

FICHE DE TRAVAUX PRATIQUES

Matrices et équations différentielles

But du TP

L'objectif de la séance de travaux pratiques est double : manipuler un logiciel libre de calcul scientifique et approfondir les méthodes numériques de résolution d'équations différentielles ordinaires en lien avec les matrices. Au travers d'une démarche de modélisation, il s'agit d'observer l'efficacité des différents algorithmes et leurs limitations dans l'étude numérique d'un problème concret.

Environnement informatique

La séance a lieu dans la salle libre-service linux ; des comptes « invités » ont été créés :

login : guest1 (ou bien guest2, guest3, etc.)

mot de passe : en201209

Au début de la séance, ouvrir un terminal (menu Applications>Accessoires), créer un dossier de travail et lancer scilab à l'aide des commandes

```
# mkdir TP_IPR
# cd TP_IPR
# scilab
```

Détail des activités

Il s'agit d'un programme à la carte, où chacun peut approfondir les thèmes de son choix.

- ◇ prise en main du logiciel Scilab,
- ◇ utilisation des méthodes de résolution numérique d'EDO,
- ◇ portrait du champ de vecteurs associé à l'EDO,
- ◇ comparaison des comportements qualitatifs des différents intégrateurs numériques (conservation d'intégrale première, etc.),
- ◇ comparaison des vitesses de convergence des différents schémas,
- ◇ étude de stabilité pour un problème raide.

PRISE EN MAIN DE Scilab

L'essentiel

Vecteurs et matrices

Les objets de base de Scilab sont les matrices : un vecteur est une matrice à une ligne ou une colonne ; un scalaire est une matrice 1×1 .

```
--> x=[1,4,-5];
--> y=[1;4;-5]
--> length(y)
--> norm(y,'inf')
--> z=1:2:10
--> t=linspace(0,10,100);
--> t(6)
--> A=zeros(3,3)
```

--> A(1,3)=3
--> B=[A,y]
--> C=[A;x]
--> size(B)
--> D=eye(3,3)+0.1*A;
--> D(3,:)
--> solution=D\y
--> D*solution

Fonctions

Voici comment définir ses propres fonctions dans Scilab : on crée le fichier suivant dans l'éditeur de texte et le sauvegarder (dans fct.sci)

```
function y=f(t)
    y=t.^3-t.*(1-t)-2;
endfunction
```

Il faut ensuite charger la fonction dans Scilab :

```
--> getf fct.sci
```

La fonction *f* peut alors être utilisée comme toute autre fonction prédéfinie.

```
--> f(9)
--> f([1,2,3])
```

Graphiques

Pour tracer un graphe dans Scilab, il faut créer les vecteurs des abscisses et des ordonnées.

```
--> t=linspace(0,1,100);
--> plot(t,sin(t))
--> plot(t,f(t))
--> clf
--> plot(t,cos(t),'r')
```

Pour aller plus loin

Programmation

Il est possible de rassembler dans un fichier texte une séquence de commandes Scilab. La commande `exec` permet alors de les exécuter.

Exemple : le fichier `script.sce` contient les instructions suivantes

```
[ N = 10; x = zeros(1,N);  
  for i = 1 : N  
    x(i) = log(i);  
  end  
  
  i = 1;  
  while(i < 5 & x(i) < 2)  
    i = i + 1;  
  end  
  [i,x(i)]
```

L'exécution dans Scilab s'effectue comme suit

```
--> exec script.sce
```

Régression linéaire

Parmi les nombreuses fonctions prédéfinies dans Scilab, on utilise fréquemment la commande `reglin` qui permet de calculer les coefficients de la droite des moindres carrés d'un nuage de points. Pour une description détaillée, voir l'aide :

```
--> help reglin
```

MÉTHODES NUMÉRIQUES DE RÉOLUTION D'EDO

L'essentiel

On considère l'équation différentielle avec condition initiale suivante :

$$\begin{cases} y'(t) = f(y(t)), & 0 \leq t \leq T, \\ y(0) = y_0, \end{cases}$$

où $f : \mathbb{R}^d \rightarrow \mathbb{R}^d$ est une fonction régulière et $y_0 \in \mathbb{R}^d$.

Pour l'approximation numérique, on note h le pas de la subdivision uniforme $(t_n)_{n=0}^N$ de l'intervalle $[0, T] : t_n = nh$ et y_n est une approximation de $y(t_n)$ pour $0 \leq n \leq N$.

On rappelle ci-dessous quelques méthodes classiques.

Euler explicite

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

Heun (ordre 2)

$$y_{n+1} = y_n + \frac{h}{2}f(y_n) + \frac{h}{2}f(y_n + hf(y_n))$$

Runge-Kutta d'ordre 4

$$\begin{cases} k_1^n & = f(y_n) \\ k_2^n & = f(y_n + \frac{h}{2}k_1^n) \\ k_3^n & = f(y_n + \frac{h}{2}k_2^n) \\ k_4^n & = f(y_n + hk_3^n) \\ y_{n+1} & = y_n + \frac{h}{6}(k_1^n + 2k_2^n + 2k_3^n + k_4^n) \end{cases}$$

Remarque : Pour un système non-autonome $y'(t) = f(t, y(t))$, on peut se ramener au cas autonome (f indépendant du temps), en considérant l'équation supplémentaire $t' = 1$. Par exemple, la méthode de Heun devient

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

Pour aller plus loin

Euler implicite

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

méthode du point milieu $y_{n+1} = y_n + hf(\frac{y_n + y_{n+1}}{2})$

Soulignons que les méthodes implicites nécessitent la résolution d'une équation (généralement non-linéaire) à chaque pas de temps. On peut utiliser la méthode de Newton initialisée avec l'itéré précédent. Précisément, y_n étant connu, il s'agit de trouver z , racine de l'équation $\varphi(z) = 0$ avec, pour la méthode d'Euler implicite,

$$\varphi(z) = z - y_n - hf(z).$$

La méthode de Newton construit la suite (z_k) à l'aide de la relation de récurrence

$$z_0 = y_n \quad \text{et} \quad \varphi'(z_k)(z_{k+1} - z_k) = -\varphi(z_k).$$

où $\varphi'(z) = I - hf'(z)$ désigne la matrice jacobienne. On arrête l'algorithme quand l'incrément $|z_{k+1} - z_k|$ est inférieur à un certain seuil (un nombre maximal d'itération est aussi imposé).

Remarque : Si le problème est non-raide (voir plus loin un exemple de problème raide), on peut aussi utiliser plus simplement une itération de point fixe,

$$z_0 = y_n \quad \text{et} \quad z_{k+1} = y_n + hf(z_k).$$

Mise en œuvre sur un exemple issu d'une modélisation

Lotka-Volterra. On rappelle le modèle proie prédateur de Lotka-Volterra, où $u(t)$ représente les prédateurs, et $v(t)$ représente les proies,

$$u'(t) = u(t)(v(t) - 2), \quad v'(t) = v(t)(1 - u(t)). \quad (1)$$

avec une condition initiale fixée, par exemple $u(0) = 3, v(0) = 4$.

On pourra comparer les diverses méthodes numériques introduites précédemment avec la *méthode d'Euler symplectique* définie par

$$u_{n+1} = u_n + h u_{n+1}(v_n - 2), \quad v_{n+1} = v_n + h v_n(1 - u_{n+1}).$$

qui est implicite pour u , mais explicite pour v (c'est une méthode partitionnée). Elle peut se réécrire explicitement comme $u_{n+1} = u_n / (1 - h(v_n - 2)), v_{n+1} = v_n + h v_n(1 - u_{n+1})$.

Pistes de travail

◇ PROGRAMMATION DE LA FONCTION DÉFINISSANT L'EDO

Le système différentiel s'écrit sous la forme $Y' = F(t, Y)$. Dans Scilab, il est représenté par la fonction vectorielle

```
function Yprime=F(t,Y)
```

◇ PORTRAIT DU CHAMP DE VECTEURS

On peut représenter le champ de vecteur associé au système à l'aide de la commande `fchamp`. Exemple :

```
--> deff("[Yprime] = F(t,Y)", "Yprime=[Y(1)*(Y(2)-2); Y(2)*(1-Y(1))]" )
--> xx=0:0.2:2;
--> yy=0:0.2:4;
--> fchamp(F,0,xx,yy)
```

Ceci permet aussi de visualiser les points d'équilibre du système différentiel.

◇ UTILISATION DE LA COMMANDE `ode` déjà implémentée dans Scilab

```
--> Y0=[3;4];
--> t=linspace(0,5,100);
--> Y=ode(Y0,0,t,F);
--> plot(Y(1,:),Y(2,:))
```

◇ PROGRAMMATION ET COMPARAISON DE MÉTHODES NUMÉRIQUES

On peut programmer, sur le modèle de la commande ode, les méthodes précédentes. Il est alors intéressant de comparer leur performance pour différentes valeurs du pas h .

◇ CONSERVATION DE L'INTÉGRALE PREMIÈRE ET INTÉGRATEURS SYMPLECTIQUES

Le système (1) possède une intégrale première, qui est conservée au cours du temps,

$$I(u, v) = \ln u - u + 2 \ln v - v.$$

On peut vérifier que les méthodes d'Euler symplectique et du point milieu conservent bien cette intégrale première si on intègre sur de très nombreuses périodes (sans biais linéaire), contrairement aux autres méthodes considérées.

◇ ÉTUDE DU COMPORTEMENT QUALITATIF DES MÉTHODES NUMÉRIQUES

Si on linéarise le problème (1) au voisinage de l'équilibre (1, 2), on obtient le système

$$u'(t) = v(t), \quad v'(t) = -2u(t).$$

Chacune des méthodes d'Euler explicite, implicite et symplectique appliquée à ce système linéarisé conduit à une récurrence du type $y_{n+1} = Ay_n$, avec $y_n = (u_n, v_n)^T$, et A est une matrice de taille 2×2 . En étudiant les modules des valeurs propres de A (commande spec), on peut en déduire le comportement de y_n lorsque $n \rightarrow +\infty$.

◇ COMPARAISON DES VITESSES DE CONVERGENCE

Pour comparer les ordres de convergence des différentes méthodes, on se place dans le cas modèle $y'(t) = -y(t)$ avec $y(0) = 1$ pour lequel la solution exacte est connue.

En faisant varier le pas h , on peut obtenir les courbes d'erreur pour chaque méthode :

$$\mathcal{E}(h) = \max_{n=0, \dots, N} |y(t_n) - y_n|.$$

Un exemple de problème raide

Le modèle de Volterra-Lotka (1) étudié précédemment n'est pas un problème raide : il n'y a pas de contrainte particulière sur la taille du pas de temps choisi pour obtenir la stabilité numérique des schémas explicites.

On considère maintenant le problème d'équation différentiel raide suivant (Curtis et Hirschfelder, 1952)

$$y'(t) = \lambda(y(t) - \cos t), \quad y(0) = y_0, \tag{2}$$

avec $\lambda < 0$ grand, par exemple $\lambda = -10^7$.

Si on linéarise ce problème, on obtient l'équation linéaire test (équation de Dahlquist)

$$y'(t) = \lambda y(t), \quad y(0) = y_0.$$

on voit que la méthode d'Euler explicite, qui donne $y_n = (1 + h\lambda)^n y_0$, reste bornée lorsque $n \rightarrow +\infty$ si et seulement si $h \leq 2/|\lambda| = 2 \cdot 10^{-7}$. Pour éviter cette restriction de stabilité très contraignante sur le pas de temps h , on peut alors utiliser alors une méthode numérique implicite pour résoudre (2), comme la méthode d'Euler implicite.