TP1 – Un exemple de programmation

11 octobre 2022

Le but de ces quelques pages est de proposer, au travers d'un exemple simple mais représentatif, une manière de programmer efficacement lors de l'épreuve de modélisation. L'idée générale est de commencer par des codes minimaux, qui sont enrichis par la suite. À chaque étape, on teste le programme, ce qui réduit de beaucoup les risques d'erreur dans le code.

1 Contexte

On va montrer comment mettre en œuvre efficacement la méthode d'Euler pour la résolution d'équations différentielles. Pour une équation du type

$$y'(t) = f\left(t, y(t)\right),$$

elle consiste – pour un pas h donné – à construire une suite d'approximations (y_n) de la solution y aux temps $t_n = nh$, par la formule

$$y_{n+1} = y_n + hf(t_n, y_n).$$

L'initialisation est fournie par la donnée de la condition initiale $y(0) = y_0$. Par la suite, on utilisera l'exemple modèle $y' = t - ty$ avec la condition initiale $y(0) = 2$, dont la solution exacte est donnée par $y(t) = 1 + \exp(-t^2/2)$.

La fonction Python correspondante est la suivante :

```
def f (t,y) : return (t-t.*y)
```

2 Programmation hiérarchique

Une manière efficace de programmer en temps limité consiste à débuter par une version très épurée de l'algorithme enrichie ensuite, ceci en testant pas-à-pas chaque modification. Le débogage est ainsi grandement facilité, et si le temps imparti pour la programmation est terminé, on est sûr de pouvoir présenter au moins une simulation convaincante !
À chaque étape importante, on peut par exemple copier et commenter la précédente version du code afin de pouvoir revenir en arrière. Dans cet énoncé, la même fonction `Euler` *aura trois versions.*

2.1 Le cœur du programme

L'itération de la méthode d'Euler s'écrit simplement `y=y+h*f(t,y)` si bien qu'une première version élémentaire de la méthode d'Euler est la suivante.

```
1  def Euler(y0,N,T) :
2      # Version 1
3      y = y0
4      t = 0
5      h = T/N
6      for i in range(0,N):
7          y = y + h*f(t,y)
8          t = t + h
9      return y
```

Pour la tester, on tape dans la fenêtre de commande

```
>>> Euler(2,5,2)
1.10692864
```

N.B. *on a préféré ici passer le nombre de points N comme argument plutôt que le pas h afin d'éviter l'utilisation d'une partie entière.*

2.2 Renvoi des arguments

Bien sûr, l'appel de la fonction précédente `Euler(y0,N,T)` ne fournit que l'approximation finale de $y(T)$, sans les valeurs intermédiaires... c'est un peu dommage si l'on souhaite tracer la solution. On y remédie en renvoyant la liste complète plutôt que la dernière valeur :

```
def Euler2(y0,N,T) :
    # Version 2
    y = y0
    liste_y = np.zeros(N+1)
    liste_y[0] = y0
    t = 0
    h = T/N
    for i in range(0,N) :
        y = y + h*f(t,y)
        t = t + h
        liste_y[i+1] = y
    return liste_y
```

On a choisi ici de renvoyer les valeurs dans un tableau numpy (np.array) ; on aurait pu utiliser une liste python et ajouter à chaque itération un nouveau terme avec la commande `liste_y.append(y)`. Cette méthode serait en particulier à privilégier si la construction de la liste se faisait avec une boucle `while` et si on ne connaissait pas initialement le nombre de termes à calculer.

Il peut être aussi commode que la fonction renvoie les temps successifs où sont effectuées les approximations :

```
def Euler3(y0,N,T) :
    # Version 3
    y = y0
    t = 0
    liste_y = np.zeros(N+1)
    liste_t = np.zeros(N+1)
    liste_y[0] = y0
    liste_t[0] = 0
    h = T/N
    for i in range(0,N):
        y = y + h*f(t,y)
        t = t + h
        liste_y[i+1] = y
        liste_t[i+1] = t
    return liste_y , liste_t
```

Il est évident que le vecteur `liste_t` retourné vaut aussi bien `np.linspace(0,T,N+1)`, mais dans le cas d'une méthode à pas variable, il sera plus facile d'adapter la construction proposée.

La possibilité de renvoyer à la fois les temps et les valeurs des approximations permet une utilisation très simple de la fonction pour tracer le graphe de la solution :

```
# Parametres
T = 2
N = 50
y0 = 2
# Calcul
[sol,tps] = Euler3(y0,N,T)
# Graphique
plt.plot (tps, sol)
plt.show ()
```

2.3 Bonnes pratiques

Si l'on souhaite comparer l'approximation obtenue avec la solution exacte, on peut définir une fonction :

```
def yex(t) :
    y = 1+ np.exp (-t**2/2)
    return y
```

Noter que la fonction peut s'appliquer à un tableau numpy. Elle permet un appel unique pour l'évaluation de la solution exacte sur la subdivision.

```
# Graphes solutions exacte et approchee
plt.plot(tps,sol)
plt.plot(tps,yex(tps))
plt.show ()
```

Insistons enfin sur le fait que la fonction `Euler` et ses variantes ne tracent aucune courbe. Il est préférable de dissocier le post-traitement graphique du calcul.

3 Raffinements

3.1 Habillage graphique

Lors de l'oral de modélisation, il est important, pour soi et pour le jury, que les graphes portent quelques indications qui permettent d'identifier les données : titre, légendes, axes, etc.

```
# Graphes avec habillage graphique
plt.plot(tps,sol,label="solution approchee")
plt.plot(tps,yex(tps), label="solution exacte")
plt.xlabel("Temps")
plt.ylabel("Solutions")
plt.title("Comparaison solution exacte et approchee par Euler explicite")
plt.legend()

plt.show ()
```

3.2 Erreur d'approximation – Ordre de convergence

Si l'on souhaite mettre en évidence la convergence d'ordre 1 de la méthode d'Euler, on doit effectuer des simulations pour différentes valeurs du pas h. Il suffit d'ajouter une boucle extérieure à notre programme principal.

```
# Avec une boucle sur le pas de temps (pour visualiser la convergence)
T = 2
y0 = 2
liste_N = np.arange(10,1000,100)
s = len (liste_N)
# Boucle sur N
liste_Err = np.zeros(s)
for i in range(s) :
    # Calcul
    [ sol , tps ]= Euler3 (y0 ,liste_N[i] ,T)
    # Evaluation de l'erreur
    erreur = np.linalg.norm ( sol - yex ( tps ) ,float('inf'))
    # Mise a jour du tableau d'erreurs
    liste_Err[i] = erreur
    i = i + 1

# Trace : echelle standard et logarithmique
plt.clf()
plt.subplot(1,2,1)
plt.plot ( liste_N , liste_Err,'o')
plt.title ( ' Erreur en fonction de N ')
plt.xlabel ( ' Nombre de points N ')
plt.ylabel ( ' Erreur uniforme ')
plt.show ()
plt.subplot(1,2,2)
plt.loglog (liste_N , liste_Err)
plt.grid()
```

```
plt.title ( ' Erreur en fonction de N (echelle log - log) ')
plt.xlabel ( ' Nombre de points N ')
plt.ylabel ( ' Erreur uniforme ')

A = np.polyfit ( np.log ( liste_N ) , np.log ( liste_Err ),1)
chaine =' Pente = '+ str (round(A[0],2))
plt.text (100 ,0.001 , chaine )
plt.show ()
```

Ce programme peut paraître long et assez complexe, mais il est le résultat de suites de programmes qui diffèrent les uns des autres de quelques lignes seulement. Comme tous les codes intermédiaires ont été testés et validés, il sera facile d'identifier d'éventuelles erreurs dans le code final.

3.3 Passage de fonction comme paramètre

On a vu plus haut qu'à chaque nouvelle équation différentielle, la fonction `f.m` devait être modifiée. Il peut être plus commode de définir une fonction par équation, et permettre un choix dans la fonction `Euler`. La fonction est alors passée en paramètre.

```
def Euler4(f,y0,N,T) :
    # Version 4, avec la fonction comme parametre
    y = y0;
    t = 0;
    liste_y = np.zeros(N+1)
    liste_t = np.zeros(N+1)
    liste_y[0] = y0
    liste_t[0] = 0
    h = T/N
    for i in range(0,N):
        y = y + h*f(t,y)
        t = t + h
        liste_y[i+1] = y
        liste_t[i+1] = t
    return liste_y , liste_t
```

L'appel s'effectue alors par

```
>>> Euler4(f,2,5,2)
(array([2.         , 2.         , 1.84       , 1.5712     , 1.297024   ,
       1.10692864]), array([0. , 0.4, 0.8, 1.2, 1.6, 2. ]))
```

3.4 Améliorer la portabilité de vos programmes

Il peut arriver de lancer son programme dans une autre session de python, disons sur l'ordinateur de la salle d'épreuve le jour J, et de constater avec désarroi un comportement inattendu (erreur ou résultats différents). La raison en est probablement qu'au fil de l'élaboration de vos programmes, certaines variables étaient restées en mémoire dans la session précédente sans être correctement chargées dans vos programmes. Pour se prémunir de cette difficulté, il est recommandé d'effacer les variables avant d'exécuter le code (en relançant la console).

4 Exercices

4.1 Système différentiel

Utiliser la méthode d'Euler programmée pour résoudre le système différentiel suivant :

$$\begin{cases} x'(t) &= x(t)\,(3 - y(t))\,, \\ y'(t) &= y(t)\,(-2 + 2x(t))\,, \end{cases}$$

On redéfinira la fonction `f`. On pourra tracer à partir des solutions approchées calculées les trajectoires en fonction du temps, ainsi que dans le plan de phase.

4.2 Méthode de Newton

La méthode de Newton permet d'approcher la solution d'une équation

$$F(x) = 0,$$

où $F : \mathbb{R}^d \to \mathbb{R}^d$ est différentiable. Partant d'une approximation initiale $x_0 \in \mathbb{R}^d$ de la solution, la suite (x_k) est construite par la récurrence [1]

$$x_{k+1} = x_k - [F'(x_k)]^{-1} F(x_k).$$

Programmer une fonction Python selon l'entête suivant,

```
def newton (F,DF,x0,Tol,MaxIter)
```

qui renvoie une approximation de la solution x. Les arguments d'entrée sont les suivants :

- `F` : nom de la fonction scilab définissant la fonction F,
- `DF` : idem pour la matrice jacobienne F',
- `x0` : vecteur initial $x^0 \in \mathbb{R}^d$,
- `Tol` : scalaire donnant la tolérance (critère d'arrêt sur l'incrément : $\|x^{k+1} - x^k\| <$ `Tol`),
- `MaxIter` : nombre maximal d'itérations (pour les situations de non-convergence).

N.B. *Pour gérer le critère d'arrêt [2], on utilisera une boucle* `while`*, selon le modèle suivant :*

```
err = Tol+1
iter = 0
while (err>Tol) and (iter<MaxIter) :
   iter = iter +1
   ...
```

Tester cette fonction sur l'exemple monodimensionnel simple $F(x) = e^x - e$, ainsi que sur le cas bi-dimensionnel

$$F(x_1, x_2) = \begin{bmatrix} x_1^2 + x_2^2 - 2 \\ x_1^2 - x_2^2 - 1 \end{bmatrix}.$$

4.3 Méthode d'Euler implicite

À l'aide de la fonction précédente, programmer une fonction scilab qui met en œuvre la méthode d'Euler implicite (ou *rétrograde*), dont l'itération s'écrit

$$y_{n+1} = y_n + hf(t_{n+1}, y_{n+1}).$$

N.B. *La fonction F qui définit le système à résoudre à chaque itération dépend de y_n et de h, on écrira une variante de la fonction* `newton` *précédente pour prendre en compte ces deux arguments supplémentaires.*

1. Remarque : pour les systèmes ($d > 1$), on évite de calculer la matrice inverse $[F'(x_k)]^{-1}$, mais on résout directement le système linéaire correspondant. Ainsi, on préfère écrire `DF(x)\F(x)` plutôt que `inv(DF(x))*F(x)`. Notez que matlab effectue automatiquement le remplacement, mais pas scilab.
2. Un autre exemple de critère d'arrêt possible est ($\|x^{k+1} - x^k\| > \|x^k - x^{k-1}\|$ ou $\|x^{k+1} - x^k\| = 0$). Ce critère permet d'atteindre la précision de la machine (donc sans utiliser de tolérance).

Comparer la méthode d'Euler explicite et la méthode d'Euler implicite sur le problème raide

$$y'(t) = -500(y(t) - \cos(t)),$$

ainsi que sur le système de Volterra-Lotka :

$$\left\{ \begin{array}{rcl} x'(t) & = & ax(t) - bx(t)y(t), \\ y'(t) & = & -cy(t) + dx(t)y(t). \end{array} \right.$$

On pourra aussi programmer la méthode de Crank-Nicolson :

$$y_{n+1} = y_n + \frac{h}{2}\Big[f(t_n, y_n) + f(t_{n+1}, y_{n+1})\Big].$$

et proposer une estimation numérique de la période des oscillations.

4.4 Une méthode de tir

On considère le problème aux limites suivant :

$$\left\{ \begin{array}{l} -u''(x) + a(x)u(x) = f(x), \\ u(0) = u(1) = 0, \end{array} \right.$$

où les fonctions $a \geq 0$ et f sont données. Pour résoudre ce problème, on introduit le problème de Cauchy dépendant du paramètre α :

$$\left\{ \begin{array}{l} -u''_\alpha(x) + a(x)u_\alpha(x) = f(x), \\ u_\alpha(0) = 0 \text{ et } u'_\alpha(0) = \alpha, \end{array} \right.$$

Ce dernier problème peut être résolu à l'aide d'une méthode d'intégration des équations différentielles ordinaires vues plus haut. Il suffit de trouver α pour que $u_\alpha(1) = 0$, ce qui n'est pas difficile puisque l'application $\varphi : \alpha \mapsto u_\alpha(1)$ est affine !

Pour aller plus loin, on pourra appliquer la même méthode au problème

$$\left\{ \begin{array}{l} -u''(x) + a(x)u^3(x) = f(x), \\ u(0) = u(1) = 0, \end{array} \right.$$

pour lequel l'application $\varphi : \alpha \mapsto u_\alpha(1)$ est non-linéaire. On pourra alors utiliser la méthode de Newton ou une variante pour résoudre le problème de tir.