T.C.D'I. — PC 1/ PC* 1/MP 1/MP* 2

ÉQUATIONS NON LINÉAIRES

§1. Méthode de Newton en dimension un

Exercice 1 — Approximation du nombre dérivé. Soit $f : \mathbf{R} \to \mathbf{R}$ une fonction deux fois dérivable.

1) En utilisant la formule de Taylor-Young, montrer que pour tout nombre h assez petit on a

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2).$$

2) Sachant que les nombres à virgule utilisent une quinzaine de chiffres significatifs, quelle valeur de h semble judicieuse pour l'approximation précédente ?

3) En déduire un programme `Dérivée(`f`,` x`)` qui renvoie une valeur approchée de $f'(x)$.

Exercice 2 — Méthode de Newton scalaire. Soit f une fonction dérivable. On rappelle que l'itération de Newton est

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

1) On suppose que la suite $(x_n)_{n\geqslant 0}$ vérifie la formule de récurrence ci-dessus, qu'elle converge vers un nombre $\ell \in \mathbf{R}$, et que f est de classe $\mathscr{C}^1$ en ℓ. Montrer que $f(\ell) = 0$.

2) Écrire un programme `Newton(`f`,` x_0`)` qui applique l'itération de Newton à partir de x_0, et renvoie x_{n+1} avec n le plus petit indice tel que $|x_{n+1} - x_n| \leqslant 10^{-10}$.

Exercice 3 — Racines d'un polynôme. On admet que le résultat de l'exercice précédent reste vrai si la suite $(x_n)_{n\geqslant 0}$ est à valeurs complexes. On ne sait pas en général ce qu'est $f'(z)$ pour une $f : \mathbf{C} \to \mathbf{C}$ quelconque, mais on sait au moins ce que c'est dans deux cas : si f est polynomiale, ou si f est développable en série entière.

1) Expliquer l'intérêt du programme ci-dessous.

```
def EstDedans(L, ℓ) :
    for x in L :
        if abs(x - ℓ) <= 1e-5 :
            return True
    return False
```

2) Écrire un programme `CpxAléa()` qui renvoie un nombre complexe z aléatoire tel que $-10 \leqslant \text{ré}(z) < 10$ et $-10 \leqslant \text{im}(z) < 10$. On pourra utiliser `random` du module `random`.

3) En déduire un programme `Racines(P)` qui étant donné un polynôme complexe P renvoie la liste de ses racines.

```
>>> Racines(P)
[(1.7383040313216025  - 5.707547316230016e-28j),
 (0.37907197862630376 + 1.1772717406608655j),
 (-1.248223994287105  - 1.3515326138090487j),
 (0.37907197862630376 - 1.1772717406608653j),
 (-1.248223994287105  + 1.3515326138090487j)]
```

§2. Méthode de Newton en dimension quelconque

Dans cette partie, on représente les éléments de $\mathbf{R}^d$ par des matrices de taille $d \times 1$. On donne

```
from numpy import matrix
def Zéros(m, n) :
    return matrix([[0.0 for j in range(n)] for i in range(m)])
```

qui renvoie la matrice nulle de taille $m \times n$, et on rappelle qu'on peut accéder (pour le lire ou le modifier) au coefficient en position (i, j) de la matrice `M` en écrivant `M[i, j]`, l'indiçage démarrant à zéro. Les opérations arithmétiques habituelles (y compris la multiplication par un scalaire) sont utilisables, en particulier si `M` est carrée et inversible alors `M ** (-1)` donne son inverse.

Enfin, on récupère la taille d'une matrice en écrivant `(m, n) = M.shape`.

Exercice 4 — Norme d'un vecteur. Écrire le programme `Norme(X)` qui étant donnée une matrice représentant un élément de $\mathrm{X} \in \mathbf{R}^d$ calcule et renvoie

$$\|\mathrm{X}\| = \max |x_i|.$$

Exercice 5 — Matrice jacobienne. Soit $\mathrm{F} : \mathbf{R}^d \to \mathbf{R}^d$, qu'on suppose de classe $\mathscr{C}^1$. On se propose de calculer sa *matrice jacobienne*

$$\mathrm{F}'(\mathrm{X}) = (\partial_j \mathrm{F}_i(\mathrm{X}))_{0 \leqslant i,j \leqslant d-1} \in \mathscr{M}_d(\mathbf{R}),$$

où $\mathrm{F}_i(\mathrm{X})$ est la i-ème composante de $\mathrm{F}(\mathrm{X})$ et où ∂_j désigne la dérivée par rapport à la j-ème variable.

1) Écrire un programme `Petit(d, i, h)` qui construit et renvoie la matrice-colonne de taille $d \times 1$, dont tous les coefficients sont nuls, sauf celui en position i qui vaut h.

2) S'en servir pour écrire un programme `Partielles(F, j, X)` qui calcule et renvoie la matrice-colonne $(\partial_j \mathrm{F}_i(\mathrm{X}))_i$.

3) En déduire le programme `Jacobienne(F, X)` qui construit et renvoie la matrice ci-dessus.

Exercice 6 — Méthode de Newton vectorielle. Soient $\mathrm{F} : \mathbf{R}^d \to \mathbf{R}^d$ une fonction de classe $\mathscr{C}^1$, qu'on cherche à annuler, et X_0 un vecteur quelconque. Par analogie avec la version scalaire, on considère la formule d'itération

$$\mathrm{X}_{n+1} = \mathrm{X}_n - \mathrm{F}'(\mathrm{X}_n)^{-1}\mathrm{F}(\mathrm{X}_n).$$

Écrire le programme `NewtonVectoriel(F, X0)` qui applique cette formule d'itération et renvoie le vecteur X_{n+1}, pour le plus petit indice n tel que $\|\mathrm{X}_{n+1} - \mathrm{X}_n\| \leqslant 10^{-10}$.

§3. Différences finies

On se propose de résoudre (de manière approchée) le problème aux dérivées partielles

$$\begin{cases} \dfrac{\partial u}{\partial t} = \kappa \dfrac{\partial^2 u}{\partial x^2} + \lambda(t,x)\sin(u) & \text{pour } (t,x) \in \left]0\,;\, t_{\max}\right] \times \left]0\,;\, 10\right[, \\ u(t,0) = a(t) \text{ et } u(t,10) = b(t) & \text{pour } t \in \left]0\,;\, t_{\max}\right], \\ u(0,x) = u^0(x) & \text{pour } x \in [0\,;\, 10], \end{cases}$$

dans lequel $u : [0\,;\, t_{\max}] \times [0\,;\, 10] \to \mathbf{R}$ est la fonction inconnue, des deux variables t (le temps) et x (la position), avec κ est une constante donnée, et où $\lambda : [0\,;\, t_{\max}] \times [0\,;\, 10] \to \mathbf{R}$, $a, b : [0\,;\, t_{\max}] \to \mathbf{R}$ et $u^0 : [0\,;\, 10] \to \mathbf{R}$ sont des fonctions données. Toutes ces données seront des variables *globales*.

Fixons deux entiers $\mathrm{N} \geqslant 2$ et $\mathrm{K} \geqslant 1$, posons $h = 10/\mathrm{N}$ et $\eta = t_{\max}/\mathrm{K}$, et discrétisons l'espace et le temps en introduisant, pour $0 \leqslant i \leqslant \mathrm{N}$ et $0 \leqslant k \leqslant \mathrm{K}$, les quantités

$$x_i = i \times h \quad \text{et} \quad t_k = k \times \eta.$$

On va chercher des valeurs approchées de la fonction inconnue en chacun de ces lieux et instants, ce qui, en posant

$$u_i^k = u(t_k, x_i),$$

donne $(\mathrm{N}+1) \times (\mathrm{K}+1)$ inconnues.

Pour le temps, on va procéder de proche en proche : pour un certain $k \geqslant 1$ fixé, on suppose avoir calculé *toutes* les u_i^{k-1}, et on cherche les u_i^k. Cette façon de faire est suggérée par le fait qu'on connaît, avec la fonction u^0, tous les u_i^0.

Exercice 7 — Discrétisation des dérivées

1) Justifier, si u est suffisamment régulière, que $\dfrac{\partial u}{\partial t}(t_k, x_i) = \dfrac{u_i^k - u_i^{k-1}}{\eta} + O(\eta)$.

2) De même justifier que $\dfrac{\partial^2 u}{\partial x^2}(t_k, x_i) = \dfrac{u_{i-1}^k + u_{i+1}^k - 2u_i^k}{h^2} + O(h^2)$.

Exercice 8 — Un grand système d'équations. On fixe $k \geqslant 1$. Soit $\mathrm{F} : \mathbf{R}^{\mathrm{N}+1} \to \mathbf{R}^{\mathrm{N}+1}$ la fonction définie par

$$\mathrm{F}\begin{pmatrix} u_0 \\ u_1 \\ \vdots \\ u_i \\ \vdots \\ u_{\mathrm{N}} \end{pmatrix} = \begin{pmatrix} u_0 - a(t_k) \\ \dfrac{u_1 - u_1^{k-1}}{\eta} - \kappa \times \dfrac{u_0 + u_2 - 2u_1}{h^2} - \lambda(t_k, x_1)\sin(u_1) \\ \vdots \\ \dfrac{u_i - u_i^{k-1}}{\eta} - \kappa \times \dfrac{u_{i-1} + u_{i+1} - 2u_i}{h^2} - \lambda(t_k, x_i)\sin(u_i) \\ \vdots \\ u_{\mathrm{N}} - b(t_k) \end{pmatrix}.$$

Si l'on remplace toutes les dérivées par leur expressions approchées, le problème aux dérivées partielles ci-dessus revient donc à annuler F.

Écrire le programme `FabriquerF(k, U)` qui étant donnés k et la matrice-colonne $(u_i^{k-1})_{0 \leqslant i \leqslant \mathrm{N}}$ construit et renvoie la fonction F ci-dessus.

Exercice 9 — Résolution du problème

1) Écrire le programme qui fabrique la matrice colonne $(u_i^0)_{0 \leqslant i \leqslant \mathrm{N}}$.

2) Parce qu'on en aura besoin sous cette forme plus tard : écrire un programme qui étant donnée une matrice-colonne renvoie la liste de ses coefficients.

3) En utilisant tout ce qui précède, écrire le programme qui construit et renvoie la liste de listes `Les_U`, de longueur $\mathrm{K}+1$, telle que `Les_U[k]` est la liste de longueur $\mathrm{N}+1$ formée des u_i^k.

§4. Dessin animé

Sans exercice et pour les intéressés, voici deux manières « d'afficher » les résultats : la manière simple (mais lente) et la manière compliquée (mais rapide).

Dans les deux cas on supposera avoir précalculé les solutions : d'abord on entre toutes les données...

```
from numpy import sin
```

```
def a(t) :
    return 0
def b(t) :
    return 1
```

```
κ = 1
def λ(t, x) :
    return 1
```

```
N = 50
h = 10 / N
```

```
t_max = 10
K = 100
η = t_max / K
```

```
def u⁰(x) :
    return x / 10
```

...puis on applique le programme principal.

```
Les_U = Résoudre()
```

4.1 La manière simple

Nous utiliserons ici les fonctions suivantes.

```
from matplotlib.pyplot import clf, grid, pause, plot, show
```

L'idée est assez simple : on affiche la courbe pour une certaine valeur de k, on fait une temporisation avec `pause()`, on efface tout avec `clf()`, puis on recommence.

```
BLEU = (0, 0, 1)
def Animer() :
    abscisses = [i * h for i in range(N+1)]
    for k in range(K + 1) :
        ordonnées = Les_U[k]
        plot(abscisses, ordonnées, linewidth = 2, color = BLEU)
        pause(0.1)
        clf()
```

```
>>> Animer()
```

Le paramètre de la fonction `pause` est la durée de la temporisation, en secondes. Le problème c'est qu'à l'usage, on constate que les choses sont bien plus lentes que dix courbes par seconde.

4.2 La manière compliquée

On va essayer de comprendre pourquoi c'est lent. Il y a trois choses qui interviennent : les calculs, la construction des objets graphiques, et leur affichage. On a fait en sorte d'éliminer les calculs du problème, en les faisant à part et en stockant tous les résultats dans la variable `Les_U`.

On le constatera à l'usage, ce n'est pas l'affichage qui prend du temps. On va donc chercher à améliorer la phase de construction des objets. Déjà, il faut voir que dans un graphique, il n'y a pas que la courbe : il y a aussi le graphique en lui-même (la page blanche sur laquelle tout est dessiné), le repère (avec les choix d'échelles pour chaque axe).

Pour gagner du temps, on va donc seulement reconstruire la courbe. Et mieux que cela, on ne va pas la supprimer et en construire une nouvelle : on va modifier une seule et même courbe. Parce qu'en fait ce qui est long, c'est (je schématise un peu) de créer la ligne brisée en mémoire. Modifier les coordonnées des sommets, une fois que l'objet existe, est bien plus rapide ; et c'est ce que nous allons faire. Nous utiliserons seulement deux fonctions

```
from matplotlib.pyplot import figure, show
```

et on va voir que toutes les autres (`plot` ou `grid` par exemple) ne seront plus que des méthodes associées à un objet de classe `figure`, créé une fois pour toutes.

Et quitte à faire des choses compliquées, faisons-les bien : on va calculer un fenêtrage fixe, en cherchant (verticalement) la plus petite et la plus grande valeur à afficher, et en ajoutant des marges de 5 %.

```
def CalculerFenêtre(Les_U) :
    ymin = min([min(U) for U in Les_U])
    ymax = max([max(U) for U in Les_U])
    m = (ymax - ymin) * 0.05
    return (ymin - m, ymax + m)
(ymin, ymax) = CalculerFenêtre(Les_U)
```

Ensuite on créé les objets : une figure, qui contient tout, et sur cette figure, un graphique. Dont on fixe immédiatement les échelles.

```
def CréerObjetsGraphiques() :
    Figure     = figure()
    Graphique  = Figure.add_subplot(1, 1, 1)
    Graphique.set_xlim([0, 13])
    Graphique.set_ylim([ymin, ymax])
    Graphique.grid(True)
    return (Figure, Graphique)
```

Ensuite on affiche la première courbe, celle qui correspond à $t = 0$. Et on l'a dit, on en profite pour faire les choses bien : on va ajouter à côté un petit texte qui indique la valeur de t. Ci-dessous, la fonction `draw()` est celle qui affiche les objets. Pour l'instant, pas d'amélioration de performance.

Une subtilité : la fonction `plot` renvoie non pas un objet mais un singulet contenant cet objet (j'ignore pourquoi).

```
def PremierAffichage(Figure, Graphique) :
    abscisses = [i * h for i in range(N + 1)]
    ordonnées = Les_U[0]
    (Courbe,) = Graphique.plot(abscisses, ordonnées, linewidth = 2,
                               color = BLEU)
    Texte = Graphique.text(11, (ymin + ymax) / 2, "t = " + str(0.00))
    show(block = False)
    Figure.canvas.draw()
    return (Courbe, Texte)
```

On en vient à la partie efficace : pour mettre à jour le dessin, plutôt que de tout détruire (avec `clf()`) puis tout recréer, on va seulement modifier les objets existants (avec `set_data` pour la courbe, et `set_text` pour le texte), indiquer qu'il faudra les redessiner (avec `draw_artist`) et enfin redessiner ce qui a changé (avec `draw()`) et mettre à jour l'affichage (avec `flush_events`). C'est le strict minimum, et en testant on pourra constater le temps gagné.

```
def MettreÀJour(k, Figure, Graphique, Courbe, Texte) :
    abscisses = [i * h for i in range(N + 1)]
    ordonnées = Les_U[k]
    Courbe.set_data(abscisses, ordonnées)
    Graphique.draw_artist(Courbe)
    Texte.set_text("t = " + str(int(100 * k * eta) / 100))
    Graphique.draw_artist(Texte)
    Figure.canvas.draw()
    Figure.canvas.flush_events()
```

Pour réaliser l'animation, on n'a plus qu'à mettre tout ceci bout à bout.

```
def Animer() :
    (Figure, Graphique) = CréerObjetsGraphiques()
```

```
    (Courbe, Texte) = PremierAffichage(Figure, Graphique)
    for k in range(1, K + 1) :
        MettreÀJour(k, Figure, Graphique, Courbe, Texte)
```

Et cette fois-ci, on arrive à avoir environ dix fois plus d'images par seconde. En bricolant encore plus, on pourrait encore afficher entre cinq et dix fois plus d'images par seconde.

```
Animer()
```