, xd, lambda x: yg, lambda x, yd)`

```
def jac2(theta, phi):
    return m.sin(theta)

A = scint.dblquad(jac2, 0, 2*m.pi, lambda x: 0, lambda x: m.pi)
```

Aire d'une sphère de rayon unitaire.

Volume d'une boule de rayon unitaire.

```
def jac3(r, theta, phi):
    return r**2*m.sin(theta)

V = scint.tplquad(jac3, 0, 2*m.pi, lambda x: 0, lambda x: m.pi,
                  lambda x, y: 0, lambda x, y : 1)
```

Utiliser `cumtrapz` du module `scipy.integrate` permet d'intégrer une liste de valeurs, par exemple pour passer des mesures d'accélération à la vitesse en chaque point de mesure.

Ainsi `scipy.intergrate.cumtrapz(val, x=t, initial=i0)` permet d'obtenir l'évolution de l'aire sous la courbe formée par les éléments de `val`, avec `t`, la liste de leurs abscisses et `i0` la valeur initiale.

Remarque : la fonction ne semble pas tenir compte de la valeur initiale. Le troisième exemple serait peut-être la solution à adopter pour obtenir le résultat recherché :

```
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=3)
array([3. , 2.5, 6. ])
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=0)
array([0. , 2.5, 6. ])
>>> scint.cumtrapz([2, 3, 4], x=[0, 1, 2], initial=0) +3
array([3. , 5.5, 9. ])
```

1.4 Intégrer une équation différentielle

Objectif : Résoudre une équation différentielle du type $\vec{Y}' = \vec{F}(\vec{Y}, t)$.

Le module `scipy.integrate` propose la fonction `odeint` telle que `odeint(F, Y0, T)` permet d'obtenir une estimation de la solution, pour chaque valeur de `T`, de l'équation différentielle $Y' = F(Y,t)$ en partant de `T[0]` avec comme valeur de l'état initial `Y0`. L'état du système peut contenir N variables avec $N \in \mathbb{N}^*$.

En ramenant l'équation différentielle scalaire du second ordre (1), à une équation différentielle vectorielle du premier ordre (2), le problème de Cauchy initial (3) peut être traité avec `odeint`.

$$\begin{cases} \frac{1}{\omega_0^2}.y''(t)+\frac{2.\xi}{\omega_0}.y'(t)+y(t)=\mathrm{K}.(u(t-t_0)-u(t-t_1)) \\ y(t_0)=y_0 \text{ et } y'(t_0)=v_0 \end{cases} \quad (1)$$

avec $u(t)$ l'échelon unitaire, nul si t est négatif, égale à 1 sinon.

$$\begin{cases} \frac{\mathrm{d}y(t)}{\mathrm{d}t} &= v(t) \\ \frac{\mathrm{d}v(t)}{\mathrm{d}t} &= (\mathrm{K}.(u(t-t_0)-u(t-t_1))-y(t)).\omega_0^2-2.\xi.\omega_0.v(t) \\ y(t_0) &= y_0 \\ v(t_0) &= v_0 \end{cases} \quad (2)$$

$$\begin{cases} \frac{\mathrm{d}\mathrm{Y}(t)}{\mathrm{d}t} &= \mathrm{F}(\mathrm{Y},t) \\ \mathrm{Y}(t_0) &= [y_0,v_0] \end{cases} \quad (3)$$

avec $\mathrm{Y}(t)=[y(t),v(t)]$

```
def sys_amort(Y, t):
    if t < t0 or t > t1:
        u = 0
    else:
        u = 1
    return [Y[1], (K*u - Y[0]) * w0**2 - 2*xi*w0*Y[1]]
```

- fonction associé au problème de Cauchy
- définition des paramètres et simulations
- présentation des résultats

```
K, xi, w0, t0, t1, y0, v0 = 3, 0.25, 20, 0, 1, 0, 0
T = np.linspace(t0-1, t1+1, 200)
sim = scint.odeint(sys_amort, [y0, v0], T)
```

```
plt.figure(0)
plt.title(r'Déplacement $y(t)$')
plt.plot(T, sim[:,0])
plt.figure(1)
plt.title(r'Vitesse $v(t)$')
plt.plot(T, sim[:,1])
```

1.5 Résoudre un système système de Cramer

Objectif : Résoudre un système linéaire du type A.X = B où A est une matrice inversible.

Pour faire de l'algèbre linéaire avec `Python`, le plus simple peut être d'utiliser le module `numpy.linalg` contenant les méthodes `det(A)` pour le calcul du déterminant de `A`, `inv(A)` pour le calcul de la matrice inverse de `A` et `solve(A,B)` pour obtenir la solution du problème linéaire A.X = B (où X est l'inconnue).

La classe `array` (tableaux `numpy`) possède beaucoup de méthodes notamment `T` pour obtenir la transposée d'un tableau et `dot(B)` pour multiplier un tableau à droite par `B`.

```
A = np.array([[2, 1, -1], [6, -1, -2], [-6, -2, 3]])
## Vérification du déterminant
print(nalg.det(A))
X = np.array([2, 3, 7]).T
## Produit matriciel de A par B
B = A.dot(X)
```

```
## Résolution par calcul
## de la matrice inverse
inv_A = nalg.inv(A)
X_sol_1 = inv_A.dot(B)
## Résolution avec solve
X_sol_2 = nalg.solve(A,B)
```

2 Méthodes numériques sans module

2.1 Listes de `floats`

2.1.1 Construire une liste de valeurs suivant une répartition linéaire

```
deb, fin, nbp = 10, 50, 5
L1 = [deb + i*(fin - deb)/(nbp - 1) for i in range(nbp)]
```

```
deb, fin, pas = 0.1, 1.2, 0.25
v , L2 = deb, []
while v < fin :
    L2.append(v)
    v += pas
```

```
deb, fin, nbp = -1, 2, 4
L5, e = [], deb
for i in range(nbp):
    L5.append(10**e)
    e += (fin - deb) / (nbp - 1)
```

2.1.2 Charger un fichier de points

Tout d'abord, il faut lire les données.

```
fich = open('donnees.csv', 'r')
data = fich.readlines()
fich.close()
```

On obtient une liste des lignes du fichier `donnees.csv`. Reste à en extraire les valeurs.

```
donnees = []
for ligne in data:
    donnees.append(ligne[:-1].split(';'))
```

Remarque : les lignes se terminent par `\n`, c'est pourquoi on élimine ce caractère en utilisant `L[:-1]`.

2.1.3 Écrire un fichier de points

On possède une liste `donnees` de listes de valeurs qu'on souhaite ouvrir dans un tableur. Pour cela, on va écrire les données dans un fichier d'extension `.csv`, en séparant les colonnes par des `;`.

```
fich = open('resultats.csv', 'w')
for ligne in donnees:
    a_imp = ''
    for e in ligne:
        a_imp += str(e) + ';'
    fich.write(a_imp[:-1] + '\n')
fich.close()
```

2.2 Résoudre $f(x) = 0$

Objectif : résoudre un problème du type $f(x) = 0$.

2.2.1 Méthodes par dichotomie

On construit la fonction `dichotomie(f, g, d, eps)` avec `f` la fonction dont on cherche un zéro, `g` et `d` respectivement les bornes gauche et droite de l'intervalle de recherche et `eps` la tolérance pour le critère d'arrêt.

Deux programmes avec test sur les antécédents. L'un en limitant le nombre de lignes ; l'autre en limitant l'appel à la fonction `f` pour le cas d'une fonction au coût d'appel élevé :

```
def dichotomie(f, g, d, eps=10**-8):
    while d - g > 2*eps:
        c = (g+d) / 2
        if f(c)*f(g) > 0:
            g = c
        else:
            d = c
    return (g+d) / 2
```

```
def dichotomie(f, g, d, eps=10**-8):
    fg = f(g)
    while d - g > 2*eps:
        c = (g+d) / 2
        fc = f(c)
        if fc * fg > 0:
            g = c
            fg = fc
        else:
            d = c
    return (g+d) / 2
```

2.2.2 Méthodes de Newton

Trois programmes pour la méthode de Newton :

- la version compacte avec critère d'arrêt sur les images
- la version avec un critère d'arrêt sur les antécédents (stagnation de l'algorithme)
- la version sans calcul de la dérivée (méthode de la fausse position)

```
def newton(f, x, fp, eps=10**-8):
    while abs(f(x)) > eps:
        x = x - f(x)/fp(x)
    return x
```

```
def newton(f, x, fp, eps=10**-8):
    x1 = x - f(x)/fp(x)
    while abs(x - x1) > eps:
        x = x1
        x1 = x - f(x)/fp(x)
    return x1
```

```
def fausse_position(f, x0, x, eps=10**-8):
    while abs(f(x)) > eps:
        nx = x - f(x)*(x-x0)/(f(x)-f(x0))
        x0 = x
        x = nx
    return x
```

2.2.3 Comparaison des vitesses de convergence

On peut considérer qu'au voisinage de la solution, il existe $(K, p) \in \mathbb{R}^2$ tel que $\varepsilon_{n+1} = K.\varepsilon_n^p$, avec ε_n l'erreur entre la solution du problème et l'approximation de la solution à la énième itération :

- $K = 0{,}5$ et p pour la largeur ε de l'intervalle de recherche de la solution avec la méthode par dichotomie
- $p = \dfrac{1+\sqrt{5}}{2} \approx 1{,}6$ (le nombre d'Or) pour la méthode de la fausse position
- $p = 2$ (vitesse quadratique) pour la méthode de Newton. A une erreur de 10^{-1}, succède une erreur de 10^{-2}, puis 10^{-4}, puis 10^{-8}...

2.3 Intégrer une fonction ou une liste de valeurs

2.3.1 Méthode de degré 0

Méthodes qui donnent un résultat vrai pour tout polynôme de degré 0 et pour lesquelles il existe au moins un polynôme de degré 1 pour lequel le résultat est faux.

Méthode des rectangles « gauche »

```
def rec_g(f, a, b, n):
    p = (b-a) / n
    x, s = a, 0
    for i in range(n):
        s += f(x)
        x += p
    return s * p
```

Méthode des rectangles « droit »

```
def rec_d(f, a, b, n):
    p = (b-a) / n
    x, s = a+p, 0
    for i in range(n):
        s += f(x)
        x += p
    return s * p
```

2.3.2 Méthode de degré 1

Méthodes qui donnent un résultat vrai pour tout polynôme de degré 1 et pour lesquelles il existe au moins un polynôme de degré 2 pour lequel le résultat est faux.

Méthode des rectangles « point milieu »

```
def rec_m(f, a, b, n):
    p = (b-a) / n
    x, s = a+p/2, 0
    for i in range(n):
        s += f(x)
        x += p
    return s * p
```

Méthode des trapèzes

```
def trapz(f, a, b, n):
    p = (b-a) / n
    x, s = a, 0
    for i in range(n):
        s += f(x) + f(x+p)
        x += p
    return s * p / 2
```

Méthode des trapèzes (limite le nombre d'appels à `f`)

```
def trapz(f, a, b, n):
    p = (b-a) / n
    x, s = a + p, (f(a) + f(b)) / 2
    for i in range(n-1):
        s += f(x)
        x += p
    return s * p
```

2.3.3 Méthode de degré 3

La méthode de Simpson donne un résultat vrai pour tout polynôme de degré 3 et il existe au moins un polynôme de degré 4 pour lequel le résultat est faux.

```
def simps(f, a, b, n):
    p = (b-a) / n
    x, s = a, 0
    for i in range(n):
        s += f(x) + f(x+p) + 4* f(x+p/2)
        x += p
    return s * p / 6
```

2.3.4 Les 5 méthodes en quelques lignes

Quelques lignes pour avoir les 5 méthodes sous la main :

- rectangle « gauche » `m = 0`
- rectangle « droit » `m = 1`
- rectangle « point milieu » `m = 0.5`

```
def rectangle(f, a, b, n, m):
    p = (b-a) / n
    x, s = a + p*m, 0
    for i in range(n):
        s += f(x)
        x += p
    return s * p
```

```
def trap_rec(f, a, b, n):
    return (rectangle(f, a, b, n, 0) + rectangle(f, a, b, n, 1)) / 2

def simp_rec(f, a, b, n):
    return (trap_rec(f, a, b, n) + 2*rectangle(f, a, b, n, 0.5)) / 3
```

2.3.5 Comparaison des vitesses de convergence

$$\int_a^b f(x).dx \approx \frac{b-a}{n}.\sum_{i=0}^{n-1}\sum_{j=0}^{m}\omega_j.f(\lambda(i,j)) \quad \text{avec} \quad x_i = a + i.\frac{b-a}{n} \quad ; \quad \lambda(i,j) \in [x_i, x_{i+1}] \quad \text{et} \quad \sum_{j=0}^{m}\omega_j = 1$$

On peut montrer, moyennant une régularité suffisante de f, qu'il existe $K \in \mathbb{R}$ tel que : $E_{rr}(f,n,N) \leq \frac{K}{n^{N+1}}$ avec N le degré de la méthode. L'évolution de l'erreur des méthodes des rectangles « gauche » ou « droit » est en $\frac{1}{n}$, quand elle est en $\frac{1}{n^2}$ pour les méthodes des trapèzes et du point milieu (elle même deux fois plus rapide que la méthode des trapèzes) et en $\frac{1}{n^4}$ pour la méthode de Simpson.

2.3.6 Intégrer une liste de valeurs

Comme vu précédemment, plus le degré de la méthode est élevé, plus le résultat sera précis (en utilisant le même nombre de points). Lorsqu'on utilise une mesure d'une grandeur et que l'on souhaite l'intégrer, il est n'est pas possible d'avoir recours au point milieu.

On utilise donc ici une méthode des trapèzes pour l'intégration numérique d'une liste de valeurs :

```
def primitive(Y, X, v0):
    V = [v0]
    for i in range(len(Y)-1):
        V.append((Y[i] + Y[i+1]) * (X[i+1]-X[i]) / 2 + V[-1])
    return V
```

2.4 Résolution d'un problème de Cauchy

Dans une premier temps, on considère un problème de Cauchy de dimension 1 et `F`, la fonction associée au problème.

2.4.1 Méthode d'Euler explicite

Méthode très facile à programmer mais diverge pour des pas trop grands. Elle s'apparente à la méthode des rectangles « gauche ».

```
def euler_explicite(F, y0, T):
    Y = [y0]
    for i in range(len(T)-1):
        Y.append(Y[i] + (T[i+1]-T[i])*F(Y[i], T[i]))
    return Y
```

2.4.2 Méthode d'Euler implicite

Ressemble à la méthode d'Euler explicite mais s'apparente à la méthode des rectangles « droit ». Son côté implicite oblige, dans le cas général, à utiliser une méthode de résolution d'un problème de type $f(x) = 0$. Ici, on triche en appelant `scipy.optimize.newton` pour ne pas avoir à donner la dérivée de `F(Y,t)` par rapport à `Y`.

```
def euler_implicite(F, y0, T):
    Y = [y0]
    for i in range(len(T)-1):
        yip1 = sciop.newton(lambda x: Y[i] + (T[i+1]-T[i])*F(x, T[i+1]) - x, Y[-1])
        Y.append(yip1)
    return Y
```

2.4.3 Méthode de Heun

C'est une méthode qui ressemble à du trapèze explicite. Pour éviter d'utiliser la valeur « à droite » (que l'on cherche ...), l'idée est de l'estimer à partir de la méthode d'Euler explicite.

La vitesse de convergence de cette méthode est très bonne.

```
def heun(F, y0, T):
    Y = [y0]
    for i in range(len(T)-1):
        k1 = F(Y[i], T[i])
        k2 = F(Y[i] + (T[i+1]-T[i])*k1, T[i+1])
        Y.append(Y[i] + (T[i+1]-T[i])*(k1+k2)/2)
    return Y
```

2.4.4 Méthode RK4

Si la méthode de Heun est la méthode RK2, la méthode RK4 donne des résultats très proches de ceux obtenus avec `odeint`. Elle s'apparente à la méthode de Simpson.

```
def rk4(F, y0, T):
    Y = [y0]
    for i in range(len(T)-1):
        k1 = F(Y[i], T[i])
        k2 = F(Y[i] + (T[i+1]-T[i])*k1/2, (T[i]+T[i+1])/2)
        k3 = F(Y[i] + (T[i+1]-T[i])*k2/2, (T[i]+T[i+1])/2)
        k4 = F(Y[i] + (T[i+1]-T[i])*k3, T[i+1])
        Y.append(Y[i] + (T[i+1]-T[i])*(k1+2*k2+2*k3+k4)/6)
    return Y
```

2.4.5 Méthode d'Euler en dimension N

La méthode d'Euler explicite en dimension N s'énonce bien même en n'utilisant que des listes. Il faut simplement penser à boucler sur la dimension du vecteur d'état.

En utilisant des tableaux `numpy`, la structure aurait été très proche de celle en dimension 1.

```
def eul_exp_N(F, Y0, T):
    N = len(Y0)
    Y = [Y0]
    for i in range(len(T)-1):
        nY = []
        for j in range(N):
            y = Y[i][j] + (T[i+1]-T[i])*F(Y[i], T[i])[j]
            nY.append(y)
        Y.append(nY)
    return Y
```

2.4.6 Méthode de Heun en dimension N

En dimension N, la méthode de Heun s'avère un peu technique mais donne des résultats très satisfaisants.

```
def heun_N(F, Y0, T):
    N = len(Y0)
    Y = [Y0]
    for i in range(len(T)-1):
        ## Estimation de Y[i+1] par Euler Explicite nY_exp
        nY_exp = []
        for j in range(N):
            y = Y[i][j] + (T[i+1]-T[i])*F(Y[i], T[i])[j]
            nY_exp.append(y)
        ## Récurrence de Heun
        nY = []
        for j in range(N):
            F_Heun_j = (F(Y[i], T[i])[j] + F(nY_exp, T[i+1])[j])/2
            y = Y[i][j] + (T[i+1]-T[i])*F_Heun_j
            nY.append(y)
        Y.append(nY)
    return Y
```

2.4.7 Méthode Runge-Kutta 4 en dimension N

En dimension N, RK-4 devient vraiment n'importe quoi à programmer uniquement avec des listes. Les tableaux `numpy` permettraient de réduire drastiquement la taille du code.

```
def rk4_N(F, Y0, T):
    N = len(Y0)
    Y = [Y0]
```

```
    for i in range(len(T)-1):
        ## Calcul de K1 = F(Y[i], T[i])
        ## pente à gauche type Euler explicite
        K1 = []
        ## et 1ère estimation de la valeur au milieu
        y_mil_1 = []
        for j in range(N):
            K1.append(F(Y[i], T[i])[j])
            y_mil_1.append(Y[i][j]+(T[i+1]-T[i])/2*K1[-1])
        ## Calcul de K2 = F(Y[i]+ h/2*F(Y[i],T[i]), (T[i]+T[i+1])/2)
        ## 1ère estimation de la pente au milieu
        K2 = []
        ## et 2ème estimation de la valeur au milieu
        y_mil_2 = []
        for j in range(N):
            K2.append(F(y_mil_1, (T[i]+T[i+1])/2)[j])
            y_mil_2.append(Y[i][j]+(T[i+1]-T[i])/2*K2[-1])
        ## Calcul de K3 = F(Y[i]+ h/2*K2, (T[i]+T[i+1])/2)
        ## 2ème estimation de la pente au milieu
        K3 = []
        ## et 1ère estimation de la valeur à droite
        y_droite = []
        for j in range(N):
            K3.append(F(y_mil_2, (T[i]+T[i+1])/2)[j])
            y_droite.append(Y[i][j]+(T[i+1]-T[i])*K3[-1])
        ## Calcul de K4 = F(Y[i]+ h*K3, T[i+1])
        ## estimation explicite de la pente à droite
        K4 = []
        for j in range(N):
            K4.append(F(y_droite, T[i+1])[j])
        ## Détermination de Y[i+1] de façon explicite
        ## avec pondération type Simpson des pentes gauche, milieu, droite
        nY = []
        for j in range(N):
            p_rk4_j = (K1[j] + 2*K2[j] + 2*K3[j] + K4[j]) / 6
            nY.append(Y[i][j] + (T[i+1]-T[i])*p_rk4_j)
        Y.append(nY)
    return Y
```

2.4.8 Comparaison des méthodes

Voir figures en première page.

2.5 Résoudre un système système de Cramer

Difficile d'égaler la puissance de `numpy`. Néanmoins, sans module, on peut faire des choses.

2.5.1 Produit matriciel

Avec `A` et `B` deux matrices représentées par des listes de listes, `prodmat(A, B)` permet d'obtenir la liste de listes issue du produit matriciel `A.B`.

```
def prodmat(A, B):
    ## dimension de A et B
    nl_A, nc_A = len(A), len(A[0])
    nl_B, nc_B = len(B), len(B[0])
    ## vérification de la compatibilité
    ## entre A et B pour le produit A.B
    assert(nc_A == nl_B)
    P = [] ## résultat du produit A.B
    for i in range(nl_A):
        P.append([]) ## nouvelle ligne
        for j in range(nc_B):
            ## nouvel élément
            ## somme des A[i,k].B[k,j]
            s = 0
            for k in range(nc_A):
                s += A[i][k]*B[k][j]
            P[i].append(s)
    return P
```

2.5.2 Opérations élémentaires sur les lignes d'une matrice (liste de listes)

`permute(A, i, j)` permute les lignes `i` et `j` de la matrice `A`.

```
def permute(A, i, j):
    nc = len(A[0])
    for k in range(nc):
        mem = A[i][k]
        A[i][k] = A[j][k]
        A[j][k] = mem
```

`multiplie(A, i, x)` multiplie la ligne `i` de la matrice `A` par `x`.

```
def multiplie(A, i, x):
    nc = len(A[0])
    for k in range(nc):
        A[i][k] *= x
```

`ajout(A, i, j, x)` ajoute la ligne `i` multipliée par `x` à la ligne `j` de la matrice `A`.

```
def ajout(A, i, j, x):
    nc = len(A[0])
    for k in range(nc):
        A[j][k] += x*A[i][k]
```

`ind_max(L)` renvoie l'indice de l'élément de `L` ayant la plus grande valeur absolue.

```
def ind_max(L):
    i_max, v_max = 0, abs(L[0])
    for i in range(1, len(L)):
        if v_max < abs(L[i]):
            i_max, v_max = i, abs(L[i])
    return i_max
```

2.5.3 Calcul de l'inverse d'une matrice

```
def inv_mat(M_originale):
    ## dimensions de M et verif matrice carrée
    nl, nc = len(M_originale), len(M_originale[0])
    assert(nl == nc)
    ## On recopie M et on créé la matrice identité
    M, I = [], []
    for i in range(nl):
        M.append([])
        I.append([])
        for j in range(nc):
            M[i].append(M_originale[i][j])
            I[i].append(0.)
        I[i][i] = 1.
    ## Triangularisation
    for j in range(nc-1):
        ## Recherche du plus gros coeff sous la diagonale
        i_max = j + ind_max([M[k][j] for k in range(j, nl)])
        permute(M, j, i_max)
        permute(I, j, i_max)
        ## vérif matrice inversible
```

```
        assert (M[j][j] !=0)
        for i in range(j+1, nl):
            ## Calcul du pivot
            pivot = - M[i][j]/M[j][j]
            ## on met des zéros sous la diagonale
            ajout(M, j, i, pivot)
            ajout(I, j, i, pivot)
    ## Diagonalisation
    for j in range(nl-1, 0, -1):
        for i in range(j):
            ## Calcul du pivot
            pivot = - M[i][j]/M[j][j]
            ## on met des zéros au dessus de la diagonale
            ajout(M, j, i, pivot)
            ajout(I, j, i, pivot)
    ## Normalisation
    for i in range(nl):
        multiplie(I, i, 1/M[i][i])
    return I
```

2.5.4 Résolution d'un problème de type A.X = B

```
def resout(A_original, B_original):
    ## dimensions de A et verif matrice carrée
    nl, nc = len(A_original), len(A_original[0])
    assert(nl == nc)
    ## dimensions de B et vérif compatibilité
    nl_B = len(B_original)
    B = [] ## On le copie pour avoir une liste de liste
    assert(nl == nl_B)
    if type(B_original[0]) == list:
        print('c est une liste')
        for i in range(nl):
            B.append([B_original[i][0]])
    else:
        for i in range(nl):
            B.append([B_original[i]])
    ## On recopie A
    M = []
    for i in range(nl):
        M.append([])
        for j in range(nc):
            M[i].append(A_original[i][j])
    ## Triangularisation
    for j in range(nc-1):
        ## Recherche du plus gros coeff sous la diagonale
        i_max = j + ind_max([M[k][j] for k in range(j, nl)])
        permute(M, j, i_max)
        permute(B, j, i_max)
        ## vérif matrice inversible
        assert (M[j][j] !=0)
        for i in range(j+1, nl):
            ## Calcul du pivot
            pivot = - M[i][j]/M[j][j]
            ## on met des zéros sous la diagonale
            ajout(M, j, i, pivot)
            ajout(B, j, i, pivot)
```

```
    ## On place des 1 sur la diagonale
    for i in range(nl):
        div = 1/M[i][i]
        multiplie(M, i, div)
        multiplie(B, i, div)
    ## Remontée
    sol = []
    for i in range(nl-1, -1, -1):
        x = B[i][0]
        for j in range(i+1, nl):
            x -= M[i][j] * sol[nl-1 - j]
        sol.append(x)
    sol.reverse()
    return sol
```

2.5.5 Calcul du déterminant

```
def determinant(M_originale):
    ## dimensions de M et verif matrice carrée
    nl, nc = len(M_originale), len(M_originale[0])
    assert(nl == nc)
    ## On recopie M
    M = []
    for i in range(nl):
        M.append([])
        for j in range(nc):
            M[i].append(M_originale[i][j])
    ## Triangularisation
    for j in range(nc-1):
        ## Recherche du plus gros coeff sous la diagonale
        i_max = j + ind_max([M[k][j] for k in range(j, nl)])
        permute(M, j, i_max)
        ## Test matrice inversible
        if M[j][j] ==0:
            return 0
        for i in range(j+1, nl):
            ## Calcul du pivot
            pivot = - M[i][j]/M[j][j]
            ## on met des zéros sous la diagonale
            ajout(M, j, i, pivot)
    valeur = 1
    for i in range(nl):
        valeur *= M[i][i]
    return valeur
```