Chapitre 1
Algorithmique numérique

Table des matières

Dans ce chapitre, nous allons étudier 3 grandes méthodes de résolution de problèmes par l'informatique. En effet, depuis le développement des ordinateurs (et la croissance exponentielle de leur capacité de calcul), il est de plus en plus rentable de simuler des systèmes physiques, biologiques, chimiques, etc par rapport à des études expérimentales. Les méthodes que vous allez voir ici ont déjà été vues l'an dernier, mais l'objectif de cette année est de vous familiariser suffisamment avec elles afin que vous soyez vous mêmes capables de les implémenter en python.

1 Résolution d'équations différentielles : méthode d'Euler

1.1 Présentation du problème

La première méthode que nous allons voir est la méthode d'Euler qui est une méthode de résolution d'équation différentielle. Comme toutes les méthodes numériques, elle est incapable de donner un résultat parfaitement exact (ne serait-ce que parce que l'ensemble des nombres que peut stocker une mémoire d'ordinateur est fini), mais en pratique, les incertitudes de mesure/réalisation des conditions initiales limitent la précision attendue, donc ce n'est pas un souci.

Nous allons donc tenter de résoudre une équation différentielle d'ordre n quelconque (elle n'a pas à être à coefficients constants, ni même linéaire). Cela signifie que cette équation différentielle sur une fonction y d'une seule variable (généralement t) peut se mettre sous la forme $y^{(n)} = f(y, y', \dots, y^{(n-1)})$: la dérivée n-ième de y peut s'écrire comme une fonction de y et de ses dérivées jusqu'à la $n-1$-ième.

Par exemple, l'équation d'évolution de pendule simple $\ddot{\theta} + \frac{g}{L}\sin(\theta) = 0$ rentre dans ce champ d'étude (car $n = 2$ et $f(\theta) = -\frac{g}{L}\sin(\theta)$), de même que celle de l'oscillateur de Van der Pol d'équation $\ddot{x} - \epsilon\omega_0(1-x^2(t))\dot{x} + \omega_0^2 x = 0$ (avec ici aussi $n = 2$, et $f(x, x', t) = \epsilon\omega_0(1-x^2(t))\dot{x} - \omega_0^2 x$), quand bien même ces équations différentielles ne sont pas linéaires et donc n'admettent pas de solutions analytiques générales, d'où l'intérêt des méthodes numériques de résolution.

1.2 Méthode d'Euler

Pour résoudre une telle équation différentielle selon la méthode d'Euler pour l'intervalle de temps de 0 à T, il est nécessaire de suivre les étapes suivantes :

- donner les n conditions initiales $y(0)$, $y'(0)$,...,$y^{(n-1)}(0)$, ainsi qu'un intervalle de temps ϵ ;
- découper l'intervalle $[0, T]$ en N intervalles de durée ϵ ;
- pour chacun des instants $k\epsilon$ (avec bien sûr $k \geqslant 0$), calculer $y^{(n)}(k\epsilon)$ grâce à la fonction f, les n valeurs $y(k\epsilon)$, $y'(k\epsilon)$,...,$y^{(n-1)}(k\epsilon)$ étant connues ;
- calculer les n valeurs des fonctions aux instants suivants $y^{(j)}((k+1)\epsilon) = y^{(j)}(k\epsilon) + \epsilon y^{(j+1)}(k\epsilon)$;
- réitérer jusqu'à avoir toutes les valeurs des $n+1$ fonctions $y, y', \dots, y^{(n)}$.

1.2.1 Exemple

Pour un pendule simple d'équation d'évolution $\ddot{\theta} + \frac{g}{L}\sin\theta = 0$ lâché sans vitesse initiale depuis la postion θ_0 :

- les conditions initiales sont $\theta(0) = \theta_0$ et $\dot{\theta}(0) = 0$;
- on calcule d'abord $\ddot{\theta}(0) = -\frac{g}{L}\sin\theta(0) = -\frac{g}{L}\sin\theta_0$;
- puis on obtient $\dot{\theta}(\epsilon) = \dot{\theta}(0) + \epsilon\ddot{\theta}(0) = -\epsilon\frac{g}{L}\sin\theta_0$ ainsi que $\theta(\epsilon) = \theta_0 + \epsilon\dot{\theta}(0) = \theta_0$;
- on peut alors attaquer la deuxième itération où l'on calcule $\ddot{\theta}(\epsilon) = -\frac{g}{L}\sin\theta(\epsilon) = -\frac{g}{L}\sin\theta_0$, puis $\dot{\theta}(2\epsilon) = \dot{\theta}(\epsilon) + \epsilon\ddot{\theta}(\epsilon) = -2\epsilon\frac{g}{L}\sin\theta_0$ ainsi que $\theta(2\epsilon) = \theta_0 + \epsilon\dot{\theta}(\epsilon) = \theta_0 - \epsilon^2\frac{g}{L}\sin\theta_0$;
- *et cætera*

1.3 Mise en œuvre

Voir TD.

1.4 Discussion

1.4.1 Influence du choix du pas du temps

Comme vous avez pu le voir sur le premier exercice du TD, le choix d u pas de temps a une influence non négligeable sur les résultats obtenus : si le pas est trop long, la solution obtenue s'éloigne trop de la solution exacte (l'approximation $f(x+h) = f(x) + hf'(x)$ devenant trop grossière quand h est grand), alors que si le pas est trop petit, le temps de calcul devient très important (pas très grave vu l'efficacité des ordinateurs actuels) et des erreurs d'arrondis peuvent s'accumuler.

1.4.2 Amélioration : méthode d'Euler implicite

Comme vu lors du troisième exercice du TD, la méthode d'Euler que nous avons vue n'est pas stable. Par exemple, la solution obtenue pour l'évolution de la tension aux bornes d'un condensateur d'un circuit RLC soumis à une tension triangle diverge aux temps longs, ce qui n'est pas physiquement acceptable.

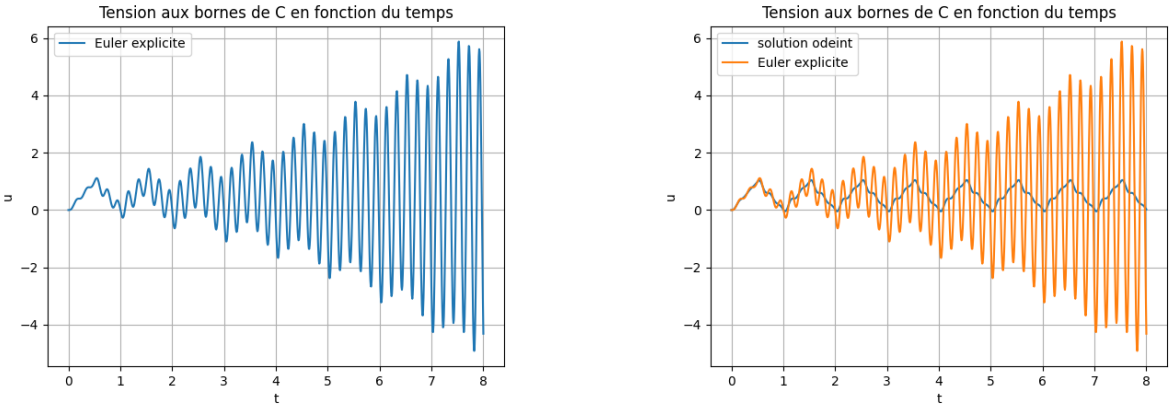

Figure 1 – Gauche : Evolution de la tension aux bornes du condensateur calculée avec la méthode d'Euler explicite. Droite : Comparaison avec la solution obtenue par la fonction *odeint* de *scipy.integrate*

Méthode *odeint* Pour la résolution d'équations différentielles, la bibliothèque `scipy.integrate` propose la fonction `odeint` dont le fonctionnement est détaillé ci-après. La connaissance de cette fonction n'est pas au programme en informatique (mais elle fait partie des compétences numériques attendues en physique-chimie).

La fonction `odeint(f,X_0,T)` a pour but de donner une solution approchée au problème posé de la manière suivante :

- on cherche à résoudre l'équation du premier ordre $X' = f(X,t)$ pour les valeurs des instants appartenant à la liste T avec la condition initiale $X(0) = X_0$;
- pour des équations d'ordre n supérieur à 1, on résout en fait pour le vecteur $X = [x, x', \dots, x^{(n-1)}]$ dont la dérivée est $X' = [x', \dots, x^{(n-1)}, x^{(n)} = f(x, x', \dots, x^{(n-1)})]$.

Ainsi, la courbe de droite de la figure 1 a été obtenue avec les instructions suivantes :

```
1  import scipy.integrate as integr
2  def fRLC(X,t) :
3      u,v = X[0],X[1]
4      return np.array([v,(triangle(t,1.) - R*C*v-u)/(L*C)])
5  T = np.linspace(0.,8.,1000)
6  S_ex = integr.odeint(fRLC,[0.,0.],T)
7  u_ex = [i[0] for i in S_ex]
```

La première ligne est l'importation de la bibliothèque. Les 3 suivantes servent à la définition de la fonction f qui donne le calcul pour la dérivée du vecteur (u, u') comme $(u', u'' = (e(t) - RCu' - u)/LC)$, et la suivante définit la liste des instants (ici, 1000 points entre 0 et 8). La sixième ligne est l'instruction utilisant `odeint` pour la résolution de l'équation différentielle avec les conditions initiales $u = 0$ et $u' = 0$. La solution donnée `S_ex` est alors un tableau de 1000 entrées (une pour chaque instant), et chaque entrée est la valeur du vecteur (u, u') à cet instant. La dernière ligne sert donc à n'extraire de `S_ex` que la première valeur afin d'obtenir `u_ex` le tableau donnant la valeur de la tension à chaque instant.

On observe alors la courbe de droite de la figure 1 (pour des valeurs des paramètres $R = 0,75$, $L = 0,1$ et $C = 0,01$), où l'on voit alors la divergence des solutions obtenues entre ces deux méthodes. On dit en particulier que la méthode d'Euler explicite est *instable*.

Méthode d'Euler implicite La raison de l'instabilité vient de l'approximation faite à chaque étape quand on calcule $X(t+\epsilon) = X(t) + \epsilon f(X(t), t)$. Ainsi la solution obtenue "suit la tangente" à chaque itération, mais pour un signal périodique, lors des passages des extrema (maximum ou minimum), la solution obtenue n'arrive pas à suivre.

Une solution que l'on peut mettre en œuvre est d'utiliser la méthode d'Euler implicite où l'on va chercher une meilleure approximation, on va en l'occurrence chercher $X(t+\epsilon)$ comme la solution de l'équation $Y = X(t) + \epsilon f(Y, t)$.

Pour résoudre l'équation, on peut utiliser la fonction `fsolve` du module `scipy.optimize` qui prend pour argument une fonction d'une variable f et un point de départ du calcul x_0 et renvoie une solution x_s proche de x_0 telle que $f(x_s) = 0$ (à la sensibilité des calculs près).

On peut alors écrire les instructions suivantes :

```
1  import scipy.optimize as sco
2  sol_t = [0.,0.]
3  sol_E_im = [sol_t]
4  for i in range(999) :
5      g = lambda y: y-sol_t-(0.008*fRLC(y,T[i+1]))
6      sol_t = sco.fsolve(g,sol_t)
7      sol_E_im.append(sol_t)
8  u_E_im = [i[0] for i in sol_E_im]
```

et obtenir les courbes de la figure 2. On observe alors que la méthode d'Euler implicite, bien qu'un peu plus complexe à mettre en œuvre (il ne suffit plus de calculer x', mais chercher la solution d'une équation) permet d'obtenir une solution plus proche de la réalité.

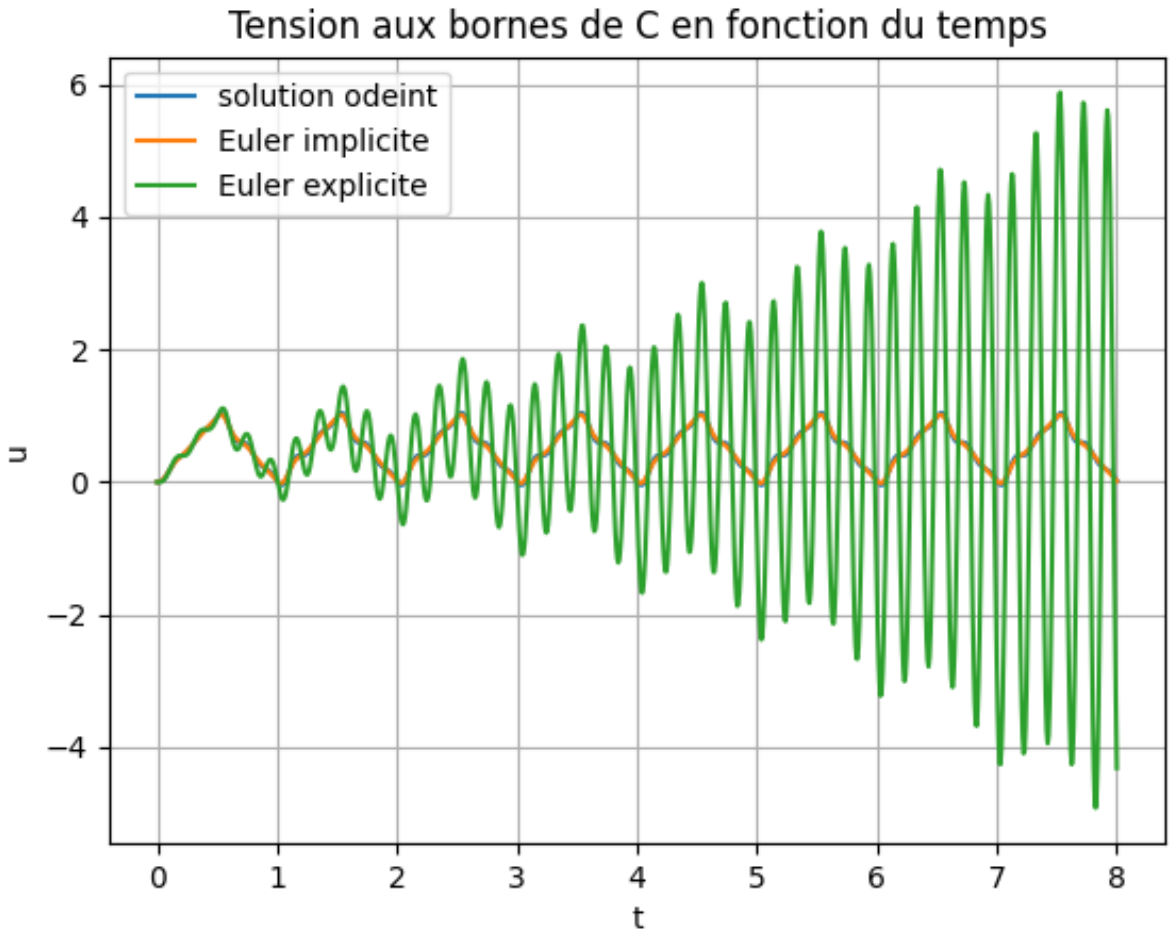

Figure 2 – Comparaison des 3 solutions obtenues

Sans rentrer dans les détails, toutes les méthodes numériques repose sur des approximations du calcul de l'intégrale dans $X(T+\epsilon) = X(T) + \int_T^{T+\epsilon} f(X(t),t)dt$. Il est possible d'obtenir une meilleure convergence des solutions en utilisant des approximations de ce calcul d'intégrale de plus en plus précises. On peut alors citer les méthodes suivantes :

- méthode d'Euler explicite : $f(X(t),t) \simeq f(X(T),t)$;
- méthode d'Euler implicite : $f(X(t),t) \simeq f(X(T+\epsilon),t)$;
- méthode de Heun : on calcule l'intégrale avec la méthode des trapèzes (on obtient la moyenne des deux méthodes d'Euler) ;
- Runge-Kutta ordre 2 : on fait d'abord Euler explicite pour avoir la valeur de $X(t+\epsilon/2)$, et on calcule alors la dérivée en ce point milieu pour aller jusqu'à $X(t+\epsilon)$;
- Runge-Kutta ordre 4 : on calcule l'intégrale avec la méthode de Simpson (à peu près équivalent aux trapèzes appliquée au point milieu avant de prolonger).

2 Résolution de systèmes linéaires inversibles : méthode du pivot de Gauss

2.1 Présentation du problème

On cherche dans ce chapitre à déterminer les solutions d'un système de n équations à n inconnues. Par exemple, pour $n = 3$, le système :

$$\begin{aligned} x + 3y - z &= 2 \\ 2x - y + 3z &= 1 \\ y + z &= 0 \end{aligned}$$

On peut alors se ramener à la recherche de la solution (vectorielle) X de l'équation $AX = Y$, avec X et Y des vecteurs de taille n et A une matrice carrée de taille $n * n$. Dans notre exemple on aurait $X = (x,y,z)$, $Y = (2,1,0)$ et $A = \begin{pmatrix} 1 & 3 & -1 \\ 2 & -1 & 3 \\ 0 & 1 & 1 \end{pmatrix}$.

D'après le cours de mathématiques, si la matrice A est inversible, il y a alors une unique solution $X = A^{-1}Y$, et nous allons déterminer cette solution, sans avoir à calculer l'inverse de A : la méthode du pivot de Gauss.

Pour fixer les idées, voici comment nous procéderions avec l'exemple :

- on commence par enlever les dépendances en x de toutes les lignes sauf la première. Ici, cela consiste à retrancher deux fois la première ligne à la seconde, et laisser la troisième inchangée. On a lors le système :

$$\begin{aligned} x + 3y - z &= 2 \\ -7y + 5z &= -3 \\ y + z &= 0 \end{aligned}$$

- on enlève ensuite les dépendances en y de la dernière ligne en lui ajoutant $1/7$ de la deuxième ligne pour obtenir :

$$\begin{aligned} x + 3y - z &= 2 \\ -7y + 5z &= -3 \\ \frac{12}{7}z &= -\frac{3}{7} \end{aligned}$$

- on résout alors la dernière ligne $z = -\frac{1}{4}$, qu'on injecte dans la deuxième pour trouver $y = \frac{1}{4}$, et on injecte alors les résultats dans la première ligne pour trouver $x = 2 - 3y + z = 1$. Cette dernière étape s'appelle la remontée.

On voit donc que dans la première partie de l'algorithme, il suffit en partant de deux lignes L_i et L_j d'une matrice d'arriver à ajouter α fois la i-ème ligne à la j-ème. Cette opération s'appelle une transvection.

On peut aussi s'apercevoir que si le système de départ était écrit différemment, la première partie peut bloquer. Par exemple, avec le système équivalent :

$$\begin{aligned} y + z &= 0 \\ x + 3y - z &= 2 \\ 2x - y + 3z &= 1 \end{aligned}$$

il n'est pas possible d'éliminer x des dernières lignes en ajoutant un multiple de la première. Il faudrait dans ce cas inverser deux lignes (par exemple les deux premières) et on aura aussi besoin de l'opération qui échange deux lignes d'une matrice.

On remarque aussi que les opérations subies par A doivent aussi être subies par Y, donc on écrira ces fonctions de telle sorte qu'elles fonctionnent avec des matrices de taille quelconque.

En Python, nous coderons les matrices comme des listes de listes, par exemple $A = [[1,3,-1],[2,-1,3],[0,1,1]]$ et $Y = [[2],[1],[0]]$.

On utilisera donc les deux fonctions :

```
def transvection(A,i,j,alpha) :
    m = len(A[0]) #nombre de colonnes de A
    for k in range(m) :
        A[j][k] = A[j][k] + alpha A[i][k]

def echange_lignes(A,i,j) :
    A[i],A[j] = A[j],A[i]
```

2.2 Mise sous forme triangulaire supérieure de A

Imaginons une matrice carrée A de taille n sur laquelle les i premières étapes ont été faites. On a alors une matrice du type :

$$\begin{pmatrix} \lambda_0 & * & * & \cdots & * & \cdots & * & * \\ 0 & \lambda_1 & * & \cdots & * & \cdots & * & * \\ 0 & 0 & \lambda_2 & * & * & \cdots & * & * \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & \lambda_i & * & \cdots & * \\ 0 & 0 & \cdots & 0 & * & * & \cdots & * \\ \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 & * & * & \cdots & * \end{pmatrix}$$

où les étoiles peuvent représenter n'importe quel nombre. L'étape suivante consiste donc à faire des transvections afin de remplacer les étoiles sous λ_i par des zéros.

Cependant, il est possible que λ_i soit nul, donc on pourrait imaginer tester cette condition, mais de manière générale, à cause de la manière dont sont stockés les flottants, les tests d'égalité sont une mauvaise idée. De plus, pour la ligne j dont le coefficient sous λ_i vaut $A[j][i]$, on doit faire une transvection avec $\alpha = -A[j][i]/\lambda_i$, et dans le cas où λ_i est petit, on risque d'avoir des erreurs d'arrondis qui deviennent considérables.

On va donc à cette étape chercher la ligne j dont le coefficient sous λ_i est le plus grand en valeur absolue, et échanger les lignes i et j. On appelle cette méthode la *méthode du pivot partiel*.

On écrit donc la fonction suivante qui prend comme argument une matrice A et un indice i, et qui renvoie l'indice de la ligne $j \geqslant i$ dont le i-ème coefficient est le plus grand en valeur absolue :

```
def indice_pivot(A,i) :
    n = len(A)
    aj = abs(A[i][i])
    j = i
    for k in range(i+1,n) :
        if abs(A[k][i]) > aj :
            aj = abs(A[k][i])
            j = k
    return j
```

On a donc maintenant tout en mains pour écrire un programme qui met les système d'équations sous la forme attendue (il ne faut pas oublier de faire les mêmes opérations sur Y).

```
def forme_triangulaire(A,Y) :
    n = len(A)
    for i in range(n) :
        j = indice_pivot(A,i)
        if j!=i :
            echange_lignes(A,i,j)
            echange_lignes(Y,i,j)
```

```
8           for j in range(i+1,n) :
9               alpha = - A[j][i]/A[i][i]
10              transvection(A,i,j,alpha)
11              transvection(Y,i,j,alpha)
12      return [A,Y]
```

2.3 Phase de remontée

Après la mise sous forme triangulaire, on doit résoudre le système $BX = W$ où B (respectivement W) a été obtenue à partir de A (resp.Y) lors du passage sous forme triangulaire supérieure. B est donc une matrice triangulaire supérieure (tous les coefficients sous la diagonale sont nuls).

Si on appelle x_i, y_i, b_{ij} les coordonnées du vecteur X solution, de Y et de la matrice B, on doit avoir quel que soit i :

$$\sum_{j=1}^{n} b_{ij}x_j = w_i \quad \Longleftrightarrow \sum_{j=i}^{n} b_{ij}x_j = w_i.$$

On peut donc remonter les résultats en partant de x_n puisque si l'on connait tous les x_j pour $j > i$, alors on peut écrire :

$$x_i = \frac{w_i - \sum_{j>i} b_{ij}x_j}{b_{ii}}.$$

On peut donc écrire le programme suivant qui prend une matrice triangulaire supérieur B et un vecteur W comme arguments et renvoie la solution X de l'équation $BX = W$:

```
1  def remontee(B,W) :
2      n = len(B)
3      X = [[0] for i in range(n)]
4      for i in range(n-1,-1,-1) :
5          s= 0.
6          for j in range(i+1,n) :
7              s+= B[i][j]*X[j][0]
8          X[i] = [(W[i][0] - s)/B[i][i]]
9      return X
```

2.4 Algorithme du pivot de Gauss et limites

On a alors toutes les briques en main pour écrire l'algorithme du pivot de Gauss :

```
1  def pivot_gauss(A,Y) :
2      B,W = forme_triangulaire(A,Y)
3      return remontee(B,W)
```

On a ensuite vu en TD quelles sont les limites de cet algorithme dus à la précision de stockage des flottants en Python :

- la réponse donnée par l'algorithme est une réponse approchée, des erreurs d'arrondis pouvant s'accumuler ;
- l'algorithme peut donner une solution alors qu'il n'y en a pas (bien que la matrice A ne soit pas inversible, une erreur d'arrondi la rend inversible) ;
- l'algorithme peut ne pas trouver de solution alors qu'il y en a une (bien que la matrice soit inversible, lors des calculs, un nombre différent de 0 mais trop petit pour la précision de la machine est changé en 0, rendant la matrice non inversible).

Toutefois, dans la très grande majorité des cas, cette méthode donne une réponse satisfaisante au problème posé.

3 Interpolation polynomiale de Lagrange

3.1 Présentation du problème

Si on imagine le suivi temporel d'une grandeur assisté par ordinateur, il est nécessaire de procéder à une numérisation du signal (pour d'évidentes raisons de capacités de stockage). Ainsi la fonction $s(t)$ qui est celle d'intérêt se

retrouve "résumée" en une suite de couples (s_i, t_i) des valeurs mesurées s_i aux différents instants t_i. Toutefois, lors d'une utilisation ultérieures, il peut être nécessaire de déterminer la valeur de s à un instant qui n'est pas un des t_i, et il va donc falloir fournir la meilleure approximation possible à partir des données (s_i, t_i), on parle d'*interpolation*.

On va donc supposer dans ce chapitre que nous avons à disposition deux tableaux S et T correspondant aux valeurs échantillonnées aux différents instants, et nous allons déterminer la meilleure approximation possible de la fonction de départ $s(t)$.

3.2 Interpolations par morceaux

Le principe de l'interpolation par morceaux consiste à traiter chaque morceau entre deux instants t_i comme indépendant des autres, et de recoller les solutions, d'où le nom d'interpolation par morceaux.

On supposera pour la suite que la liste T des instants est une liste déjà triée de n éléments équirépartis, et on souhaite créer une liste de valeurs F de la fonction interpolée en m points équirépartis.

3.2.1 Interpolation constante

Dans un premier temps, on va prendre pour valeur approchée $s(x) = s(t_i)$ où t_i est le plus grand élément de T inférieur ou égal à x.

On peut alors écrire le code suivant afin de créer la liste des valeurs de l'interpolation :

```
X = np.linspace(a,b,m)

def intervalle(x,liste) :
    j=0
    while liste[j] < x :
        j += 1
    return j-1

F_const = []
for i in range(m) :
    F_const.append(S[intervalle(X[i],T)])
```

3.2.2 Interpolation affine

Comme on peut s'en douter, la fonction obtenue par interpolation constante est une fonction constante par morceaux, donc il y a des discontinuités à chaque changement d'intervalle $[T_i, T_{i+1}]$. Or on sait que les signaux physiques sont souvent des fonctions continues du temps, on va donc vouloir supprimer ces discontinuités, ce qui est possible en prenant une interpolation qui soit une fonction affine par morceaux, et qui prend les valeurs mesurées à chaque instant de mesure ("on relie alors ces points par des droites").

On peut alors écrire les lignes de code suivantes :

```
def affine(a,fa,b,fb,x) :
    return (x-a)/(b-a)*fb + (b-x)/(b-a)*fa

F_aff = []
for i in range(m) :
    j = intervalle(X[i],T)
    F_aff.append(affine(T[j],S[j],T[j+1],S[j+1],X[i]))
```

3.2.3 Interpolation par splines

La fonction alors obtenue par interpolation affine est certes continue, mais non dérivable. On peut alors vouloir obtenir une interpolation plus régulière, surtout si les points de mesure stockés dans S ont été accompagnés d'une mesure de la dérivée aux mêmes instants stockée dans une liste D.

On va alors prendre pour interpolation sur $T_i, T_{i+1}]$ la fonction polynomiale de degré le plus bas qui prend les valeurs mesurées de la fonction et de sa dérivée. Puisqu'il y a 4 équations à vérifier (2 pour la fonction, 2 pour sa dérivée), il fait 4 coefficients, donc un polynôme de degré 3 : on parle de *splines cubiques*.

Les lignes suivantes permettent alors de faire l'interpolation souhaitée :

```
def coef_spline(a,fa,da,b,fb,db) :
    A = np.array([[a**3,a**2,a,1],[3*a**2,2*a,1,0],[b**3,b**2,b,1],[3*b**2,2*b,1,0]])
    Y = np.array([fa,da,fb,db])
    return np.linalg.solve(A,Y)

F_splines= []
for i in range(m) :
    j = intervalle(X[i],T)
    coefs = coef_spline(T[j],S[j],D[j],T[j+1],S[j+1],D[j+1])
    F_splines.append(coefs[0]*X[i]**3 + coefs[1]*X[i]**2+coefs[2]*X[i] + coefs[3])
```

On obtient donc une fonction de classe $\mathcal{C}^1$.

On peut alors tracer à la figure 3 les différentes interpolations et voir qu'elles s'approchent de plus en plus de la fonction recherchée (ici la fonction sinus) :

- lorsque l'on augmente n le nombre de points échantillonnés ;
- en augmentant la régularité de la méthode (de constant, à affine, à cubique par morceaux).

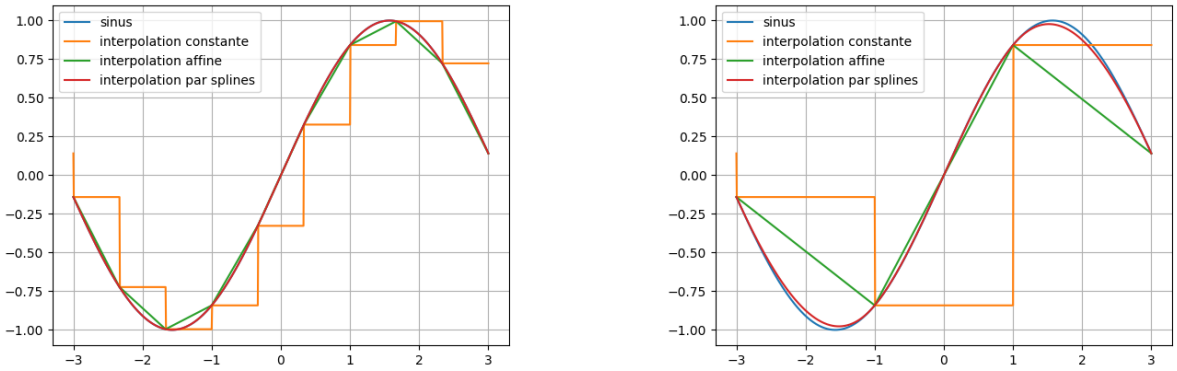

Figure 3 – Gauche : Différentes interpolations de la fonction sinus pour $n = 10$ points de mesure. Droite : Différentes interpolations de la fonction sinus pour $n = 4$ points de mesure.

3.3 Interpolation polynomiale de Lagrange

3.3.1 Présentation

Une autre méthode d'interpolation, qui a pour avantage de donner une fonction de classe $\mathcal{C}^\infty$, consiste à déterminer le polynôme de degré minimal qui prend la valeur s_i à chaque t_i. Il y a n équations à vérifier, donc il existe un unique polynôme de degré $n-1$ à trouver.

Une méthode pourrait consister à écrire le système de n équations à n inconnues puis la méthode du pivot de Gauss (ou numpy.linalg.solve) pour trouver les n coefficients. Il est toutefois plus astucieux d'écrire le polynôme recherché comme combinaison linéaire des polynômes de Lagrange associés aux couples (s_i, t_i) : le i-ème polynôme est alors celui qui vaut 1 en t_i et zéro pour tous les autres t_j.

On peut donc écrire pour cette distribution des t_i, $L_i(t) = \Pi_{j\neq i} \frac{(t-t_j)}{(t_i-t_j)}$.

L'interpolation polynomiale de Lagrange est alors la fonction :

$$f(t) = \sum_i s_i L_i(t)$$

.

On peut alors appliquer cette technique pour essayer d'interpoler la fonction sinus comme précédemment.

```
m =100
X_2 = np.linspace(-3,3,m)

def L_i(T,i,t) :
    k = len(T)
```

```
6       result = 1
7       for j in range(k) :
8           if j!=i :
9               result = result*(t-T[j])/(T[i]-T[j])
10      return result
11
12  def F_Lag(n) :
13      T_2 = np.linspace(-3,3,n)
14      S_2 = [np.sin(t) for t in T_2]
15      F =[]
16      for i in range(m) :
17          f_i = 0
18          for j in range(n) :
19              f_i += S_2[j]*L_i(T_2,j,X_2[i])
20          F.append(f_i)
21      return F
```

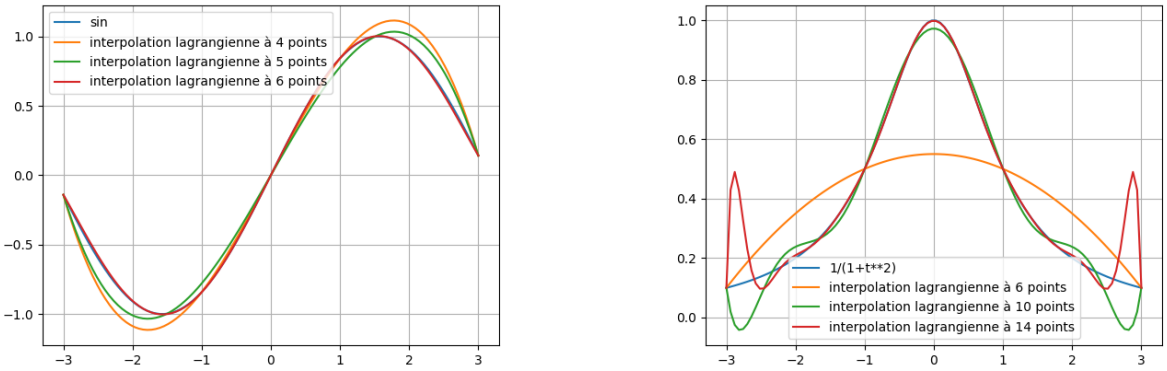

Figure 4 – Gauche : Interpolations lagrangiennes de la fonction sinus. Droite : Interpolations lagrangiennes de la fonction lorentzienne.

3.3.2 Phénomène de Runge et résolution

On voit alors fig 4 que pour seulement 6 points de mesure, la méthode est déjà très efficace. Toutefois, si la fonction à interpoler est une lorentzienne d'équation $s(t) = \frac{1}{1+t^2}$, un phénomène étrange apparait : si on augmente le nombre de points de mesure, la fonction est dans l'ensemble plus proche de la fonction recherchée, mais des écarts de plus en plus importants ont lieu aux extrémités. Il s'agit du *phénomène de Runge*.

Une méthode de résolution de ce problème consiste à ne pas prendre comme instants de mesures une liste équirépartie, mais plutôt les n racines du n-ème polynôme de Tchebychev à une transformation affine près.

Les polynômes de Tchebychev sont les polynômes obtenus lors de la linéarisation de $\cos(n\theta)$ comme $T_n(\cos\theta) = \cos(n\theta)$ (par exemple $T_2(X) = 2X^2 - 1$ puisque $\cos(2\theta) = 2\cos^2\theta - 1$).

On prend alors comme liste des T entre les instant a et b les n points :

$$x_k = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{2k+1}{2n}\pi\right), k \in [\![0, n-1]\!].$$

Pour la lorentzienne, on voit le résultat alors obtenu à la gauche de la figure 5. Les divergences aux bords du domaine d'intégration ont alors disparues, avec un accord qui s'améliore à mesure que le nombre de points d'interpolation augmente.

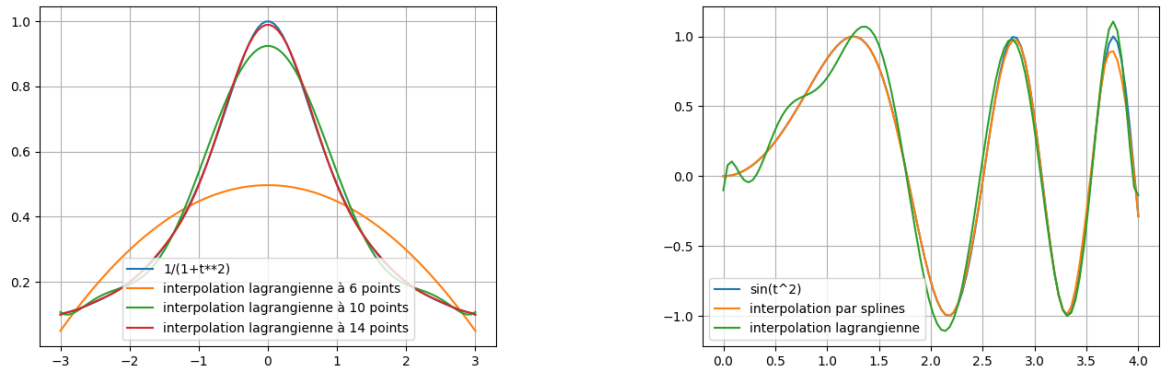

Figure 5 – Gauche : Interpolations lagrangiennes de la fonction lorentzienne en prenant les points de mesure aux abscisses de Tchebychev. Droite : Comparaison des interpolations lagrangienne et par splines de la fonction $\sin(t^2)$ pour 12 points de mesure.

3.3.3 Comparaison des interpolations par splines et lagrangienne

On peut maintenant comparer les deux méthodes d'interpolations les plus précises que l'on a vu, l'interpolation lagrangienne et l'interpolation par splines.

On va ici essayer d'interpoler la fonction $s(t) = \sin(t^2)$, avec 12 points de mesure, répartis selon les abscisses de Tchebychev entre 0 et 4.

On voit sur la droite de la figure 5 que l'interpolation par splines cubiques permet de mieux représenter la fonction en général, même en utilisant les abscisses de Tchebychev pour les points de mesure dans l'interpolation polynomiale de Lagrange (en gardant des points équirépartis pour l'interpolation par morceaux).

On en conclut qu'il n'y a pas de meilleure interpolation de manière générale, mais que selon la fonction des propriétés de la fonction que l'on cherche à interpoler, on privilégiera l'une ou l'autre des méthodes, en faisant attention au choix des instants de mesure de la fonction (par exemple, dans le dernier cas, l'interpolation par morceaux donnerait de meilleurs résultats avec des points plus proches vers la droite, là où la fonction varie le plus rapidement).