MT94 - Cahier d'intégration

Introduction aux Mathématiques Appliquées
Printemps 2017

Alexandre Brasseur

Table des matières

Avant-propos

Ce cahier d'intégration a pour vocation d'exposer tout ce que j'ai pu apprendre dans l'UV MT94. Cette introduction aux mathématiques appliquées est le juste mélange entre les théories mathématiques vues en cours, leurs applications en Travaux Pratiques avec le logiciel de calcul numérique Scilab et le compte-rendu de ces connaissances dans ce rapport en LaTeX. On apprend donc beaucoup au cours du semestre et ce qui fait que MT94 est une très bonne UV.

Premièrement, elle permet d'appliquer toutes les connaissances mathématiques apprises depuis MT90 jusqu'à SY01. On comprend enfin comment donner un sens à toutes ses formules et comment elles servent à résoudre des problèmes. On a un aperçu du côté pratique des mathématiques. C'est un peu comme se servir d'un tournevis la première fois après l'avoir étudié pendant des années, le tournevis est cependant très complexe. On s'approche ainsi du travail de l'ingénieur et cela est plaisant.

Deuxièmement, on n'aperçoit certes qu'une partie des mathématiques appliquées, mais on a envie de continuer. Explorer entièrement tous les outils, les méthodes et leurs applications vus dans cette matière prendrait je pense plus d'un semestre. Cependant une fois lancée, cette envie de trouver de nouvelles applications, d'optimiser des algorithmes et de découvrir une nouvelle facette des mathématiques, ne s'arrête pas. Ce rapport n'est pas complet et ne le sera probablement jamais, tout comme les sciences mathématiques. Je n'ai pas la prétention d'y rapporter toutes les connaissances sur le sujet mais de continuer à y intégrer les miennes.

Troisièmement, cette UV est bien enseignée. Stéphane Mottelet et Djalil Kateb sont passionnés de mathématiques et cela se voit. Je les remercie pour avoir renforcé en moi cet intérêt pour cette science.

Enfin, l'ensemble des codes Scilab que j'ai créé sont disponibles à cette adresse : `https://abrasseu@gitlab.utc.fr/abrasseu/MT94.git`. Bonne lecture!

CHAPITRE I

Problèmes non-linéaires

Dans ce chapitre, l'objectif est de résoudre des équations, d'annuler des fonctions, d'abord dans $\mathbb{R}$, puis dans $\mathbb{R}^n$, c'est-à-dire trouver x^* tel que $f(x^*) = 0$. De nombreux problèmes peuvent être modélisés par des équations (cas de f à valeurs dans $\mathbb{R}$) voire par des systèmes dynamiques (cas de f à valeurs dans $\mathbb{R}^n$). En réalité, ces modèles sont bien souvent non-linéaires et les résoudre de manière analytique est donc impossible. On a donc recours à des méthodes numériques qui, par l'utilisation d'algorithmes spécifiques, permettent d'obtenir une solution approchée correcte.

Dans cette partie nous traitons les problèmes de résolution. Nous verrons les problèmes d'optimisation dans la seconde partie des problèmes non-linéaires.

I) Méthodes numériques de résolution dans $\mathbb{R}$

1) Ordre de convergence théorique

L'ordre de convergence d'une méthode de résolution numérique correspond à la vitesse à laquelle elle converge vers la solution x^*.

DÉFINITION I.1 :
Une méthode est d'ordre de convergence α si et seulement si $\exists \alpha$ tel que :

$$\frac{|x_{k+1} - x^*|}{|x_k - x^*|^\alpha} = C \qquad \text{avec } C \in \mathbb{R} \tag{I.1}$$

Nous utiliserons cette définition pour trouver l'ordre des méthodes présentées dans ce chapitre, cependant nous voudrons représenter graphiquement ces ordres. On a donc :

$$\begin{aligned} \text{(I.1)} &\Longrightarrow |x_{k+1} - x^*| = C \times |x_k - x^*|^\alpha \\ &\Longrightarrow \ln|x_{k+1} - x^*| = \ln C + \alpha \times \ln|x_k - x^*| \\ &\Longrightarrow \ln|x_{k+1} - x^*| = F(|x_k - x^*|) \end{aligned}$$

On peut donc représenter la convergence d'une méthode par une droite F définie comme :

$$F(X) = \ln C + \alpha X \qquad \text{avec l'écart } X = |x_k - x^*| \tag{I.2}$$

Ainsi une méthode sera plus rapide et donc plus efficace qu'une autre si son coefficient α est supérieur ; et secondairement on peut comparer les constantes C.

2) Méthode de la dichotomie

a) Idée

La dichotomie, aussi appelée bissection, provient du grec *dikhotomia* qui représente une division en deux parties. Cette méthode consiste à se rapprocher de la solution par divisions successives par deux de l'intervalle de départ contenant la solution.

On choisit d'abord d'un intervalle $[a, b]$ contenant la solution x^*. Le choix de l'intervalle de départ est important car si la solution n'y est pas comprise, l'algorithme ne converge pas et s'il est trop grand, la convergence sera plus longue.

Pour tout entier n, on définit deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ correspondant aux bornes de l'intervalle à l'itération n avec $a_0 = a$ et $b_0 = b$; ainsi que la suite (x_n) telle que :

$$x_n = \frac{a_n + b_n}{2}$$

A chaque itération, on compare le signe de $f(a_n)$ à celui de $f(x_n)$ et on réduit l'intervalle de moitié en affectant x_n à a_{n+1} ou b_{n+1}, ainsi $[a_{n+1}, b_{n+1}]$ se resserre sur x^* et x_{n+1} se rapproche de la solution.

Pour que cette méthode soit applicable, il suffit que f soit continue pour pouvoir trouver des valeurs intermédiaires comme cela est garanti par le théorème des valeurs intermédiaires.

b) Convergence

Par construction, on divise l'intervalle par deux à chaque itération donc on a :

$$(b_n - a_n) = \left(\frac{1}{2}\right)^n (b_0 - a_0)$$

$$|x_n - x^*| \leq \frac{1}{2}(b_n - a_n) = (\frac{1}{2})^{n+1}(b_0 - a_0) \xrightarrow[n\to\infty]{} 0$$

On remarque bien que l'erreur est réduite de manière quasi-linéaire.

3) Méthode du point fixe

a) Idée

On cherche $g : \mathbb{R} \to \mathbb{R}$ telle que $f(x) = 0 \iff g(x) = x$. Ainsi on ramène le problème d'annulation de la fonction f a la recherche d'une point fixe pour la fonction g.

Exemple : $f(x) = x^2 - 2$

$$x^2 - 2 = 0 \iff g(x) = x \iff g(x) = \frac{x+2}{x+1}$$

De plus, il faut que g soit dérivable et que sa dérivée au point fixe ne dépasse pas 1 en norme. Alors la convergence de cette méthode est garantie par le théorème suivant :

THÉORÈME I.1 :
Soit $g : \mathbb{R} \to \mathbb{R}$ dérivable et admettant un point fixe x^* telle que $|g'(x^*)| < 1$ alors $\exists [a;b]$ tel que $x^* \in [a;b]$ et la suite $(x_n)_{n\geq\mathbb{N}}$ définie comme suit, converge vers x^*

$$\begin{cases} x_{n+1} = g(x_n) \\ x_0 \in [a;b] \end{cases}$$

En revanche il y a un problème : pour que cette méthode convergence assez rapidement x_0 doit être proche de x^*, il faut donc bien le choisir.

b) Convergence

Si $|g'(x^*)| < 1$ alors

$$\frac{|x_{n+1} - x^*|}{|x_n - x^*|} < |g'(x^*)|$$

Et donc :

$$|x_{n+1} - x^*| < |g'(x^*)||x_n - x^*|$$

Ainsi la méthode du point fixe est d'ordre 1. Par récurrence :

$$|x_{n+1} - x^*| < |g'(x^*)|^n |x_0 - x^*| \xrightarrow[n\to\infty]{} 0$$

Elle converge bien vers x^* car $|g'(x^*)| < 1$. De plus la convergence de cette méthode est d'ordre 1.

4) Méthode de Newton

a) Idée

Cette méthode est plus puissante mais il faut que f soit deux fois continûment dérivable (*i.e.* $f \in \mathscr{C}^2$). Le but est d'approcher la solution grâce à la tangente de la courbe. On fait un développement de Taylor-Young de f en x. Soit $x_0 \in \mathbb{R}$:

$$f(x) = \underbrace{f(x_0) + f'(x_0)(x - x_0)}_{T_{x_0}} + \frac{(x - x_0)^2}{2} f''(x_0 + \theta(x - x_0))$$

On cherche à rapprocher x_0 de x dont l'image est la solution désirée, cette aproximation est réalisée quand la différence entre $f(x)$ et $f(x_0)$ est négligeable, c'est-à-dire quand la tangente T_{x_0} s'annule. On définit x_1 tel que $T_{x_0}(x_1) = 0$ donc si $f'(x_0) \neq 0$ on a :

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

Par récurrence on a alors :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{I.3}$$

L'algorithme de la méthode est alors le suivant :

Algorithme I.1 : Méthode de Newton

Data : $x_0 \in \mathbb{R}$ donné
Result : $x^* \in \mathbb{R}$ la solution approchée à ϵ près
while $|f(x_n)| > \epsilon$ *and* $f'(x_n) \neq 0$ **do**
  $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$;
end while

Nous avons :

$$\text{(I.3)} \iff x_{n+1} = g(x_n) \text{ avec } g(x) = x - \frac{f(x)}{f'(x)}$$

$$g'(x) = 1 - \frac{\left(f'(x)\right)^2 - f(x)f''(x)}{\left(f'(x)\right)^2}$$

$$\implies g'(x^*) = 1 - \frac{\left(f'(x^*)\right)^2}{\left(f'(x^*)\right)^2} = 0$$

$|g'(x^*)| < 1$ donc d'après le théorème (I.1) la méthode converge bien.

b) Convergence

On suppose $f \in \mathscr{C}^3$. On a :

$$x_{n+1} - x^* = g(x_n) - g(^*)$$

On effectue un développement de Taylor-Young :

$$g(x_n) = g(x^*) + (x_n - x^*)g'(x^*) + \frac{(x_n - x^*)}{2} g''(\xi)$$

Avec ξ compris entre x_n et x^*.

$$|x_{n+1} - x^*| \leq C|x_n - x^*|^2$$

Avec $C = \frac{1}{2}\max g''(\xi)$. La méthode de Newton est donc d'ordre quadratique, ce qui la rend donc plus efficace. Concrètement cela signifie que le nombre de décimales significatives double à chaque itération.

5) Méthode de la sécante

a) Idée

Cette méthode ressemble à celle de Newton car elle la reprend en approchant $f'(x)$ par le taux de variation. Ici la seule hypothèse est que

Algorithme I.2 : Méthode de la sécante

Data : x_0 et $x_1 \in \mathbb{R}$ donnés
Result : $x^* \in \mathbb{R}$ la solution approchée à ϵ près
while $(|f(x_n)| > \epsilon)$ **do**
  $x_{n+1} = x_n - \frac{f(x_n)}{f(x_n)-f(x_{n-1})}(x_n - x_{n-1})$;
end while

b) Convergence

La convergence de cette méthode est d'ordre le nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$.

6) Comparaison des méthodes

On compare expérimentalement les méthodes en cherchant à annuler $f(x) = x^2 - 2$. La solution recherchée ici est seulement $\sqrt{2}$. En effet, les différentes méthodes ont leurs paramètres centrés sur la solution poisitive, ils convergeront donc vers celle-ci.

On a les codes Scilab suivants pour les différentes méthodes :

```
function [x,i] = Dichotomie(f,a,b,ITE_MAX,EPS)
    x(1) = a;
    for i=2:ITE_MAX
        if (abs(f(x(i-1)))<EPS)    // Arret précision
            break;
        end

        // Code Méthode
        x(i) = (a+b)/2;
        if (f(a)*f(x(i))>0)
            a = x(i);
        else
            b = x(i);
        end
    end
endfunction
```

Code Source I.1 – Méthode de la dichotomie

```
function [x,i] = PointFixe(f,g,x0,ITE_MAX,EPS)
    x(1) = x0;
    for i=2:ITE_MAX
        if (abs(f(x(i-1)))<EPS)    // Arret précision
            break;
        end

        // Code Méthode
        x(i) = g(x(i-1));
    end
endfunction
```

Code Source I.2 – Méthode du Point Fixe

```
function [x,i] = Newton(f,df,x0,ITE_MAX,EPS)
    x(1) = x0;
    for i=2:ITE_MAX
        if (abs(f(x(i-1)))<EPS)    // Arret précision
            break;
        end

        // Code Méthode
        x(i) = x(i-1) - f(x(i-1))/df(x(i-1));
    end
endfunction
```

Code Source I.3 – Méthode de Newton dans $\mathbb{R}$

```
function [x,i] = Secante(f,a,b,ITE_MAX,EPS)
    x(1) = a; x(2) = b;
    for i=3:ITE_MAX
        if (abs(f(x(i-1)))<EPS)    // Arret précision
            break;
        end

        // Code Méthode
        x(i) = x(i-1) - f(x(i-1))/(f(x(i-1)) - f(x(i-2)))*(x(i-1) - x(i-2)) ;
    end
endfunction
```

Code Source I.4 – Méthode de la sécante

Les méthodes sont définies comme des fonctions dont les paramères communs à toutes les méthodes sont :
— `f` la fonction à annuler
— `df` (dérivée de f) ou g voire rien selon la méthode
— `x0` ou `a` et `b` les conditions de départ spécifique à la méthode
— `ITE_MAX` le nombre maximal d'itérations autorisées avant d'arrêter la méthode
— `EPS` la tolérance pour considérer que la fonction a été annulée
En retour, on obtient :
— `x` le vecteur colonne des solutions avec x_k l'approximation à l'itération k
— `i` l'itération a laquelle l'approximation est comprise dans le seuil de tolérance
Enfin on compare ces méthodes :

```
function y=f(x)        // Fonction à annuler
    y = x^2 - 2;
endfunction
function y=df(x)    // f'
    y = 2*x;
endfunction
function y=g(x)        // Point fixe
    y = (x+2)/(x+1);
endfunction
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD1-Pb_Non_Lineaires\1.0-Methodes.sce");

// Paramètres
ITE_MAX = 50; EPS = 1e-10;
a = 1; b = 2;
x0 = (a+b)/2;
solution = sqrt(2);

// Résultats
[Dich, ite(1)] = Dichotomie(f,a,b,ITE_MAX,EPS);
[PtFx, ite(2)] = PointFixe(f,g,x0,ITE_MAX,EPS);
[Nwtn, ite(3)] = Newton(f,df,x0,ITE_MAX,EPS);
[Scte, ite(4)] = Secante(f,a,b,ITE_MAX,EPS);

// Erreurs
errDich = abs(Dich-solution);
errPtFx = abs(PtFx-solution);
errNwtn = abs(Nwtn-solution);
errScte = abs(Scte-solution);

// Ordres
[alpha(1), C(1)] = reglin(log(errDich(1:$-1))',log(errDich(2:$))');
[alpha(2), C(2)] = reglin(log(errPtFx(1:$-1))',log(errPtFx(2:$))');
[alpha(3), C(3)] = reglin(log(errNwtn(1:$-2))',log(errNwtn(2:$-1))');
[alpha(4), C(4)] = reglin(log(errScte(1:$-2))',log(errScte(2:$-1))');
X = 0:2; ordres = alpha*X + repmat(C, 1, length(X));

// Remplissage des résulats avec la dernière approximation  pour l'affichage
maxIte = max(ite);
Dich($+1:maxIte) = Dich($); errDich($+1:maxIte) = errDich($);
PtFx($+1:maxIte) = PtFx($); errPtFx($+1:maxIte) = errPtFx($);
Nwtn($+1:maxIte) = Nwtn($); errNwtn($+1:maxIte) = errNwtn($);
Scte($+1:maxIte) = Scte($); errScte($+1:maxIte) = errScte($);

// Approximation
scf(0); clf; subplot(1,2,1);
plot((1:maxIte)', [Dich, PtFx, Nwtn, Scte]);
plot(ite', [Dich(ite(1)), PtFx(ite(2)), Nwtn(ite(3)), Scte(ite(4))], 'kx'); // Derniers résulats
title("$\huge \text{Approximation de la solution}$");
xlabel("Itération"); ylabel("Approximation");
legend("Dichotomie", "Point Fixe", "Newton", "Secante");
legend(sprintf('$\\Large \\text{Dichotomie : } k=%d$',ite(1)), sprintf('$\\Large \\text{Point Fixe : }
 ↪  k=%d$',ite(2)), sprintf('$\\Large \\text{Newton : } k=%d$',ite(3)), sprintf('$\\Large
 ↪  \\text{Sécante : } k=%d$',ite(4)));

// Erreur
subplot(1,2,2);
plot((1:maxIte)', log([errDich, errPtFx, errNwtn, errScte] +1));
title("$\huge \text{Erreur}$");
xlabel("Itération"); ylabel("Erreur (logarithme)");
legend("$\Large \text{Dichotomie}$", "$\Large \text{Point Fixe}$", "$\Large \text{Newton}$", "$\Large
 ↪  \text{Secante}$");

// Ordres
scf(1); clf;
plot(X', ordres');
title("$\huge \text{Ordre des méthodes}$");
legend(sprintf('$\\Large \\text{Dichotomie : } \\alpha=%f$',alpha(1)), sprintf('$\\Large \\text{Point
 ↪  Fixe : } \\alpha=%f$',alpha(2)), sprintf('$\\Large \\text{Newton : } \\alpha=%f$',alpha(3)),
 ↪  sprintf('$\\Large \\text{Sécante : } \\alpha=%f$',alpha(4)), 2);
```

Code Source I.5 – Comparaison des méthodes

La démarche est la suivante : on récupère les résulats des méthodes, on calcule l'écart par rapport à la solution analytique `solution` $= \sqrt{2}$, puis les ordres de convergence α par régression linéaire sur le logarithme

de l'écart absolu avec `reglin`. On affiche ensuite l'évolution des approximations et des écarts en fonction des itérations I.1 et l'ordre des méthodes I.2.

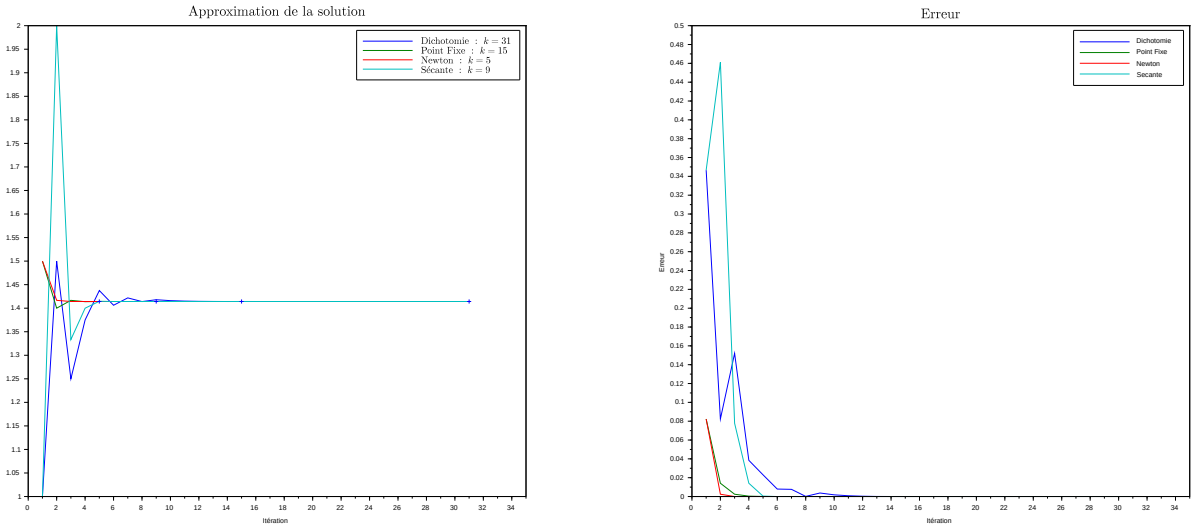


FIGURE I.1 – Résultats pour $f(x) = x^2 - 2$

Graphiquement on voit que les méthodes convergent assez vite vers la solution.

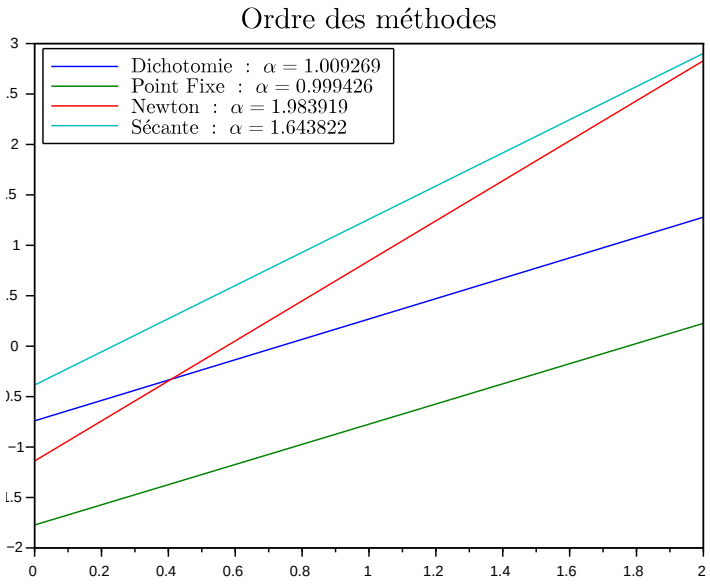


FIGURE I.2 – Représentation graphique des ordres de convergence des méthodes

On retrouve bien les même ordres de convergence pour chaque méthode. Ainsi il est préférable d'utiliser la méthode de Newton. L'outil `fsolve(x0, f, Jf)` du Scilab est pratique aussi. Il s'inspire d'une méthode complexe et peu documentée, la méthode hybride de Powell. Il suffit de renseigner $x_0 \in \mathbb{R}^n$ et f, la matrice Jacobienne J_f est optionnelle mais permet d'éviter à l'outil de devoir l'approcher et donc d'être plus rapide. Nous comparerons la méthode de Newton à `fsolve` dans l'application (I.8).

II) Méthodes numériques dans $\mathbb{R}^n$

1) Rappels

DÉFINITION I.2 : Différentiabilité
Soit $f : \mathbb{R}^n \to \mathbb{R}^n$ et $x_o \in \mathbb{R}^n$.

f est différentiable en x_0 si $\exists J_f \in \mathcal{M}_{n,n}$ tel que $\forall h \in \mathbb{R}^n$:

$$f(x_0+h) = f(x_0) + J_f h + \|h\|\epsilon(h)$$

où $lim_{h\to 0}\epsilon(h) = 0$ et la norme utilisée est la norme euclidienne. La matrice Jacobienne de f en x_0 est $J_f(x_0)$ telle que : $(J)_{ij} = \frac{\partial f_i}{\partial x_j}(x_0)$.

2) Méthode de Newton dans $\mathbb{R}^n$

Une version de la méthode de Newton existe aussi pour le cas plus général d'un système dynamique dans $\mathbb{R}^n$. On remplace alors la dérivée de f par sa matrice Jacobienne. Il faut donc que f soit différentiable.

Soit $x_0 \in \mathbb{R}^n$ choisi. $\forall x \in \mathbb{R}^n$:

$$f(x) = \underbrace{f(x_0) + f'(x_0)(x-x_0)}_{T_{x_0}} + \frac{(x-x_0)^2}{2} f''(x_0 + \theta(x-x_0))$$

avec T_{x_0} le plan tangent en x. On définit x_1 tel que

$$T_{x_0}(x_1) = 0 \tag{I.4}$$

ce qui forme un système d'équations linéaires.

$$\begin{aligned} \text{(I.4)} &\iff J_f(x_0) \times (x_1 - x_0) = -f(x_0) \\ &\implies x_1 = x_0 - f(x_0) \times \left(J_f(x_0)\right)^{-1} \end{aligned}$$

En pratique, il est préférable de résoudre le système linéaire (I.4) avec la méthode de Gauss, que d'inverser la jacobienne pour calculer x_1.

Algorithme I.3 : Méthode Newton dans $\mathbb{R}^n$

Data : $x_0 \in \mathbb{R}^n$ donné
Result : $x^* \in \mathbb{R}^n$ la solution approchée à ϵ près
while *($\|f(x_n)\| > \epsilon$ et $J_f(x_n)$ inversible)* **do**
  $x_{n+1} = x_n - J_f(x_n) \backslash f(x_n)$;
end while

On a l'implémentation dans Scilab suivante :

```
function [x,i] = NewtonRn(f,Jf,x0,ITE_MAX,EPS)
    x(:,1) = x0;
    for i=2:ITE_MAX
        if (abs(f(x(:,i-1)))<EPS)    // Arret précision
            break;
        end

        // Code Méthode
        x(:,i) = x(:,i-1) - Jf(x(:,i-1))\f(x(:,i-1));
    end
endfunction
```

Code Source I.6 – Méthode de Newton dans $\mathbb{R}^n$

Les paramères sont :
— `f` la fonction à annuler
— `Jf` la Jacobienne de f
— `x0` $\in \mathcal{M}_{n1}$ les conditions initials
— `ITE_MAX` le nombre maximal d'itérations autorisées avant d'arrêter la méthode
— `EPS` la tolérance pour considérer que la fonction a été annulée

En retour on obtient `x` $\in \mathcal{M}_{ni}$ où i est le nombre d'itérations effectuées. On obtient donc l'évolution de l'approximation à chaque colonne.

III) Applications

1) Cinématique inversée

Savoir résoudre des problèmes non-linéaires est intéressant dans de nombreux domaines, dont celui de la robotique. On a par exemple les problèmes de cinématique inversée : le but est de retrouver la bonne configuration d'un bras robotique pour qu'il atteigne un point particulier sous certaines contraintes.

On modélise ensuite la situation par le système suivant :

$$M(\theta) = A \iff \begin{cases} l_1\cos(\theta_1) + l_2\cos(\theta_1+\theta_2) - x_A &= 0 \\ l_1\sin(\theta_1) + l_2\sin(\theta_1+\theta_2) - y_A &= 0 \end{cases} \tag{I.5}$$

On pose $f : \mathbb{R}^2 \to \mathbb{R}^2$ tel que

$$f(\theta) = \begin{pmatrix} l_1\cos(\theta_1) + l_2\cos(\theta_1+\theta_2) - x_A \\ l_1\sin(\theta_1) + l_2\sin(\theta_1+\theta_2) - y_A \end{pmatrix} \tag{I.6}$$

On a équivalence entre (I.5) et $f(\theta) = 0$.

On résout alors (I.6) par une méthode numérique, ici celle de Newton. Pour cela, nous devons calculer la Jacobienne de f :

$$J_f(\theta) = \begin{pmatrix} -l_1\sin(\theta_1) - l_2\sin(\theta_1+\theta_2) & -l_2\sin(\theta_1+\theta_2) \\ l_1\cos(\theta_1) + l_2\cos(\theta_1+\theta_2) & l_2\cos(\theta_1+\theta_2) \end{pmatrix}$$

Nous avons le programme Scilab suivant :

```
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD1-Pb_Non_Lineaires\dessine_bras.sce");

function Y = f(X)
    Y = [
        l1*cos(X(1)) + l2*cos(sum(X))-M(1)
        l1*sin(X(1)) + l2*sin(sum(X))-M(2)
    ];
endfunction
function Y = Jf(X)      // Jacobienne calculée
    Y = [
        -l1*sin(X(1))-l2*sin(sum(X)), -l2*sin(sum(X))
        l1*cos(X(1))-l2*cos(sum(X)),  l2*cos(sum(X))
    ];
endfunction

// Données
l1 = 1;
l2 = 1;
X = [1; 1];

// Affichage des mouvements successifs
NB_POSITIONS = 50;
myColor = jetcolormap(NB_POSITIONS);
t = linspace(0, 2*%pi, NB_POSITIONS)
clf;
for i=1:NB_POSITIONS
    M = [
        1+0.5*cos(t(i))
        1+0.5*sin(t(i))
    ];
    X = fsolve(X, f);
    dessine_bras(X, 1, myColor(i,:));
end
```

Code Source I.7 – Cinématique inversée

On cherche aussi à décrire une trajectoire circulaire de centre $(1,1)$ et de rayon 0.5. Pour cela on calcules les `NB_POSITIONS` positions successives lorsque le bras atteint le point M, ce dernier variant autour du cercle à chaque itération. La fonction `dessine_bras(X,1)` dessine le bras avec les angles X_1 et X_2 et le cercle si le second paramètre est égale à 1. On obtient les positions successives du bras articulé :

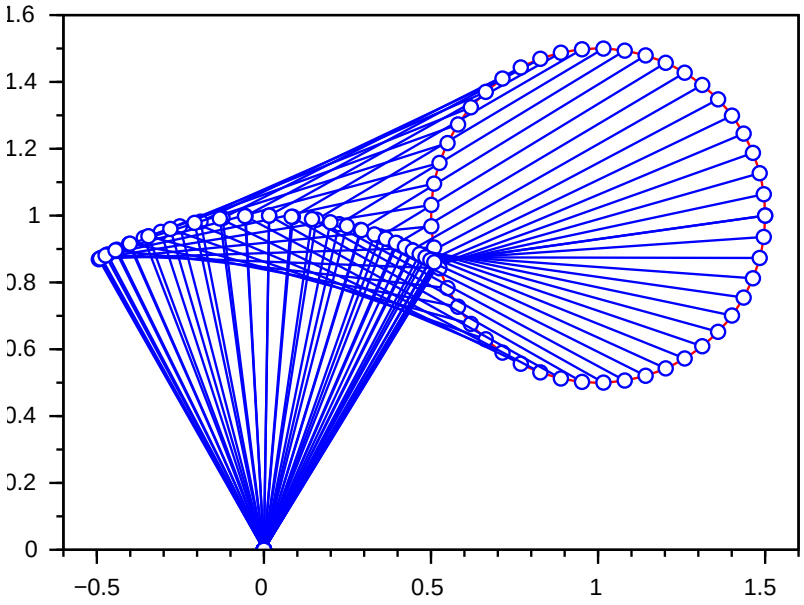


FIGURE I.3 – Mouvement autour d'un cercle

2) GPS

Pour se localiser dans l'espace, un GPS (*Global Positioning System* ou en français *Géo-Positionnement par Satellite*) résoud un problème non-linéaire dans $\mathbb{R}^3$. Pour fonctionner, il doit être connecté à au moins 3 satellites afin de garantir une solution unique. En connaissant la distance [1] qui les séparent et leur position dans l'espace, on forme un système d'équations non-linéaires. De plus des problèmes de synchronisation temporelle entre les satellites et le GPS peuvent s'ajouter pour calculer les erreurs.

On va ici résoudre ce problème dans le cas de 3 satellites sans tenir compte de problèmes temporels. Soient $S_1, S_2, S_3 \in \mathcal{M}_{31}$ tels que $S_i = (x_i, y_i, z_i)$ les positions des satellites dans l'espace, ainsi que $d \in \mathcal{M}_{31}$ où d_i est la distance séparant le satellite S_i au GPS. On cherche $X = (x, y, z) \in \mathcal{M}_{31}$ tel que $f(X) = 0$:

$$f(X) = \begin{pmatrix} \|X - S_1\|^2 - d_1 \\ \|X - S_2\|^2 - d_2 \\ \|X - S_3\|^2 - d_3 \end{pmatrix} \tag{I.7}$$

On pose : $g_i(X) = \|X - S_i\|^2$

Pour trouver la Jacobienne sans calculer chaque dérivée partielle on effectue le développement limité suivant :

$$\begin{aligned} g_i(X+h) &= \|(X-S)+h\|^2 \\ &= ((X-S)+h)^T((X-S)+h) \\ &= (X-S)^T(X-S) + h^T(X-S) + (X-S)^T h + h^T h \\ &= \underbrace{\|X-S\|^2}_{g(X)} + \underbrace{2(X-S)^T}_{J_g(X)} h + \underbrace{\|h\|^2}_{\text{Reste}} \end{aligned}$$

Ainsi on a la Jacobienne :

$$J_g(X) = 2 \times \begin{pmatrix} (X-S_1)^T \\ (X-S_2)^T \\ (X-S_3)^T \end{pmatrix} \tag{I.8}$$

On peut donc résoudre le problème avec Scilab.

1. Le satellite envoie un signal contenant l'heure d'émission, le GPS peut alors calculer la distance à partir de l'heure de réception et de la vitesse de l'onde.

```
function Y=f(X)
    Y = [
        norm(X-S1)^2 - d(1)^2;
        norm(X-S2)^2 - d(2)^2;
        norm(X-S3)^2 - d(3)^2;
    ];
endfunction
function Y=Jf(X)        // Jacobienne calculée à la main
    Y = 2*[
        (X-S1)'
        (X-S2)'
        (X-S3)'
    ];
endfunction

// Données
S1 = [-11716.227778; -10118.754628; 21741.083973];
S2 = [-12082.643974; -20428.242179; 11741.374154];
S3 = [14373.286650;  -10448.439349; 19596.404858];
d  = [22163.847742;  21492.777482;    21492.469326];
R = 6371;            // Rayon Terre
X0 = zeros(3,1);

// Verification Jacobienne pour X0
Jth = Jf(X0);
Jex = numderivative(f, X0);
ecartJ = abs(Jth-Jex);


// Résulats
Xfsolve = fsolve(X0, f, Jf);

exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD1-Pb_Non_Lineaires\1.5-NewtonRn.sce");
[Xnewton, ite] = NewtonRn(f,Jf,X0,1000,1e-10);
Xnewton = Xnewton(:,$);
ecartX = abs(Xnewton-Xfsolve);
altitude = norm(Xnewton) - R;

// Affichage
disp("== Vérification de la Jacobienne calculée ==");
disp(Jth, "Jacobienne théorique");
disp(Jex, "Jacobienne approchée");
disp(ecartJ, "Ecart entre les deux:");

disp("== Comparaison des résultats ==");
disp(Xnewton, "Résulat NEWTON");    // résultat
disp(Xfsolve, "Résulat FSOLVE");
disp(ecartX, "Ecart relatif");

disp(altitude, "Altitude par rapport à la surface de la Terre");
```

Code Source I.8 – Simulation d'un GPS

Ici, j'ai fait diverses comparaisons :
— entre la méthode de Newton et l'outil `fsolve` de Scilab I.1
— entre la matrice Jacobienne (I.8) et celle calculée par Scilab avec `numderivative`
J'ai alors obtenu les coordonnées et leur écart absolu :

Méthode de Newton	Macro `fsolve`	Écart absolu
595.0250498015592	595.0250498015607	1.4779288903810084E-12
-4856.025050498366	-4856.025050498369	2.7284841053187847E-12
4078.329999324317	4078.3299993243168	4.547473508864641E-13

TABLE I.1 – Résulats de la simulation du GPS

On obtient une altitude par rapport au rayon moyen de la Terre de −1.7135655 m. On est ainsi relativement proche de la surface du globe car les données possèdent des incertitudes de même ordre.

CHAPITRE II

Fractales

Les fractales ont été définies par Benoit Mandelbrot en 1975 dans son oeuvre *Les Objets Fractals*. Ce sont des objets complexes que la géométrie traditionnelle peine à décrire. Le plus souvent, ce sont des figures qui se répètent à l'infini avec n'importe quel niveau de zoom : on parle d'autosimilarité à toutes les échelles.

Les fractales peuvent être appliquées à de nombreux domaine :

— en médecine avec la modélisation d'un poumon
— la modélisation de structures de plantes comme le chou romanesco
— en finance avec la prévision de krachs boursiers par la théorie multifractale
— en urbanisme avec les murs antibruits
— en géologie avec l'étude du relief
— mais aussi dans les arts, par leur aspect intriguant et infini

Nous allons ici étudier comment générer des fractales, puis nous verrons quelques exemples.

I) Outils

1) Distance et dimension de Hausdorff

On cherche ici à mesurer convenablement la distance entre deux compacts de $\mathbb{R}^2$. On définit la distance entre un point x et un compact B par : $d(x,B) = \min_{b\in B} d(x,b)$ Naïvement on pourrait définir la distance donc par : $d(A,B) = \max_{a\in A} d(a,B)$ Mais on a alors un problème car $d(A,B) \neq d(B,A)$. Nous avons une solution :

DÉFINITION II.1 :
On définit la distance de Hausdorff d_H entre deux compacts A et B par :

$$d_H(A,B) = \max(d(A,B), d(B,A)) = \max\left(\max_{a\in A} d(a,B);\ \max_{b\in B} d(b,A)\right) \tag{II.1}$$

De plus, on définit la dimension de Hausdorff d d'un compact $K \subset \mathbb{R}^2$:

DÉFINITION II.2 : Dimension de Hausdorff
On note $N(\epsilon)$ le nombre de carrés ou disques de longueur ϵ recouvrant un compact K, on a la dimension de Hausdorff de K notée $\dim_H$ telle que :

$$\dim_H = \lim_{\epsilon\to 0} \frac{\ln N(\epsilon)}{\ln \frac{1}{\epsilon}} \tag{II.2}$$

On peut aussi l'expliquer plus simplement. On prend un élément d'un compact K et on le réduit par d'un facteur k (on a le rapport de réduction $r = \frac{1}{k}$), pour recouvrir le compact initial il faut n réductions. On a donc $n \times k^d = 1$ d'où la dimension :

$$d = \frac{\ln n}{\ln k} \tag{II.3}$$

EXEMPLE :
Pour $\mathbb{R}$: on prend un segment que l'on réduit de k, on a besoin de $n = k$ segments réduits.
Pour $\mathbb{R}^2$: on prend un carré que l'on réduit de k, on a besoin de $n = k^2$ carrés réduits.
Pour $\mathbb{R}^3$: on prend un cube que l'on réduit de k, on a besoin de $n = k^3$ cubes réduits.
Par exemple si on réduit un cube d'un facteur 2, il faut $8 = 2^3$ cubes réduits.

2) Système de Fonctions Itérées (ISF)

Les ISF sont des algorithmes qui permettent de construire des fractales de manière itérée, évitant ainsi les méthodes récursives. Avant de les définir plus précisément nous allons voir les transformations utilisées dans ces méthodes.

Soit $u : \mathbb{R}^2 \to \mathbb{R}^2$ une transformation linéaire du plan et soit A la matrice associée à cette application. Les transformations les plus courantes sont :

l'homothétie de rapport λ :	$A = \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix}$
la rotation d'angle θ et de centre l'origine :	$A = R_\theta = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$
la symétrie axiale par rapport à l'axe des ordonnée :	$A = S_{Ox} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
la combinaison de rotation θ et d'homothétie ρ :	$A = \begin{pmatrix} a & -b \\ b & a \end{pmatrix}$
Où on a :	$\rho = \sqrt{a^2+b^2}$ $\quad \cos\theta = \frac{a}{\rho}$ $\quad \sin\theta = \frac{-b}{\rho}$

TABLE II.1 – Transformations courantes

En revanche, la translation par $t \in \mathbb{R}^2$ n'est pas linéaire car $u(0) = 0 + t \neq 0$. Il s'agit d'une transformation affine.

DÉFINITION II.3 : Transformation Affine
Une transformation affine $T : \mathbb{R}^2 \to \mathbb{R}^2$ d'un point $M(x; y)$ est la composition d'une transformation linéaire A et d'une translation t :

$$T(M) = AM + t = (ax + by + e,\, cx + dy + f) \tag{II.4}$$

$$= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix} \tag{II.5}$$

DÉFINITION II.4 : Compact
Un compact de $\mathbb{R}^2$ est un sous-ensemble fermé et borné de $\mathbb{R}^2$

DÉFINITION II.5 : Contraction
Une transformation affine du plan est une contraction de facteur $r \in]0;1[$ si l'image d'un segment est un segment de longueur inférieur (longueur initial $\times r$).

DÉFINITION II.6 : Système de Fonctions Itérées
Un Système de Fonctions Itérées (ISF) est une collection finie de n transformation affines T_i.

DÉFINITION II.7 : Attracteur
L'attracteur d'un IFS de $T_1, T_2, ..., T_n$ est l'unique compact $K \in \mathbb{R}^2$ tel que

$$K = T(K) = \bigcup_{i=1}^{n} T_i(K) = T_1(K) \cup T_2(K) \cup ... \cup T_n(K) \tag{II.6}$$

On pose : E_k le compact formé à l'itération k d'un ISF à partir du compact de départ $E_0 \subset \mathbb{R}^2 : E_{k+1} = T_i(E_k)$. On parle d'ISF aléatoire quand la transformation T_i est choisie aléatoirement avec un probabilité p_i parmi les n transformations possibles. Il faut que $\sum_{i=1}^{n} p_1 = 1$. Ce type d'ISF converge de la même façon que la version déterministe.

THÉORÈME II.1 : Théorème de Banach
Si $f : \mathbb{R}^2 \to \mathbb{R}^2$ est une contraction de facteur $r \in]0;1[$ telle que

$$d_H(f(A), f(B)) \leq r \times d_H(A, B)$$

alors il existe un point fixe K appelé attracteur tel que $K = f(K)$.

Ce théorème nous permet alors de prouver la convergence des ISF. La suite $(E_k)_{k\in\mathbb{N}}$ converge vers l'attracteur K décrit en (II.6) si ses transformations T_i sont contractantes selon la distance de Haussdorf vue en II.2.

Comme tous les ISF que nous construirons par la suite n'auront que des transformations contractantes, alors ils convergeront vers leur point fixe qui correspond à leur fractale.

II) Quelques fractales

1) Ensemble de Mandelbrot

L'ensemble de Mandelbrot est sûrement l'une des fractales les plus connues et les plus étudiées. Elle a été découvert par Gaston Julia et Pierre Fatou, puis repris par Benoît Mandelbrot qui lui donnera son nom.

i) Construction

On définit l'ensemble de Mandelbrot $\mathbb{M}$ de la façon suivante : un nombre complexe c appartient à $\mathbb{M}$ si la suite (II.7) est bornée et donc ne diverge pas.

$$\begin{cases} z_{n+1} = z_n^2 + c \\ z_0 = 0 \end{cases} \tag{II.7}$$

On considère que comme borne suffisante 2 : $|z_n| < 2\ \forall n \in \mathbb{N}$ car si $|z_n| \geq 2$ alors $|z_{n+1}| \geq 2z_n$ et la suite diverge.

On a l'implémentation sous Scilab suivante :

```
clear;
ITE_MAX = 400;

// Grille des points
x = linspace(-2, 1, 800);
y = linspace(-1.5, 1.5, 600);
[X, Y] = ndgrid(x,y);

Z = X + %i*Y;
C = Z;                          // Z0 = C = chaque point
Borne = max(abs(C),2);

for i=1:ITE_MAX
    Z = Z.^2 + C;
end

// Sélection des points
InMandelbrot = zeros(length(x), length(y));
Cv = abs(Z)<Borne;
InMandelbrot(Cv) = 1;

clf;
set(gca(), "isoview", "on");
set(gcf(), "color_map", graycolormap(ITE_MAX));
grayplot(x,y,InMandelbrot);
```

Code Source II.1 – Ensemble de Mandelbrot

On crée une grille de 800×600 pixels, et à chaque point (x, y) on associe le complexe $c = x + iy$. On va tester si chaque complexe correspondant ne diverge pas pour l'inclure dans l'ensemble de Mandelbrot. On définit la matrice complexe `Z` qui correspond aux coordonnées complexes de chaque point et `C` $= Z_0$. On itère ensuite chaque point dans Z avec la fonction (II.7).

La matrice `InMandelbrot` permet de savoir si les complexes restent bornés : on met 1 si le point correspond n'a pas divergé après les itérations, il appartient ainsi à l'ensemble de Mandelbrot ; sinon on laisse à 0. On affiche ensuite les points à 1 et on obtient :

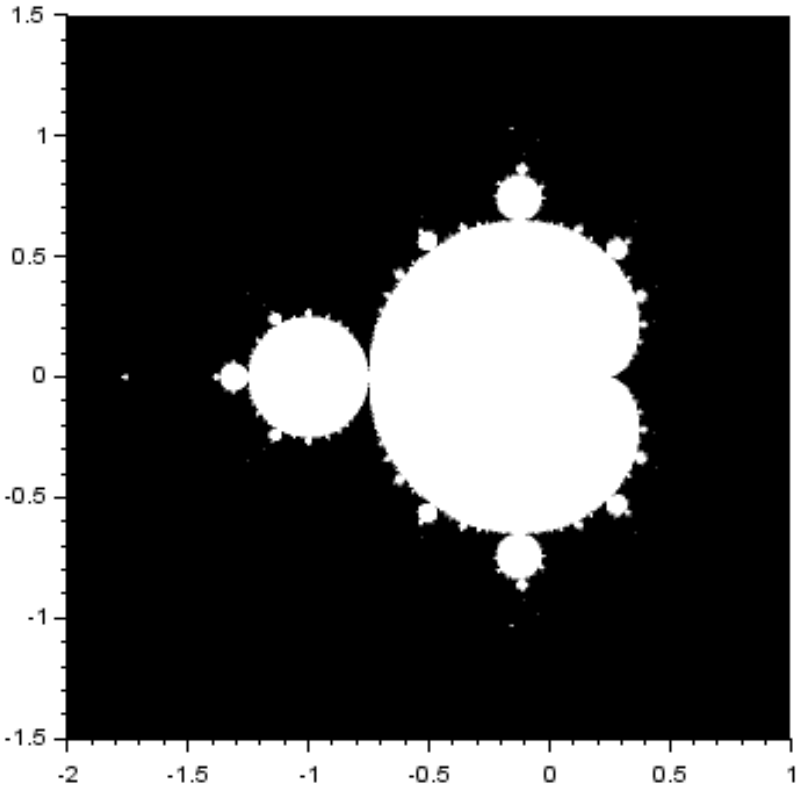


FIGURE II.1 – Ensemble de Mandelbrot

ii) Propriétés

Propriété II.1 : L'ensemble de Mandelbrot a été démontré connexe.

On peut aussi retrouver les décimales de π avec l'ensemble de Mandelbrot. On note $N(z)$ le nombre d'itérations nécessaires avant que le point d'affixe z ne diverge de l'ensemble de Mandelbrot. Prenons la suite $z_i = \frac{1}{4} + 10^{-2i}\ \forall i \in \mathbb{N}^*$. On calcules $N(z_i)$ avec le programme suivant :

```
clear;
ITE_MAX = 10^10;
NB_I = 5;

Z = ones(1,NB_I)./4;
Cv = []
for i=1:NB_I
    Z(i) = Z(i) + 10^(-2*i);
    Cv = [Cv %T];
end
C = Z;

Borne = max(abs(C),2);
N = zeros(1, NB_I);

for k=1:ITE_MAX
    if or(Cv) == %F then    // Toutes les valeurs ont divergé
        break
    end;
    Z(Cv) = Z(Cv).^2 + C(Cv);
    Cv = abs(Z)<Borne;
    N(Cv) = N(Cv) + 1;
end

for i=1:NB_I    // Affichage
    printf("\nPour i = %d, z = %1.16f, N(z) = %d",i,C(i),N(i));
end
```

Code Source II.2 – Pi et Mandelbrot

On récupère les résultats suivants :

i	z_i	$N(z_i)$
1	0.26	28
2	0.2501	310
3	0.250001	3138
4	0.25000001	31412
5	0.2500000001	314155

TABLE II.2 – Retrouver les décimales de π avec l'ensemble de Mandelbrot

On retrouve étonnement les décimales de π dans $N(z_i)$ quand i augmente. En comparaison, on a : $\pi = 3.1415927$. Ainsi il s'agit d'une méthode originale pour retrouver les décimales de π mais extrêmement lente.

2) Ensembles de Julia

Soit $c \in \mathbb{C}$ fixé, les complexes de la suite $(z_n)_{n\in\mathbb{N}}$ appartiennent à l'ensemble de Julia $\mathbb{J}_c$ s'il sont bornés par 2 de la même manière que pour l'ensemble de Mandelbrot.

$$\begin{cases} z_{n+1} = z_n^2 + c \\ z_0 \in \mathbb{C} \end{cases} \tag{II.8}$$

On a l'implémentation sous Scilab suivante :

```
clear; clf;
RES = 600; UPTO = 1;
ITEMAX = 300;

// Grille
x = linspace(-UPTO, UPTO, RES);
y = linspace(-UPTO, UPTO, RES);

// Choix du c
c(1) = 0.32    +0.043*%i;
c(2) = -0.338  -0.622*%i;
c(3) = -0.8    +0.156*%i;
c(4) = -0.755  -0.042*%i;
//c(4) = -0.1011 + 0.9563*%i;
//c(4) = -1.401155;
//c(4) = 0.3774  + 0.19*%i;
//c(4) = 0.3     - 0.49*%i;
//c(4) = -0.708  -0.23*%i;

for k=1:length(c)
    ck = c(k);
    [X, Y] = meshgrid(x,y);
    Z = X+ %i*Y;

    Color = zeros(length(x), length(y));
    Borne = max(abs(ck),2);

    for i=1:ITEMAX
        Z = Z.^2 + ck;
        Cv = abs(Z)<Borne;
        Color(Cv) = Color(Cv)+1;
    end

    subplot(1,4,k);
    set(gca(), "isoview", "on");
    set(gcf(), "color_map", oceancolormap(ITEMAX));
    grayplot(x,y,Color);
    title(sprintf("$$\\huge c = %f + %f i$$", real(ck), imag(ck)), 'position', [-0.7 1.1]);
end
```

Code Source II.3 – Ensemble de Julia

On crée de la même manière que précédemment une grille de points complexes que l'on va itérer, sauf qu'ici chaque complexe correspond à z_0 et c est fixé. `Cv` est une matrice binaire : si z est encore borné alors `Cv(z)` $= 1$. `Color` est la matrice de coloration de la map pixel en fonction du nombre d'itérations effectuées avant que le complexe ne diverge : on augmente `Color(x,y)` tant que le complexe associé est borné.

On a choisi ici plusieurs valeurs de c donnant les fractales suivantes :

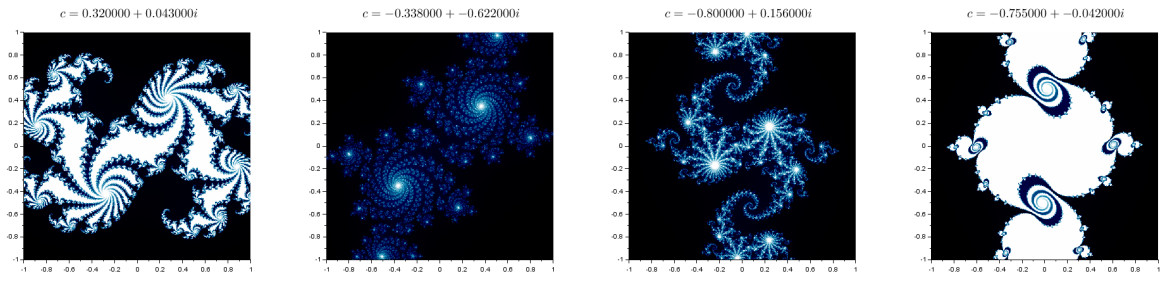


FIGURE II.2 – Ensembles de Julia

On observe que pour $c \in \mathbb{M}$ et que ces ensembles sont connexes. Il s'avère que pour chaque $c \in \mathbb{M}$, l'ensemble de Julia $\mathbb{J}_c$ est connexe.

3) Ensemble de Cantor

i) Construction

On construit l'ensemble de Cantor E_n de profondeur n de la manière suivante :

Algorithme II.1 : Ensemble de Cantor

Data : Segment $E_0 = [0,1]$
n la profondeur désirée
Result : E_n l'ensemble de Cantor de profondeur n
for *i allant de* 0 *à n* **do**
  Partager chaque segment en trois parties égales
  Retirer la partie centrale de chaque segment
end for

Cet algorithme décrit un ISF déterministe avec les transformations suivantes :

$$T_1(x) = \frac{1}{3}x$$
$$T_2(x) = \frac{1}{3}x + \frac{2}{3}$$
$$E_{k+1} = T_1(E_k) \cup T_2(E_k)$$

D'après le théorème (II.1), cet ISF converge bien vers l'ensemble triadique de Cantor E_∞ tel que :

$$E_\infty = \bigcap_{i=0}^{\infty} E_i \tag{II.9}$$

où E_i l'ensemble obtenu à l'étape i.

On implémente l'ISF déterministe sous Scilab :

```
function Cantor(n, a, b)
    if n>0
        plot([a,b], [n,n]);
        c = a+(b-a)/3;
        d = b-(b-a)/3;
        Cantor(n-1, a, c);
        Cantor(n-1, d, b);
    end
endfunction

n = 7;
a = 0; b = 1;

clf;
Cantor(n, a,b);
replot([0 0 1 n+1]);
```

Code Source II.4 – Ensemble de Cantor

Le déterministe se fait par récursivité. On obtient le résultat suivant :

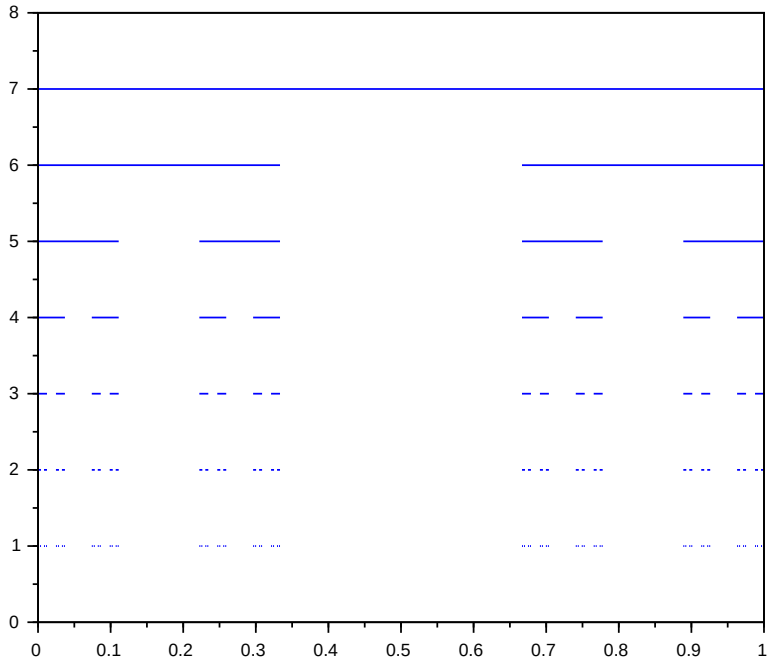


FIGURE II.3 – Ensemble de Cantor de profondeur $n = 7$

ii) Propriétés

L'ensemble de Cantor est auto-similaire dans le sens où le local et le global se ressemblent, c'est-à-dire qu'à différentes échelles la figure paraît la même. Sa structure est telle qu'elle ne peut être décrite par la géométrie traditionnelle mais seulement par la représentation triadique suivante.

THÉORÈME II.2 :
On a la décomposition en base 3 suivante $\forall x \in E_\infty$:

$$\underline{x}_{10} = \sum_{k=1}^{\infty} \frac{x_i}{3^i} \tag{II.10}$$

$$\Longleftrightarrow \underline{x}_3 = 0, x_1 x_2 x_3 \dots x_k \dots \tag{II.11}$$

Avec $x_k = 0$ ou 2, $\forall k$

EXEMPLE :
On a la décomposition suivante :

$$\begin{aligned}\left(\frac{1}{3}\right)_3 &= 0,02222222\dots \\ &= 0 + \frac{0}{3} + \frac{2}{3^2} + \dots + \frac{2}{3^k} \\ &= \frac{2}{3^2}\left(1 + \frac{2}{3} + \dots + \frac{2}{3^{k-2}}\right) \\ &= \frac{2}{9} \times \frac{1}{1-\frac{1}{3}} = \frac{1}{3} = 0.1\end{aligned}$$

Donc $\frac{1}{3} \in E_\infty$

Propriété II.2 : $|E_\infty| = 0$

Preuve :
On a : $|E_0| = 1$, $|E_1| = \frac{2}{3}$, $|E_2| = \frac{4}{9} = \left(\frac{2}{3}\right)^2$, etc. Par récurrence on obtient : $|E_k| = \left(\frac{2}{3}\right)^k$. Donc $|E_k| \xrightarrow[k\to\infty]{} 0$.

Propriété II.3 : E_∞ est indénombrable.

Preuve :
Montrons que l'ensemble E_∞ est en bijection avec $[0;1]$ qui est indénombrable.

$$\forall x \in E_\infty : \text{(II.10)} \rightarrow \frac{x}{2} = 0, \frac{x_1}{2}\frac{x_2}{2}\ldots\frac{x_k}{2}\ldots \text{ avec } \frac{x_k}{2} \in [0;1]$$

On associe à $\frac{x}{2}$ le nombre en base 2 : $0, y_1 y_2 y_3 \ldots y_k \ldots$ avec $y_k \in [0;1]$

iii) Dimension

La dimension de E_∞ est :

$$\left.\begin{array}{l} r = \frac{1}{3} \\ N = 2 \end{array}\right\} \quad \text{à l'étape } k : \quad 2^k = (3^k)^d \Longrightarrow 2 = 3^d \Longrightarrow d = \frac{\ln 2}{\ln 3} \in]0;1[$$

4) Flocon de Von Koch

i) Construction

On construit le flocon de la manière suivante :

Algorithme II.2 : Flocon de Von Koch

Data : Triangle équilatéral C_0 de coté 1
n la profondeur désirée
Result : C_n le flocon de Von Koch de profondeur n
for *i allant de* 1 *à n* **do**
  Partager chaque segment en 3 segments égaux
  Remplacer le segment central par deux segments égaux formant un triangle équilatéral ayant pour base le segment remplacé
end for

On obtient le résultat suivant :

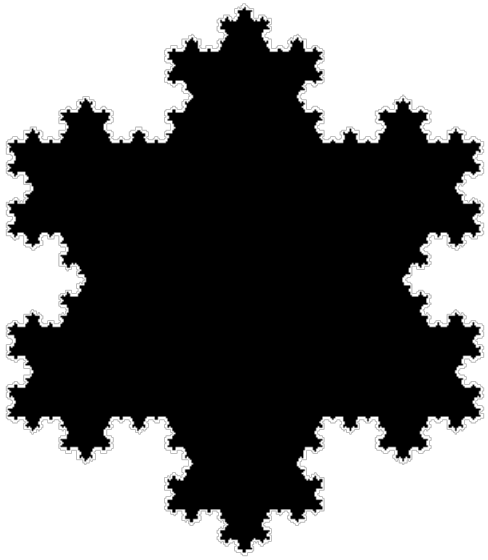


FIGURE II.4 – Flocon de Von Koch

La courbe de Von Koch correspond à un segment du flocon. Elle est de longueur infinie mais contenant dans une surface d'aire finie $[0;1] \times [0;1]$.

ii) Propriétés

Propriété II.4 : Le flocon de Von Koch a un périmètre infinie.

Preuve :

$$l_0 = 1 \quad \mapsto l_1 = \frac{4}{3} \quad \mapsto \ldots \mapsto l_k = \left(\frac{4}{3}\right) \underset{k\to\infty}{\to} +\infty$$

Propriété II.5 : Le flocon de Von Koch a une aire finie.

Preuve :

$$\begin{aligned} &A_0 \in \mathbb{R} \\ &A_1 = A_0 + 3 \times \frac{A_0}{9} = A_0 + \frac{A_0}{3} \\ &A_2 = A_1 + 12 \times \frac{A_0}{9^2} = A_0 + \frac{A_0}{3} + \frac{2}{3^2} A_0 \end{aligned}$$

Par récurrence :

$$\begin{aligned} A_k &= A_0 + \sum_{i=0}^{k} \frac{2^i}{3^{i+1}} A_0 = A_0 + \frac{A_0}{3} \times \frac{1 - \left(\frac{2}{3}\right)^k}{\frac{1}{3}} = A_0 + A_0 \times \left(1 - \left(\frac{2}{3}\right)^k\right) \\ A_\infty &= \lim_{k\to\infty} A_k = 2A_0 \end{aligned}$$

Le flocon de Von Koch définit donc une surface finie contenue dans une courbe de longueur infinie.

5) Triangle de Sierpinski

i) Construction

On construit cette fractale de la manière suivante :

Algorithme II.3 : Triangle de Sierpinski

Data : Triangle équilatéral E_0 de coté 1
n la profondeur désirée
Result : E_n le triangle de Sierpinski de profondeur n
for *chaque triangle jusqu'à la profondeur n* **do**
    Partager le triangle en 4 triangles égaux
    Retirer le triangle central
end for

On a l'implémentation sous Scilab suivante :

```
function Sierpinsky(a, b, c, n)
    if n>0 then
        // Calcul milieux segments
        D = (a+b)/2;
        E = (b+c)/2;
        F = (c+a)/2;
        K = [D E F];

        // Tracé et appels récursif
        xfpoly(K(1,:), K(2, :), 8);
        Sierpinsky(a,D,F,n-1);
        Sierpinsky(D,b,E,n-1);
        Sierpinsky(F,E,c,n-1);
    end

endfunction

a = [0; 0];
b = [0.5; sqrt(3)/2];
c = [1; 0];
T = [a b c];

clf;
xfpoly(T(1,:), T(2,:), 1);
set(gca(), 'isoview', 'on');

n = 9;
replot([0 0 1 1]);
Sierpinsky(a,b,c,n);
```

Code Source II.5 – Triangle de Sierpinski

On obtient le résultat suivant :

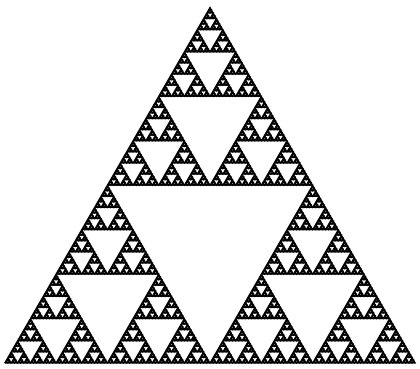


FIGURE II.5 – Triangle de Sierpinski de profondeur $n = 7$

On peut aussi construire l'ISF suivant composé d'une homothétie de rapport $r = \frac{1}{2}$ et quelques translations :

$$T_1(x,y) = \frac{1}{2}\begin{pmatrix} x \\ y \end{pmatrix}$$

$$T_2(x,y) = \frac{1}{2}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix}$$

$$T_3(x,y) = \frac{1}{2}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{4} \\ \frac{\sqrt{3}}{4} \end{pmatrix}$$

On implémente cet ISF de type aléatoire sous Scilab :

```
r = 1/2;
D = [1/2; 0];
H = [1/4; sqrt(3)/4];
n = 100000;

W = zeros(2, n);
for i=1:n-1
    t=floor(3*rand(1)+1);
    select t
    case 1 then
        W(:, i+1) = r*W(:,i);
    case 2 then
        W(:, i+1) = r*W(:,i) + D;
    case 3 then
        W(:, i+1) = r*W(:,i) + H;
    end
end

clf;
replot([0 0 1 1]);
set(gca(), "isoview", "on");
//title(sprintf('$$\\huge %d Points$$',n));
plot(W(1,:),W(2,:),".",'markersize',1)
```

Code Source II.6 – Triangle de Sierpinski Itératif

Le choix des transformations est équiprobable et la construction du triangle est itérative. Le résultat est le suivant :

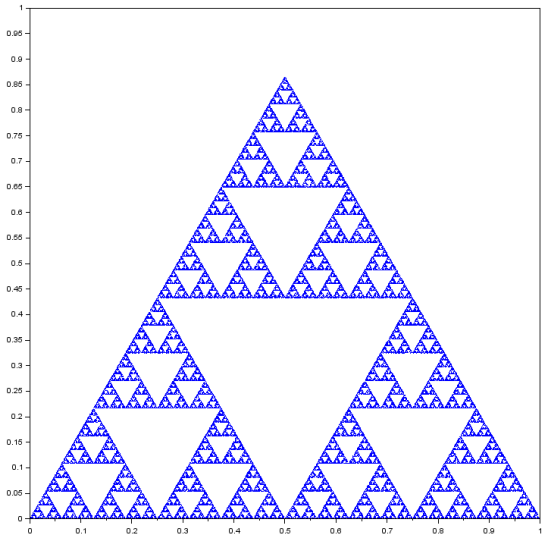


FIGURE II.6 – Triangle de Sierpinski itératif avec $n = 100000$ points

On observe une légère différence de rendu entre les deux figures II.5 et II.6 mais les deux décrivent bien la même fractale.

ii) Propriétés

On note E_∞ le triangle de Sierpinski.

Propriété II.6 : L'aire du triangle de Sierpinski est nulle.

Preuve :

On suppose que Aire $E_0 = 1$. On a :

$$\text{Aire } E_1 = 1 - \frac{1}{4}$$
$$\text{Aire } E_2 = 1 - \frac{1}{4} - 3 \times \left(\frac{1}{4}\right)^2$$

Par récurrence, on a :

$$\begin{aligned}\text{Aire } E_\infty &= 1 - \sum_{k=0}^{\infty} 3^k \times \frac{1}{4^{k+1}} \\ &= 1 - \frac{1}{4} \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k \\ &= 1 - \frac{1}{4} \times \left(\frac{1 - \left(\frac{3}{4}\right)^\infty}{1 - \frac{3}{4}}\right) \\ &= 0\end{aligned}$$

iii) Dimension

La dimension d du triangle de Sierpinski E^∞ est :

$$\left.\begin{array}{r} r = \frac{1}{2} \\ N = 3 \end{array}\right\} \quad 2 = 3^d \implies d = \frac{\ln 3}{\ln 2}$$

6) Tapis de Sierpinski

i) Construction

Le tapis ou napperon de Sierpinski part du même principe que le triangle mais avec un carré. On a alors un rapport de $r = 1/3$ et 9 sous-carrés dont 1 seul enlevés à chaque itérations.

Algorithme II.4 : Tapis de Sierpinski

Data : Carré E_0 de coté 1
n la profondeur désirée
Result : E_n le tapis de Sierpinski de profondeur n
for *chaque carré jusqu'à la profondeur n* **do**
  Partager le triangle en 9 carrés égaux
  Retirer le carré central
end for

On a l'ISF suivant décrivant une homothétie de rapport $r = \frac{1}{3}$ et quelques translations :

$$\begin{array}{lll} \text{Translations :} & d = \begin{pmatrix} \frac{1}{3} \\ 0 \end{pmatrix} & h = \begin{pmatrix} 0 \\ \frac{1}{3} \end{pmatrix} \\ \\ \text{Transformations :} & T_3(M) = rM + 2d & T_6(M) = rM + d + 2h \\ T_1(M) = rM & T_4(M) = rM + 2d + h & T_7(M) = rM + 2h \\ T_2(M) = rM + d & T_5(M) = rM + 2d + 2h & T_8(M) = rM + h \end{array} \tag{II.12}$$

On peut donc implémenter cet ISF de manière déterministe et aléatoire avec Scilab :

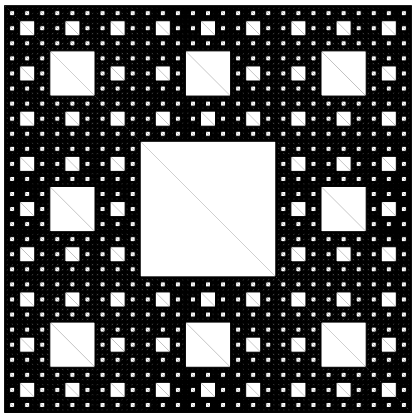
```
function TapisSierpinsky(origin, k, n)
if k>0 then
    r = 1/(3^(n-k+1));
    D = [r; 0];
    H = [0; r];

    // Calcul des points inférieurs gauche de chaque carré
    a = origin;
    b = origin + D;
    c = origin + 2*D;
    d = origin + 2*D + H;
    e = origin + 2*D + 2*H;
    f = origin + D + 2*H;
    g = origin + 2*H;
    h = origin + H;
    i = origin + D + H;

    // Enlève le carré centrale
    centre = [i d e f];
    xfpoly(centre(1,:), centre(2,:), 8);

    clear origin r D H centre i;

    TapisSierpinsky(a, k-1, n);
    TapisSierpinsky(b, k-1, n);
    TapisSierpinsky(c, k-1, n);
    TapisSierpinsky(d, k-1, n);
    TapisSierpinsky(e, k-1, n);
    TapisSierpinsky(f, k-1, n);
    TapisSierpinsky(g, k-1, n);
    TapisSierpinsky(h, k-1, n);
end
endfunction

clf;
set(gca(), 'isoview', 'on');

n = 5;
origin = [0; 0];
xfpoly([0 1 1 0], [0 0 1 1], 1);
replot([0 0 1 1]);
TapisSierpinsky(origin, n, n);
```

Code Source II.7 – Tapis de Sierpinski

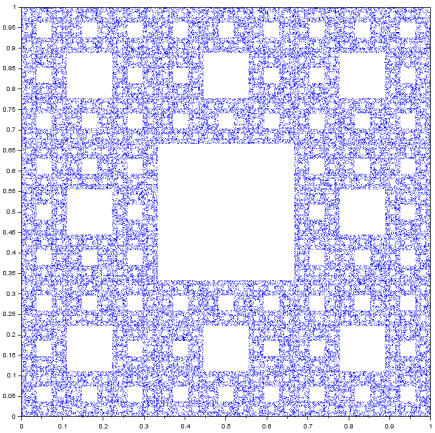
```
r = 1/3;            // Réduction
D = [1/3; 0];       // Matrice de translation à droite
H = [0;   1/3];     // Matrice de translation en haut
n = 100000;         // Nombre de points à calculer

W = zeros(2,n);     // Ensemble des points
for i=2:n           // Premier point O(0,0)
    t=floor(8*rand(1)+1);
    select t
    case 1 then
        W(:,i) = r*W(:,i-1);
    case 2 then
        W(:,i) = r*W(:,i-1) + D;
    case 3 then
        W(:,i) = r*W(:,i-1) + 2*D;
    case 4 then
        W(:,i) = r*W(:,i-1) + H;
    case 5 then
        W(:,i) = r*W(:,i-1) + 2*D + H;
    case 6 then
        W(:,i) = r*W(:,i-1) + 2*H;
    case 7 then
        W(:,i) = r*W(:,i-1) + 2*H + D;
    case 8 then
        W(:,i) = r*W(:,i-1) + 2*D + 2*H;
    end
end

clf;
replot([0 0 1 1]);
set(gca(), "isoview", "on");
//title(sprintf('$$\\huge %d Points$$',n));
plot(W(1,:),W(2,:),".",'markersize',1)
```

Code Source II.8 – Tapis de Sierpinski itératif

On obtient les résultats suivants :



(a) Version récursive de profondeur $n = 5$



(b) Version itérative avec $n = 100000$ points

FIGURE II.7 – Tapis de Sierpinski

ii) Dimension

La dimension d du tapis de Sierpinski est :

$$\left.\begin{array}{l} r = \dfrac{1}{3} \\ N = 8 \end{array}\right\} \quad 2 = 3^d \Longrightarrow d = \frac{\ln 8}{\ln 3}$$

7) Éponge de Menger

L'éponge de Menger correspond à l'application du tapis de Sierpinski en 3D. On a cette fois 20 sous-cubes 3 fois plus petits que le précédent.

J'ai d'abord crée la fonction `plotCube` qui dessine un cube de coté `r` et d'origine `origin`.

```
function plotCube(origin, r)
    ox = origin(1)
    oy = origin(2)
    oz = origin(3)

    // Côtés
    z = [oz, oz; oz+r, oz+r];
    x = [ox, ox]
    y = [oy, oy+r]
    plot3d(x,y,z)
    x = [ox+r, ox+r]
    y = [oy, oy+r]
    plot3d(x,y,z)

    z = [oz, oz+r; oz, oz+r];
    x = [ox, ox+r]
    y = [oy+r, oy+r]
    plot3d(x,y,z)
    x = [ox, ox+r]
    y = [oy, oy]
    plot3d(x,y,z)

    // Base et Plafond
    x = [ox, ox+r]
    y = [oy, oy+r]
    z = [oz, oz; oz oz];
    plot3d(x,y,z)
    z = [oz+r, oz+r; oz+r oz+r];
    plot3d(x,y,z)

endfunction
```

Code Source II.9 – Fonction pour dessiner un cube

Ensuite j'ai fait le programme suivant :

```
clear;
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD2-Fractales\Plot_Cube.sce");

function Menger(origin, k,n)
    if k > 0 then    // On continue la récursivité avec les 8 sous-cubes

        // Vecteurs de décalage
        r = 1/(3^(n-k+1));
        right    = [r, 0, 0];
        back     = [0, r, 0];
        up       = [0, 0, r];
        clear r;

        Menger(origin, k-1, n);
        Menger(origin + right, k-1, n);
        Menger(origin + 2*right, k-1, n);
        Menger(origin + back, k-1, n);
        Menger(origin + back + 2*right, k-1, n);
        Menger(origin + 2*back, k-1, n);
        Menger(origin + 2*back + right, k-1, n);
        Menger(origin + 2*back + 2*right, k-1, n);

        Menger(origin + up, k-1, n);
        Menger(origin + up + 2*right, k-1, n);
        Menger(origin + up + 2*back, k-1, n);
        Menger(origin + up + 2*back + 2*right, k-1, n);

        Menger(origin + 2*up, k-1, n);
        Menger(origin + 2*up + right, k-1, n);
        Menger(origin + 2*up + 2*right, k-1, n);
        Menger(origin + 2*up + back, k-1, n);
        Menger(origin + 2*up + back + 2*right, k-1, n);
        Menger(origin + 2*up + 2*back, k-1, n);
        Menger(origin + 2*up + 2*back + right, k-1, n);
        Menger(origin + 2*up + 2*back + 2*right, k-1, n);

        clear origin right up back;
    else            // k = 0 => On affiche le sous-cube
        r = 1/(3^n);
        plotCube(origin, r)
    end
endfunction

origin = [0, 0, 0];
n = 3;
clf;
set(gca(), 'isoview', 'on', 'auto_scale', 'off', 'data_bounds', [0,0,0; 1,1,1]);
Menger(origin, n, n);
```

Code Source II.10 – Éponge de Menger

Le code II.10 ne peut que très légèrement être modifié. Si on veut améliorer ses performances, il faut donc améliorer la fonction `plotCube`. Cependant le code II.9 n'est pas très optimisé, il doit y avoir d'autres façons plus efficaces pour dessiner un cube simplement. Pour une profondeur de $n = 2$ le programme s'arrête en moins d'une minute mais pour $n = 3$ la mémoire est très vite saturée.

Néanmoins j'ai pu obtenir le résultat suivant :

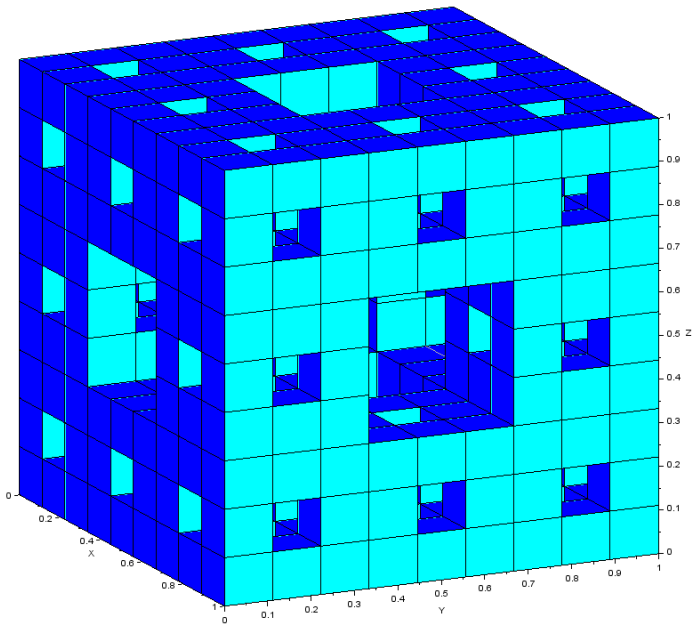


FIGURE II.8 – Éponge de Menger de profondeur $n = 2$

8) Fougère de Barnsley

La fougère de Barnsley est l'une des premières fractales créées itérativement avec un ISF aléatoire. On a le programme Scilab suivant :

```
function Y = T1(X)    // Tige
    Y = [0, 0; 0, 0.16]*X;
endfunction
function Y = T2(X)    // Sous-feuilles
    Y = [0.85, 0.04; -0.04, 0.85]*X + [0; 1.6];
endfunction
function Y = T3(X)    // Feuilles de gauche
    Y = [0.2, -0.26; 0.23, 0.22]*X + [0; 1.6];
endfunction
function Y = T4(X)    // Feuilles de droite
    Y = [-0.15, 0.28; 0.26, 0.24]*X + [0; 0.44];
endfunction

// Probabilité de choix des fonctions
p = [0.01, 0.85, 0.07, 0.07];

n = 100000;
W = zeros(2, n);

for i = 1:n-1
    t = rand();
    if (t < p(1))
        W(:,i+1) = T1( W(:,i) );
    elseif (t < p(1)+p(2))
        W(:,i+1) = T2( W(:,i) );
    elseif (t < p(1)+p(2)+p(3))
        W(:,i+1) = T3( W(:,i) );
    else
        W(:,i+1) = T4( W(:,i) );
    end
end

clf;
set(gca(),"isoview","on");
plot(W(1,:),W(2,:),"g.",'markersize',1);
```

Code Source II.11 – Fougère de Barnsley

Ici les 4 transformations ne sont pas équiprobables. On obtient :

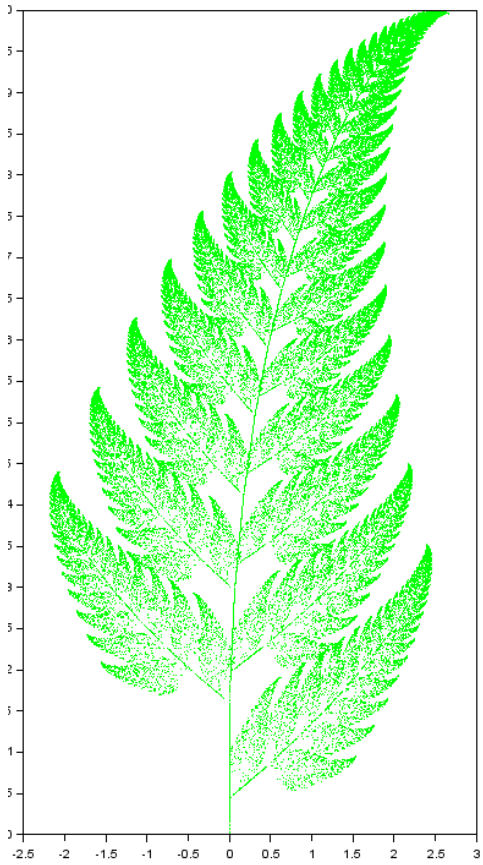


FIGURE II.9 – Fougère de Barnsley avec $n = 100000$ points

CHAPITRE III

Equations différentielles

Dans ce chapitre, nous chercherons à rédoudre des équations différentielles que

$$\begin{cases} y'(t) = f(t, y(t)) \\ y(t_0) = y_0 \end{cases} \tag{III.1}$$

avec $t \in I = [t_0; t_0 + T]$ $(T > 0)$ et $f(t, y)$: $I \times \mathbb{R}^n \to \mathbb{R}^n$. (III.1) est un problème de Cauchy.

Dans le cas des équations différentielles d'ordre $n > 1$, nous allons ramener le problème a un équation différentielle d'ordre 1 dans $\mathbb{R}^n$. On a le système différentiel d'ordre n :

$$\begin{cases} y(0) = a_0 \\ y'(0) = a_1 \\ \quad \vdots \\ y^{(n-1)}(0) = a_{n-1} \\ y^{(n)} = f(t, y, y', ..., y^{(n-1)}) \end{cases} \tag{III.2}$$

On pose alors $Y \in \mathbb{R}^n$ tel que : $\begin{cases} Y' = F(t, Y) \\ Y(0) \end{cases}$ avec :

$$Y = \begin{pmatrix} y_1 = y(t) \\ y_2 = y_1' = y' \\ \vdots \\ y_n = y_{n-1}' = y^{(n-1)} \end{pmatrix} \quad , \quad Y(0) = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} \quad \text{et} \quad F(t, Y) = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ f(t, y1, y2, ..., y_{n-1}) \end{pmatrix}$$

EXEMPLE : Vectorisation de systèmes différentielles

$$\begin{cases} y^{(3)} + ay' + cy = 0 \\ y(0) = d_0 \\ y'(0) = d_1 \\ y''(0) = d_2 \end{cases} \tag{III.3}$$

On vectorise l'équation :

$$Y = \begin{pmatrix} y_1 = y \\ y_2 = y_1' = y' \\ y_3 = y_2' = y'' \end{pmatrix} \iff \begin{cases} y_1' = y_2 \\ y_2' = y_3 \\ y_3' = -ay_2 - cy_1 \end{cases} \iff Y' = f(t, Y) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -c & -a & 0 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Que ce passe-t-il si l'on n'est pas capable de résoudre analytiquement une équation différentielle ?

EXEMPLE :

$$\begin{cases} y'(t) = e^{-t^2} \\ y(0) = 1 \end{cases} \tag{III.4}$$

$$\begin{aligned} \frac{dy}{dt} = e^{-t^2} &\implies \frac{dy}{y} = e^{-t^2} \\ &\implies \int \frac{dy}{y} = \int e^{-t^2}\,dt \\ &\implies \ln|y| = \int e^{-t^2}\,dt \end{aligned}$$

Arrivé ici, on est bloqué car on ne connaît pas de primitive de e^{-t^2} donc l'équation différentielle (III.4) n'admet pas de solution analytique. On a donc recours à un schéma numérique.

I) Schémas numériques

Les algorithmes que nous présenterons ici ont pour but d'approcher la solution d'une équation numérique.

On discrétise l'intervalle $I = [t_0; t_0 + T]$ en $\sigma = \{t_0, t_1, ..., t_N\}$ avec $t_N = t_0 + T$. Pour simplifier, on supposera des subdivisions uniformes et donc le pas est : $h = h_i = |t_{i+1} - t_i| \ \ \forall i \in [0; N-1]$.

On approche $y(t_i)$ par z_i où :

$$\begin{cases} z_{i+1} = z_i + h\Phi(t_i, z_i, h_i) \\ z_0 = y_0 \end{cases} \tag{III.5}$$

Avec Φ la fonction d'approximation spécifique à chaque schéma.

1) Caractéristiques d'un schéma numérique

a) Stabilité

Soient les suites $(u_i)_{i\in\mathbb{N}}$ et $(v_i)_{i\in\mathbb{N}}$ telles que :

$$\begin{cases} u_{i+1} = u_i + h\Phi(t_i, u_i, h) \\ u_0 \text{ donné} \end{cases} \qquad \begin{cases} v_{i+1} = u_{i+1} + \epsilon_i \\ v_0 = u_0 \end{cases} \tag{III.6}$$

(v_i) correspond au même schéma que (u_i) mais avec une erreur $\epsilon_i \in \mathbb{R}$.

DÉFINITION III.1 : Stabilité

Le schéma est dit stable si

$$\max_{i\in I}|u_i - v_i| \leq C\left(|u_0 - v_0| + \sum_{i=1}^{N}|\epsilon_i|\right) \tag{III.7}$$

où C est une constante ne dépendant pas de u_i et v_i.

THÉORÈME III.1 :

Le schéma est stable si et seulement si $\exists K$ tel que :

$$\|\Phi(t, y, h) - \Phi(t, z, h)\| \leq K\|y - z\| \tag{III.8}$$

c'est-à-dire que Φ est $K-$lipschitzienne par rapport à sa 2e variable.

Le fait qu'un schéma soit stable signifie que la propagation d'erreur sur la donnée initiale n'influe pas le résultat final. Si Φ n'est pas lipschitzienne, l'unicité de la solution n'est pas garantie.

b) Consistance

DÉFINITION III.2 : Consistance

Un schéma est dit consistant si

$$\epsilon(y) = \sum\|y_{i+1} - y_i - \phi t_i, y_i, h\| \xrightarrow[h\to 0]{} 0 \tag{III.9}$$

$$\Phi(t, y, 0) = f(t, y) \tag{III.10}$$

THÉORÈME III.2 :

Pour montrer que le schéma est consistant, il suffit de montrer qu'il existe une constante $C > 0$ telle que :

$$\|\Phi(t, y, 0) - \Phi(t, z, 0)\| \leq C\|y - z\| \tag{III.11}$$

c) Ordre de convergence

DÉFINITION III.3 : Ordre de convergence
Un schéma est d'ordre p si $\exists C > 0$ tel que :

$$\epsilon(y) = \max|y_i - z_i| \leq Ch^p \tag{III.12}$$

2) Schéma d'Euler

On a $y' = f(t, y)$. On connaît y_0, comment calculer $y_1 = y(t_1)$? On fait un développement de Taylor :

$$y_{i+1} = y(t_{i+1}) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(\epsilon)$$

On se débarasse du reste qui tend vers 0 quand h tend vers 0 :

$$y_{i+1} = y(t_{i+1}) \sim y_i + hy'(t_i) = y_i + hf(t_i, y_i) = z_{i+1}$$

On approche donc y_i par z_i, on obtient le schéma d'Euler :

$$\begin{cases} z_{i+1} = z_i + hf(t_i, z_i) \\ z_0 = y_0 \end{cases} \tag{III.13}$$

Et donc

$$\Phi(t_i, z_i, h) = f(t_i, z_i) \tag{III.14}$$

On peut aussi définir ce schéma sans développement de Taylor :

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} y'(t)dt = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y)dt \tag{III.15}$$

Or $\int_{t_i}^{t_{i+1}} f(t, y)dt$ est inconnu, on l'approxime donc par l'aire du rectangle gauche :

$$y(t_{i+1}) \sim y(t_i) + h \times f(t_i, y_i) \tag{III.16}$$

D'où le schéma d'Euler explicite (III.13) : $z_{i+1} = z_i + hf(t_i, z_i)$.

Si on approche par l'aire du rectangle droit, on a le schéma d'Euler implicite (moins utilisé) : $z_{i+1} = z_i + hf(t_{i+1}, z_{i+1})$.

Propriété III.1 : Le schéma d'Euler est stable et consistant.

Preuve :
(III.14) $\implies \|f(t, y) - f(t, z)\| \leq K\|y - z\|$ si f est lipschitzienne, alors le schéma est stable De plus par définition (III.14) le schéma est consistant.

Propriété III.2 : Le schéma d'Euler est d'ordre 1.

Preuve :

$$y_{i+1} = y(t_i) + hy'_i + \frac{h^2}{2}y''(\epsilon_i)$$
$$\text{et } z_{i+1} = z_i + hf(t_i, z_i)$$

$$\text{D'où } \|y_{i+1} - z_{i+1}\| = (y_i - z_i) + h\big(f(t_i, y_i) - f(t_i, z_i)\big) + \frac{h^2}{2}y''(\epsilon_i)$$

On suppose f L-lipschitzienne uniforme par rapport à sa seconde variable. De plus, supposons : $\exists M > 0$ tel que $\|y''(\epsilon_i)\| \leq M \ \ \forall \epsilon \in [t_0, t_0 + T]$.

On note $e_i = y_i - z_i$, on a alors :

$$\begin{aligned}\|e_{i+1}\| &\leq \|e_i\| + hK\|e_i\| + \frac{h^2}{2}M \\ &\leq \underbrace{(1+hL)}_{C}\|e_i\| + \underbrace{\frac{h^2}{2}M}_{D} \\ &\leq C\|e_i\| + D\end{aligned}$$

$(e_i)_{i\in\mathbb{N}}$ est une suite arithmético-géométrique, on a alors :

$$\begin{aligned}\|e_{i+1}\| &\leq C^{i+1}e_0 + D \times \sum_{k=1}^{i} C^i \\ &\leq C^{i+1}e_0 + D \times \frac{C^{i+1}-1}{C-1}\end{aligned}$$

Or $z_0 = y_0 \implies e_0 = 0$. De plus :

$$\begin{aligned}D \times \frac{C^{i+1}-1}{C-1} &= \frac{h^2}{2}M\frac{(1+hL)^{i+1}-1}{hL} \\ &= \frac{hM}{2L}\left((1+hL)^{i+1}-1\right)\end{aligned}$$

On utilise $(1+x)^k \leq e^{kx} \ \ \forall k, x > 0$. On a finalement :

$$\begin{aligned}\|e_{i+1}\| \leq \frac{hM}{2L}(e^{iKH}-1) &= \frac{hM}{2L}(e^{\overbrace{ih}^{t_i - t_0}}-1) \\ &\leq h\underbrace{\left(\frac{M}{2L}(e^T-1)\right)}_{K\in\mathbb{R}} \\ &\leq Kh\end{aligned}$$

3) Schéma du point-milieu

On approxime (III.15) par l'aire du rectangle du point du milieu :

$$\begin{aligned}\text{(III.15)} \iff y(t_{i+1}) &\sim y(t_i) + h \times f(t_i + \frac{h}{2}, y(t_i + \frac{h}{2})) \\ &\sim y(t_i) + h \times f(t_i + \frac{h}{2}, y_i + \frac{h}{2}f(t_i, y_i))\end{aligned}$$

D'où le schéma du point milieu :

$$\begin{cases} z_{i+1} = z_i + h \times f(t_i + \frac{h}{2}, z_i + \frac{h}{2}f(t_i, z_i)) \\ z_0 = y_0 \end{cases} \tag{III.17}$$

Et donc $\Phi(t_i, z_i, h) = f(t_i + \frac{h}{2}, y_i + \frac{h}{2}f(t_i, y_i))$.

Propriété III.3 : Le schéma du point milieu est d'ordre 1.

Preuve :
On part des suites $(u_i)_{i\in\mathbb{N}}$ et $(u_i)_{i\in\mathbb{N}}$ définies en (III.6)

$$\begin{aligned}u_{i+1} - v_{i+1} &= [u_i + h\Phi(t_i, u_i, h)] - [v_i + h\Phi(t_i, v_i, h)] \\ \|u_{i+1} - v_{i+1}\| &\leq [u_i + h\Phi(t_i, u_i, h)] - [v_i + h\Phi(t_i, v_i, h)] \quad \text{(inégalité triangulaire)} \\ &\leq \|u_i - v_i\| + h\|\Phi(t_i, u_i, h) - \Phi(t_i, v_i, h)\| + \|\epsilon_i\| \\ &\leq \|u_i - v_i\|(1 + Ch) + \|\epsilon_i\|\end{aligned}$$

On utilise le lemme de Granwall : Si $|u_{i+1} - v_{i+1}| \leq C|u_i - v_i| + |\epsilon_i|$ alors $\exists K$ tel que :

$$|u_{i+1} - v_{i+1}| \leq K\left(|u_0 - v_0| + \sum_{k=1}^{i}|\epsilon_i|\right) \tag{III.18}$$

Preuve :
Par récurrence : Au rang $i = 0$, c'est trivialement vrai.
On suppose $|u_{i+1} - v_{i+1}| \leq C\left(|u_0 - v_0| + \sum_{k=1}^{i} |\epsilon_i|\right)$ vrai pour un certain rang i. Par hypothèse, on a :

$$\begin{aligned}
|u_{i+1} - v_{i+1}| &\leq C|u_i - v_i| + |\epsilon_i| \\
&\leq C\left(C(|u_0 - v_0| + \sum_{k=1}^{i-1} |\epsilon_i|)\right) + |\epsilon_i| \quad \text{(hypothèse de récurrence)} \\
&\leq \underbrace{\max(C^2, C|\epsilon_i|)}_{K}\left(|u_0 - v_0| + \sum_{k=1}^{i} |\epsilon_i|\right)
\end{aligned}$$

Propriété III.4 : Le schéma du point milieu est stable et consistant.

4) Schéma d'Euler-Cauchy

On approxime (III.15) par l'aire du trapèze :

$$\begin{aligned}
\text{(III.15)} \iff y(t_{i+1}) &\sim y(t_i) + \frac{h}{2} \times \big(f(t_i, y_i) + f(t_{i+1}, y_{i+1})\big) \\
&\sim y(t_i) + \frac{h}{2} \times \big(f(t_i, y_i) + f(t_{i+1}, y_i + hf(t_i, y_i))\big)
\end{aligned}$$

D'où le schéma d'Euler-Cauchy :

$$\begin{cases} z_{i+1} = z_i + \frac{h}{2} \times \big(f(t_i, z_i) + f(t_{i+1}, z_i + hf(t_i, z_i))\big) \\ z_0 = y_0 \end{cases} \tag{III.19}$$

Et donc $\Phi(t_i, z_i, h) = \frac{1}{2} \times \big(f(t_i, z_i) + f(t_{i+1}, z_i + hf(t_i, z_i))\big)$.

Propriété III.5 : Le schéma d'Euler-Cauchy est stable et consistant.

Propriété III.6 : Le schéma d'Euler-Cauchy est d'ordre 2.

5) Schéma de Runge-Kutta

Enfin nous avons le schéma de Runge-Kutta :

$$\begin{cases} k_1 = f(t_i, z_i) \\ k_2 = f(t_i + \frac{h}{2}, z_i + \frac{h}{2} k_1) \\ k_3 = f(t_i + \frac{h}{2}, z_i + \frac{h}{2} k_2) \\ k_4 = f(t_i + h, z_i + h k_3) \\ z_{i+1} = z_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{cases} \tag{III.20}$$

Propriété III.7 : Le schéma de Runge-Kutta est stable et consistant.

Propriété III.8 : Le schéma de Runge-Kutta est d'ordre 4.

II) Applications

1) Comparaison des schémas

Nous implémentons les schémas sous Scilab de la manière suivante :

```
// Paramètres des schémas :
//        y0 vecteur colonne ou réel correspondant à la situation initiale
//        t vecteur ligne de discretisation de l'intervalle [t0 à t0+T]
//        y = f(t,x) l'équation à résoudre soit forme de vecteur colonne ou réel
//
// Retourne
//        y matrice où chaque colonne correspond à une itération correspondant à la fonction y approchée
 ↪

// Schéma de Euler
function y = Euler(y0, t, f)
    n = length(t);
    h = t(2)-t(1);
    y(:,1) = y0;
    for i=1:n-1
        y(:,i+1) = y(:,i) + h.*f(t(i), y(:,i));
    end
endfunction

// Schéma de Euler-Cauchy
function y = EulerCauchy(y0, t, f)
    n = length(t);
    h = t(2)-t(1);
    y(:,1) = y0;

    for i=1:n-1
        k1 = f(t(i),      y(:,i));
        k2 = f(t(i) + h, y(:,i) + h*k1);
        y(:,i+1) = y(:,i) + h*(k1 + k2)/2;
    end
endfunction

// Schéma du Point Milieu
function y = PointMilieu(y0, t, f)
    n = length(t);
    h = t(2)-t(1);
    y(:,1) = y0;

    for i=1:n-1
        k1 = f(t(i),     y(:,i));
        k2 = f(t(i)+h/2, y(:,i) + (h*k1)/2);
        y(:,i+1) = y(:,i) + h*k2;
    end
endfunction

// Schéma de Runge-Kutta
function y = RungeKutta(y0, t, f)
    n = length(t);
    h = t(2)-t(1);
    y(:,1) = y0;

    for i=1:n-1
        k1 = f( t(i),     y(:,i));
        k2 = f( t(i)+h/2, y(:,i) + (h*k1)/2);
        k3 = f( t(i)+h/2, y(:,i) + (h*k2)/2);
        k4 = f( t(i)+h,   y(:,i) + h*k3);
        y(:,i+1) = y(:,i) + h*(k1 + 2*k2 + 2*k3 + k4)/6;
    end
endfunction
```

Code Source III.1 – Schémas de résolution d'équations différentielles

Les paramètres de ces schémas sont les suivants :

- `y0` $\in \mathcal{M}_{1k}$ vecteur colonne ou réel correspondant à la situation initiale (k est l'ordre de l'équation différentielle)
- `t` $\in \mathcal{M}_{n1}$ discrétisation de l'intervalle $[t_0; t_0 + T]$
- `f(t,x)` correspond à l'équation différentielle à résoudre retourne $f(t,x) \in \mathcal{M}_{1k}$

En retour, on récupère la matrice `y` $\in \mathcal{M}_{kn}$ où chaque ligne correspond à une itération correspondant à la fonction $y(t)$ approchée à l'instant t_i.

On va comparer tous ces schémas sur l'équation différentielle suivante :

$$\begin{cases} y' = -t \times y(t) + t \quad t \in [0;4] \\ y(0) = y_0 = 0 \end{cases} \tag{III.21}$$

dont la solution analytique est $y(t) = 1 - e^{-\frac{t^2}{2}}$.

On fait les comparaisons pour plusieurs discrétisations avec n le nombre de subdivisions de l'intervalle $[0;4]$. Pour chaque valeur de n on calcule l'erreur logarithmique absolue maximale de chaque schéma par rapport à la solution analytique. Ensuite on effectue une régression linéaire des erreurs afin de récupérer l'ordre des schémas.

On obtient les résultats suivants :

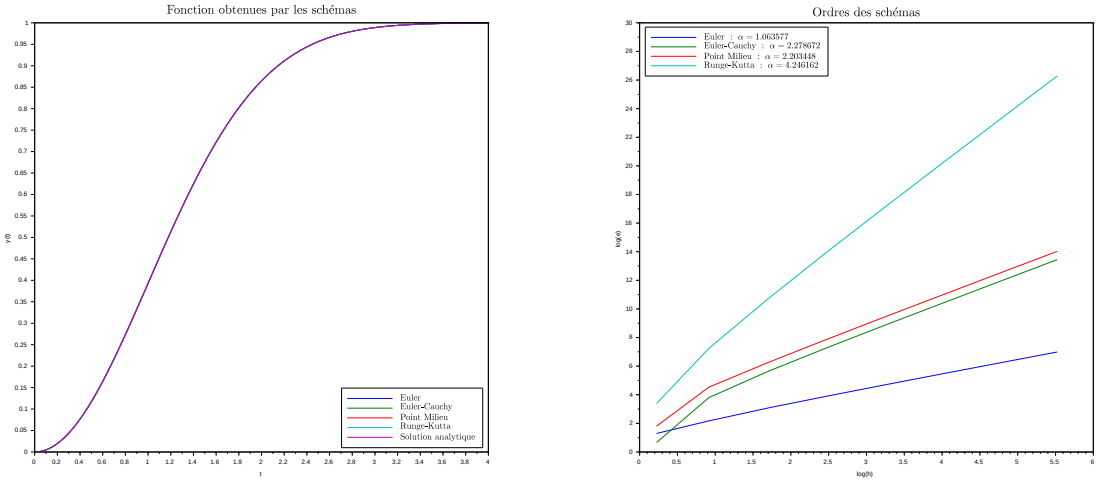


FIGURE III.1 – Comparaison des schémas

Sur le graphique de la fonction, on ne distingue pas les approximations de chaque schémas car elles sont confondues. Concernant les ordres des méthodes on obtient bien les mêmes que ceux prévues précédemment.

2) Simulation d'un pendule

On modélise ici un pendule de masse M accroché à une tige de longueur L de masse négligeable devant M. On note $\theta(t)$ l'angle formé par la tige et l'axe vertical passant par la fixation du pendule. On a l'équation différentielle :

$$\begin{cases} \theta''(t) = -\frac{g}{L} \times \sin(\theta(t)) \\ \theta'(0) = v_0 \\ \theta(0) = \theta_0 \end{cases} \tag{III.22}$$

Comme l'équation est non-linéaire à cause du $\sin(\theta)$, on ne peut pas trouver de solution analytique. En revanche, on peut approcher $\sin(\theta)$ par θ dans l'hypothèse où les angles sont petits (inférieurs à 5°). Cette approximation permet alors d'avoir l'équation différentielle linéaire suivante :

$$\begin{cases} \phi''(t) = -\frac{g}{L} \times \phi(t) \\ \phi'(0) = v_0 \\ \phi(0) = \theta_0 \end{cases} \tag{III.23}$$

où $\phi(t)$ correspond à $\theta(t)$ avec l'approximation. La solution analytique est alors :

$$\phi(t) = \theta_0 \cos\left(t\sqrt{\frac{g}{L}}\right) \tag{III.24}$$

On va alors comparer les deux modèles (III.22) et (III.23) avec différents angles θ_0. Pour l'implémentation sous Scilab on pose : $y = \begin{pmatrix} \theta \\ \theta' \end{pmatrix}$. On a :

```
clear;
function dydt = f(t,y)
    g = 9.8;
    L = 10;

    theta = y(1);
    thetaprime = y(2);
    dydt = [ thetaprime ; -g/L * sin(theta) ];
endfunction
function dydt = approx(t,y)
    g = 9.8;
    L = 10;

    theta = y(1);
    thetaprime = y(2);
    dydt = [ thetaprime ; -g/L * theta ];
endfunction
exec("D:\Documents\Cours\TC04 - Printemps
 ↪  2017\MT94\Scilab\TD3-Equations_Differentielles\0-Tous_Schemas.sce");

t0 = 0; T  = 40;
n = 40;                          // Nombre de points
h = T/n;
t = linspace(t0,t0+T,n);
c = [0, 1];                      // Centre et longueur
l = 1;

// Conditions initiales (i,j,k) : angle = i*PI/j, vitesse = k
thNum = [ 1, 64,0;
          1, 16,0;
          1, 4, 0;
          2, 3, 0 ];
thetas = [ thNum(:,1)*%pi./thNum(:,2), thNum(:,3) ];

scf(0); clf;
scf(1); clf;
for k=1:length(thetas)/2

    y0 = thetas(k,:)';

    Original = RungeKutta(y0,t, f)';
    Approximation = RungeKutta(y0,t, approx)';

    scf(0);    subplot(2,2,k);
    xtitle("$\huge\text{Comparaison des méthodes pour } \theta_0 = \frac{" + string(thNum(k,1)) +
     ↪  "\pi}{" + string(thNum(k,2)) + "} \text{ et } v_0 =" + string(thNum(k,3)) + "$", "t", "$ \theta
     ↪  $");
    plot(t',[Original(:,1), Approximation(:,1)]);

    scf(1); subplot(2,2,k);
    set(gca(),'isoview','on', 'data_bounds', [-1.5, 1.5, -0.1, 1.5]);
    plot(c(1), c(2), 'kx');
    z = [Original(1:20,1) Approximation(1:20,1)];
    x = c(1) + l*sin(z);
    y = c(2) - l*cos(z);
    plot(x,y,'.-');
    title("$\huge\text{Pendule pour } \theta_0 = \frac{" + string(thNum(k,1)) + "\pi}{" +
     ↪  string(thNum(k,2)) + "} \text{ et } v_0 = " + string(thNum(k,3)) + "$");
end

scf(0); legend("Problème original", "Approximation", 4);
scf(1); legend("Centre", "Problème original", "Approximation", 4);

// TODO Erreur cumulée
```

Code Source III.2 – Modélisation de pendules

Les différentes conditions intiales sont répertoriées dans la matrice `thetas`. Pour chaque expérience, on résout le problème original (III.22) et celui après approximation (III.23) avec la méthode de Runge-Kutta. On obtient les résultats suivant

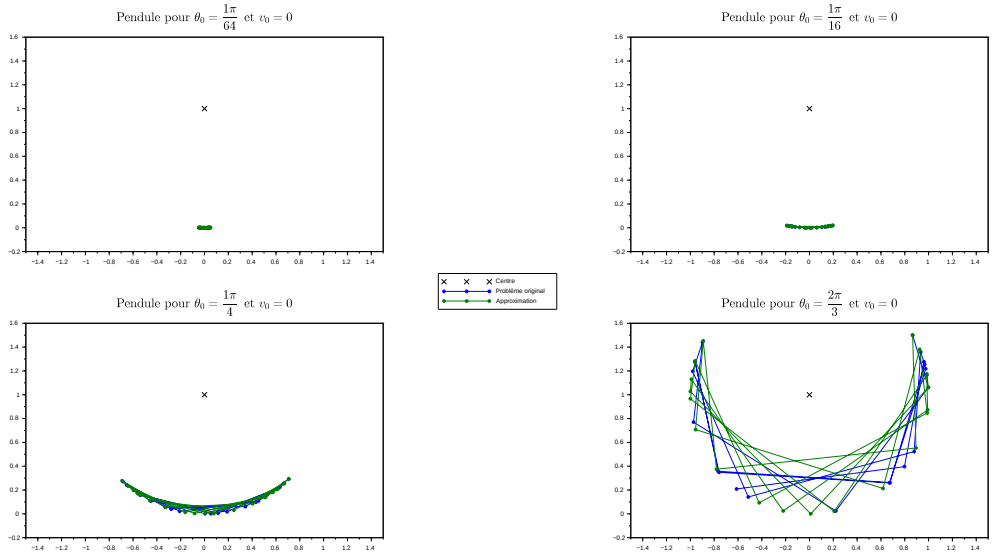


FIGURE III.2 – Pendules des différentes expériences

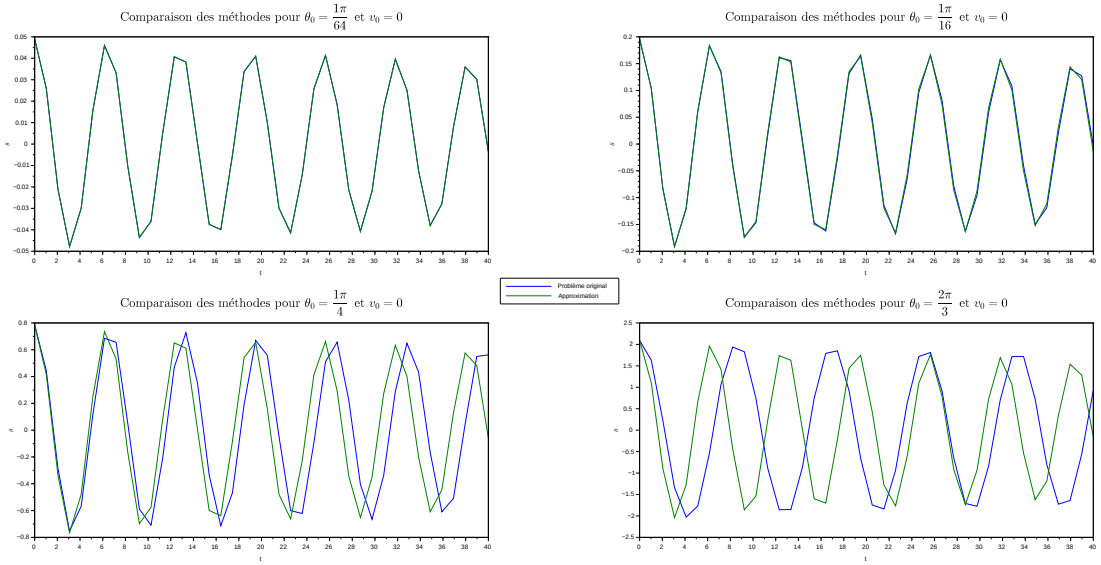


FIGURE III.3 – Comparaison des différentes expériences

On observe que pour les petits angles $\theta_0 = \frac{\pi}{64} = 2.8125°$ et même $\frac{\pi}{16} = 11.25°$ l'approxmation diffère très peu du problème original. En revanche pour des angles plus grand, on voit clairement une différence de période d'oscillation et donc un écart se creusant entre les deux solutions. Ainsi on a bien vérifié expérimentalement que l'hypothèse $sin(\theta) \sim \theta$ n'est valable que pour de petits angles.

3) Gravitation

La mécanique céleste est une affaire d'équations différentielles. Ainsi nous pouvons simuler le mouvement de plusieurs corps céleste soumis à leur attractions gravitationnelles mutuelles. Dans le cas de deux corps, le problème peut être résolu analytiquement mais pour plus de deux corps cela n'est pas possible. On pourrait tenter de contourner ce problème en le ramenant à plusieurs sous-problèmes à deux corps mais la solution ainsi obtenue ne correspond pas aux trajectoires observées. Il y a entre chaque corps une influence suffisamment importante pour modifier les trajectoires à long terme. Dans le cas de plusieurs corps, nous devons recourir aux mêmes schémas numériques qui nous ont permis de résoudre le problème du pendule (III.22).

Nous allons ici modéliser le mouvement de deux corps d'abord, puis de trois. On considère deux corps sphérique C_1 et C_2 de masses m_1 et m_2 et de centres :

$$u_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \quad \text{et} \quad u_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \tag{III.25}$$

La force gravitationnelle exercée par le corps 2 sur le corps 1 est :

$$F_{2\to 1} = G \times \frac{m_1 m_2}{\|u_2 - u_1\|^3}(u_2 - u_1) \tag{III.26}$$

avec $G = 6.67 \times 10^{-11}$ la constante de gravitation universelle. La force exercée par le C_1 sur C_2 est égale à son opposée : $F_{1\to 2} = -F_{2\to 1}$.

Avec la seconde loi de Newton, nous obtenons le système dynamique suivant :

$$\begin{cases} u_1'' = +Gm_2 \frac{u_2 - u_1}{\|u_2 - u_1\|^3} \\ u_2'' = +Gm_1 \frac{u_2 - u_1}{\|u_2 - u_1\|^3} \end{cases} \tag{III.27}$$

On considère ce problème avec la Terre pour C_1 et la Lune pour C_2. La distance moyenne Terre-Lune est de $d_{TL} = 3.84402 \times 10^8$ m et la période de rotation d'environ $T = 27,55$ jours. Nous centrons le systèmes sur la Terre, nous avons les conditions intiales suivantes :

$$C_1 \begin{cases} m_1 = 5.975 \times 10^{24} \\ u_1(0) = (0;0) \\ u_1'(0) = (0;0) \end{cases} \qquad C_2 \begin{cases} m_2 = 7.35 \times 10^{22} \\ u_2(0) = (d_{TL};0) \\ u_2'(0) = (0; \frac{2\pi}{T} d_{TL}) \end{cases} \tag{III.28}$$

Pour pouvoir utiliser les schémas numériques, il faut vectoriser les équations sous forme du premier ordre. On a donc $v \in \mathbb{R}^8$ tel que le système d'équations différentielles à résoudre correspond à $v' = f(v)$.

$$v = \begin{pmatrix} u_1 \\ u_1' \\ u_2 \\ u_2' \end{pmatrix} \tag{III.29}$$

On a l'implémentation sous Scilab suivante :

```
clear;
function dvdt = gravit(t,v)
    m1 = 5.975*10^24;
    m2 = 7.35*10^22;
    G = 6.67*10^(-11);
    // v = u1 u1' u2 u2'
    // u(x y)
    u1 = [v(1); v(2)];
    u1prim = [v(3); v(4)];
    u2 = [v(5); v(6)];
    u2prim = [v(7); v(8)];

    dvdt = [
         u1prim,
         G*m2*(u2-u1)/(norm(u2-u1)^3);
         u2prim;
         -G*m1*(u2-u1)/(norm(u2-u1)^3);
     ];
endfunction

m1 = 5.975e24;
m2 = 7.35e22;
d = 3.84402*10^8;
T = 27.55*24*60*60;
t0 = 0;
t = linspace(0,2*T,1000);

u1zero = [0; 0];
u2zero = [d; 0];
u1primzero = [0; 0];
u2primzero = [0; 2*%pi*d/T];

v0 = [u1zero; u1primzero; u2zero; u2primzero];
v = ode(v0,t0,t,gravit)

scf(0); clf;
Gx = ((m1*v(1,:) + m2*v(5,:))/(m1+m2));
Gy = ((m1*v(2,:) + m2*v(6,:))/(m1+m2));

plot(v(1,:)-Gx, v(2,:)-Gy, 'g',  v(5,:)-Gx, v(6,:)-Gy, 'b');
set(gca(),"isoview","on");

// Animation
scf(1); clf;
x = v([1 5],:);
y = v([2 6],:);
plot(x(1,1), y(1,1),'.',  x(2,1),y(2,1),'.');
h = gce();
set(gca(), 'isoview', 'on', 'data_bounds', [min(x) max(x) min(y) max(y)]);
for n = 2:length(t)
    h.children(1).data = [x(1,n) y(1,n)];
    h.children(2).data = [x(2,n) y(2,n)];
end
```

Code Source III.3 – Gravitation Terre-Lune

On représente la trajectoire de la Lune (en bleu) et de la Terre (en bleu) par rapport au centre de gravité du système. On retrouve bien la trajectoire circulaire de la Lune autour de la Terre.

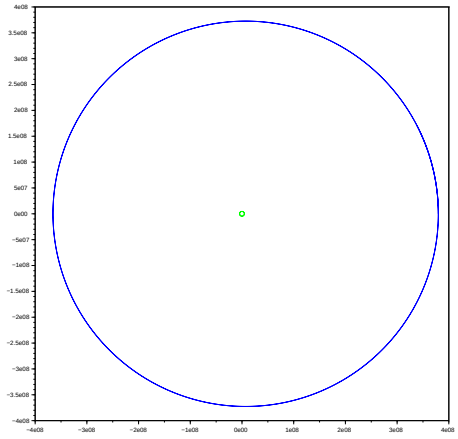


FIGURE III.4 – Gravitation Terre-Lune

4) Attracteur de Lorenz

L'attracteur de Lorenz permet de modéliser de manière simple le caractère chaotique des phénomènes météorologiques. Nous avons le système différentiel suivant :

$$\begin{cases} \frac{\partial x(t)}{\partial t} = \sigma\big(y(t) - x(t)\big) \\ \frac{\partial y(t)}{\partial t} = \rho\, x(t) - y(t) - x(t)\, z(t) \\ \frac{\partial z(t)}{\partial t} = x(t)\, y(t) - \beta\, z(t) \end{cases} \tag{III.30}$$

avec $\sigma = 10$, $\rho = 28$, $\beta = \frac{8}{3}$.

Nous résolvons le système par le programme suivant :

```
clear;
function dudt = fLorenz(t,u)
    sigma = 10;
    rho = 28;
    bet = 8/3;
    x = u(1);
    y = u(2);
    z = u(3);

    dudt = [
        sigma*(y-x);
        rho*x - y - x*z;
        x*y - bet*z;
    ];
endfunction

exec("D:\Documents\Cours\TC04 - Printemps
 ↪  2017\MT94\Scilab\TD3-Equations_Differentielles\0-Tous_Schemas.sce");
NB_POINTS = 50000;
t = linspace(0,500,NB_POINTS);
u0 = [10; 10; 10];
u = RungeKutta(u0, t, fLorenz);
clf;
param3d(u(1,:),u(2,:),u(3,:));
set(gca(), "isoview","on");
```

Code Source III.4 – Attracteur de Lorenz

Affichons les valeurs de x, y, z dans un repère 3D :

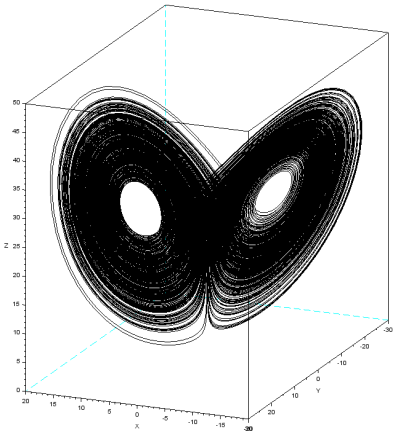


FIGURE III.5 – Attracteur de Lorenz

On observe que la solution forme une figure semblable à des ailes de papillon. Elle a un comportement qui peut sembler périodique mais qui est en réalité chaotique et imprévisible.

De plus, le point initial est vite attiré vers l'attracteur. On peut prendre un point plus éloigné il convergera quand même dans l'attracteur.

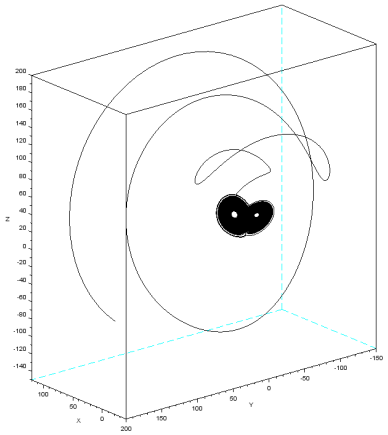


FIGURE III.6 – Attracteur de Lorenz avec une condition initiale éloignée

5) Systèmes Proies-Prédateurs de Lotka

Le modèle proies-prédacteurs de Lotka est un système différentiel modélisant de manière simplifier un écosystème d'une population de proies et d'une autre de ses prédateurs. On a le système suivant :

$$\begin{cases} x' = ax - bxy \\ y' = -cy + dxy \end{cases} \tag{III.31}$$

Avec les différentes variables et paramètres :

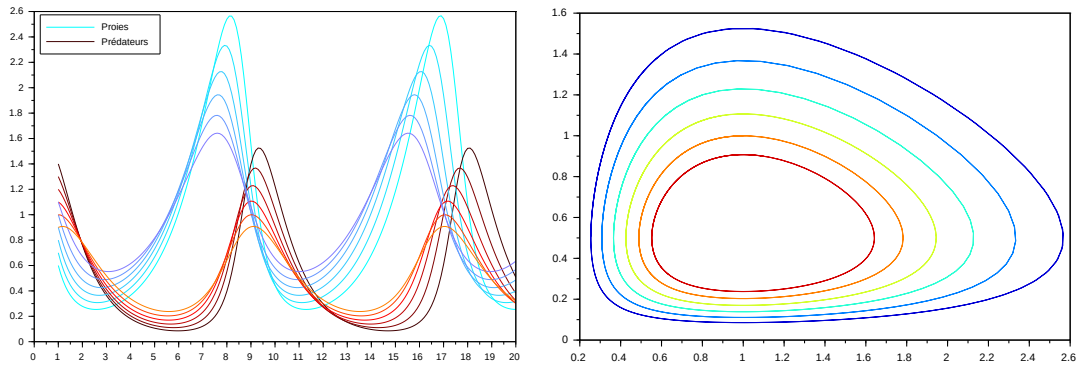
- x la population de proies
- y la population de prédateurs
- a le taux de reproduction des proies,on suppose que les proies ont un accès illimité à la nourriture, elles donc une croissance exponentielle
- c le taux de mortalité des prédateurs
- b et d les intéractions entre les deux populations

On a l'implémentation suivante :

```
clear;
function dYdt = fLotka(t,Y)
    x = Y(1);  // Proies
    y = Y(2);  // Prédacteurs

    a = 2/3;
    b = 4/3;
    c = 1;
    d = 1;

    dYdt = [
        x*(a - b*y);
        y*(c*x - d)
    ];
endfunction

exec("D:\Documents\Cours\TC04 - Printemps
 ↪  2017\MT94\Scilab\TD3-Equations_Differentielles\0-Tous_Schemas.sce");

t = 1:.1:20;
UPTO = 7;
proies = coolcolormap(1.8*UPTO);
predateurs = hotcolormap(1.8*UPTO);
total = jetcolormap(UPTO);

scf(0); clf;
scf(1); clf;
for i=1:UPTO
    y0 = [1+(i-4)/10; 1-(i-4)/10];
    y = RungeKutta(y0, t, fLotka);
    disp(y0)
    scf(0);
    plot(t, y(1,:), 'Foreground', proies(i,:));
    plot(t, y(2,:), 'Foreground', predateurs(i,:));

    scf(1);
    set(gca(), 'isoview','on');
    plot(y(1,:),y(2,:), 'Foreground', total(i,:));
end
xtitle("Cycles", "Proies", "Prédateurs");
scf(0);
legend("Proies","Prédateurs",2)
```

Code Source III.5 – Système de Lotka

On prend différentes proportions de population entre les proies et les prédateurs. On obtient les résultats suivants :



(a) Courbes de Lotka

(b) Cycles de Lotka

FIGURE III.7 – Systèmes proies-prédateurs de Lotka

On remarque que peut importe les populations initiales (strictement positives), l'évolution se fait de manière cyclique. On a d'abord la prolifération des proies, puis des prédateurs qui diminuent la quantité de proies, ayant moins à manger ces derniers diminuent aussi et ainsi de suite.

CHAPITRE IV

Valeurs propres

Dans ce chapitre, nous verrons comment étendre la notion de valeurs propres à des matrices rectangulaire dans la partie I) mais aussi comment calculer un classement des noeuds les plus importants d'un graphe dans la partie IV).

I) Décomposition en Valeurs Singulières (SVD)

1) Méthode SVD

SVD signifie *Singular Value Decomposition*, il s'agit d'une méthode de factorisation de matrices rectangulaires.

On connaît déjà la décomposition de matrices carrées $A \in \mathcal{M}_{nn}$:

— Si A est diagonalisable alors il existe P inversible et D diagonale contenant les valeurs propres de A tels que : $A = PDP^{-1}$.

— De plus si A est symétrique alors P est orthogonale et $A = PDP^{\top}$.

— En revanche si A n'est pas diagonalisable, on peut la trigonaliser avec T une matrice triangulaire telle que $A = PTP^{-1}$.

L'idée de la SVD est de permettre de factoriser les matrices rectangulaires en étendant la notion de valeurs propres.

THÉORÈME IV.1 :
On suppose $m \geq n$. Pour toute matrice $A \in \mathcal{M}_{mn}$, il existe $U \in \mathcal{M}_{mm}$ et $V \in \mathcal{M}_{nn}$ et $\Sigma \in \mathcal{M}_{mn}$ de la forme

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \dots & 0 \\ 0 & \sigma_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & \sigma_n \\ 0 & \dots\dots\dots & & 0 \\ \vdots & \dots\dots\dots & & \vdots \\ 0 & \dots\dots\dots & & 0 \end{pmatrix}$$

telle que les σ_i sont positifs et triés par ordre décroissant : $\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_n \geq 0$. On a alors : $A = U\Sigma V^T$.

Preuve :
AA^T et A^TA sont des matrices symétriques et elles admettent donc les décompositions :

$$A^TA = VD_1V^T \in \mathcal{M}_n$$
$$AA^T = UD_2U^T \in \mathcal{M}_m$$

On a :

$$\begin{aligned} A^TA &= V\Sigma^TU^TU\Sigma V^T \\ &= V\Sigma^T\Sigma V^T \\ &= VD_1^2V^T \end{aligned}$$

Par identification les σ_i^2 sont les valeurs propres de A^TA et V est la matrice de passage associée à A^TA. De même U est celle associée à AA^T. Ainsi on a $A = U\Sigma V^T$.

Nous avons donc :

$$A = U\Sigma V^T \iff AV = U\Sigma \tag{IV.1}$$

Notons $V = [\vec{v}_1 ... \vec{v}_n]$ et $U = [\vec{u}_1 ... \vec{u}_m]$. On a alors :

$$A\vec{v}_i = \sigma_i \vec{u}_i \tag{IV.2}$$

Ce qui ressemble fortement aux propriétés des valeurs propres des matrices carrées.

Soit r le nombre de valeurs singulières non-nulles : $\sigma_{r+1} = ... = \sigma_n = 0$. On a la Décomposition en Valeurs Singulières suivantes :

$$\begin{cases} A\vec{v}_i = \sigma_i \vec{u}_i & 1 \leq i \leq r \\ A\vec{v}_i = 0 & r+1 \leq i \leq n \end{cases} \tag{IV.3}$$

A est la somme de matrice de rang 1.

Nous avons donc : Im $A = \text{Vect}\{\vec{v}_1, ..., \vec{v}_r\}$ et Ker $A = \text{Vect}\{\vec{v}_{r+1}, ..., \vec{v}_n\}$ et rang $A = r$.

Sous Scilab, on utilise l'outil `[U, S, V] = svd(A)`. En comparaison, `[vp, V] = spec(S)` donne les valeurs propres vp d'une matrice carré et leur vecteurs propres associés V.

2) Compression d'image

Une des applications possibles de la SVD est de compresser des images. L'idée est de supprimer des valeurs singulières négligeables afin de réduire la taille de la factorisation. Une image en noir et blanc de résolution $m \times n$ pixels peut être vue comme une matrice $A \in \mathcal{M}_{mn}$ où a_{ij} correspond au niveaux de gris du pixel.

A partir de la décomposition en valeurs singulières (IV.3), nous avons :

$$A = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

Définissons la matrice compressée à partir de la $(k+1)$-ième valeur singulière A_k telle que :

$$A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T \tag{IV.4}$$

On remarque que $A_r = A$.

On a l'écart entre l'image originale A et la version compressée A_k :

$$\|A - A_k\|_F^2 = \sum_{i=k+1}^{m} \sigma_i^2 \tag{IV.5}$$

où $\|.\|_F$ est la **norme de Frobenius** :

$$B = (b_{ij}) \to \|B\|_F^2 = \sum b_{ij}^2$$

Sous Scilab nous utiliserons : `norm(A, 'fro')`.

Pour choisir les valeurs singulières négligeables, nous définissons un seuil $\epsilon > 0$ et considérons nuls les σ_i dont la valeur absolue est inférieure au seuil choisi avec $i > k$. On remplace alors la matrice initiale A par celle compressée A_k définit en (IV.4).

Au départ, la matrice A occupe $m \times n$ cases mémoire. Après compression, il faut stocker $2k$ vecteurs u_i et v_i soit $k(m+n)$ espaces mémoires et les k valeurs singulières considérées comme intéressantes σ_i $(1 \leq i \leq k)$. Au total, on passe à $k(m+n+1)$ cases mémoires occupées après compression. Comme $k << m$, la compression est intéressante.

Nous allons utiliser cette méthode pour compresser l'image de Lena, une photographie très utilisée en traitement d'image pour des raisons historiques (des scientifiques pressés et désireux d'utiliser une nouvelle image d'essai ont trouvé cette image dans un magazine *Playboy*).

On a l'implémentation suivante :

```
clear;
// Chargement de la matrice à partir du csv de l'image
lena = read("D:\Documents\Cours\TC04 - Printemps
 ↪  2017\MT94\Scilab\TD4-Valeurs_Propres\lena.csv",512,512)';
n=512;                // Résolution de l'image

// Map de pixels pour l'affichage
x = [1:512];
y = [512:-1:1];        // inverser car grayplot inverse

// Décomposition SVD
[U,S,V] = svd(lena);
sig = diag(S);

clf; subplot(2,3,4);
title('$\huge \text{Rapport } \frac{\sigma_i}{\sigma_1}$');
plot(sig/sig(1),'k');

// Récupération de la dernière valeur supérieure au seuil
eps = 0.005;
for k1=1:n-1
    if sig(k1+1)<eps*sig(1) then
        break;
    end:
end
k2 = 50;
plot(k1, sig(k1)/sig(1), 'rx', 'MarkerSize', 10);
plot(k2, sig(k2)/sig(1), 'b+', 'MarkerSize', 10);

// Somme pour obtenir les Ak
Ak1 = sig(1)*U(:,1)*V(:,1)';
for i=2:k1
    Ak1 = Ak1 + sig(i)*U(:,i)*V(:,i)';
end
dk1 = norm(lena-Ak1, 'fro');  // Ecart

Ak2 = sig(1)*U(:,1)*V(:,1)';
for i=2:k2
    Ak2 = Ak2 + sig(i)*U(:,i)*V(:,i)';
end
dk2 = norm(lena-Ak2, 'fro');

// Affichage images et différence
set(gcf(), 'color_map', graycolormap(256));
subplot(2,3,1); isoview(0,n,0,n);
title('$\huge\text{Lena Originale}$');
grayplot(x,y, lena);

subplot(2,3,2); isoview(0,n,0,n);
title(sprintf('$\\huge\\text{Lena Compressée avec } k=%d$', k1));
grayplot(x,y, Ak1);
subplot(2,3,5); isoview(0,n,0,n);
title(sprintf('$\\huge\\text{Lena Compressée avec } k=%d$', k1));
title(sprintf('$\\huge\\text{Différence : } %f$', dk1));
grayplot(x,y, Ak1-lena);

subplot(2,3,3); isoview(0,n,0,n);
title(sprintf('$\\huge\\text{Lena Compressée avec } k=%d$', k2));
grayplot(x,y, Ak2);
subplot(2,3,6); isoview(0,n,0,n);
title(sprintf('$\\huge\\text{Différence : } %f$', dk2));
grayplot(x,y, Ak2-lena);

tailleK1 = k1*(n+n+1);
tailleK2 = k2*(n+n+1);
printf('\n%-31s = %d', 'Taille originale', n*n);
printf('\n%-32s = %d', 'Taille après compression k1='+string(k1), tailleK1);
printf('\n%31s = %2.2f%%', 'Taux de compression', 100-tailleK1/(n*n)*100);
printf('\n%-32s = %d', 'Taille après compression k2='+string(k2), tailleK1);
printf('\n%31s = %2.2f%%', 'Taux de compression', 100-tailleK2/(n*n)*100);
```

Code Source IV.1 – Compression par SVD

Après avoir effectué la SVD sur la matrice de l'image, on va choisir k_1 et k_2 tel que $\sigma_{k_1} > \text{eps} \times \sigma_1$ et $\sigma_{k_1+1} <$

$\texttt{eps} \times \sigma_1$ avec $\texttt{eps} = 0.005$ et $k_2 = 50$.

On obtient les matrices d'images compressées A_{k_1} et A_{k_2} suivantes :

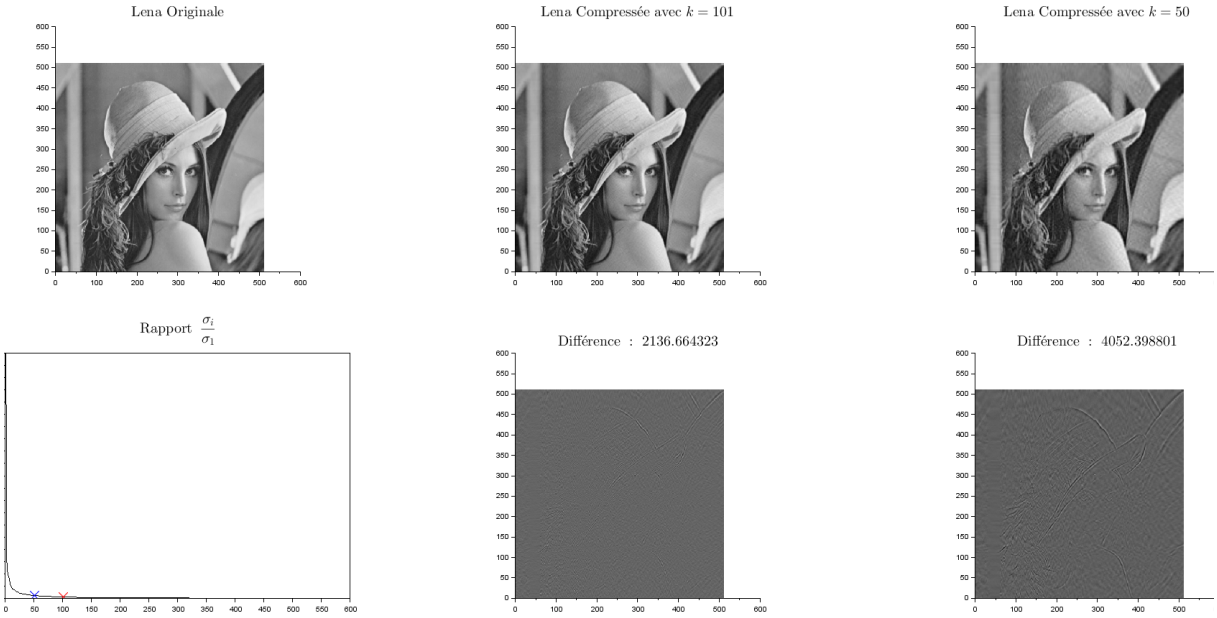


FIGURE IV.1 – Résultats de la compression

On ne remarque que peu de différences entre la photo originale et celle compressée avec k_1 mais ce n'est pas le cas pour la photo k_2.

Concernant la taille prise et la compression, on a :

Photo	Taille occupée		Compression[1]
Originale	$n \times n$	262144 valeurs	$1 - \frac{T_{\text{comp}}}{T_{\text{orig}}}$
Compressée $k_1 = 101$	$k_1(n+n+1)$	103525 valeurs	60.51%
Compressée $k_2 = 50$	$k_2(n+n+1)$	51250 valeurs	80.45%

TABLE IV.1 – Résultats de la compression des images

On conclut ainsi sur l'utilité de la compression SVD : avec la compression k_1 on réduit la taille de l'image par plus de la moitié pour un résultat quasiment semblable. D'autres techniques de compression existent, la plus connue étant la compression JPEG qui utilise des méthodes plus complexes et un sous-échantillonage de l'image.

3) Débruitage

Souvent en télécommunication, les signaux transmis sont légèrement bruités, la SVD permet alors de restituer le signal d'origine. Nous allons bruiter un signal de départ $u \in \mathbb{R}^{256}$ tel que $u(50) = u(52) = \frac{1}{4}, u(51) = \frac{1}{2}, u(150) = 1$ et le reste à 0.

On construit le premier signal bruité v tel que chacune de ses composantes soient une moyenne pondéré des composantes voisines, on crée alors un effet de flou diffu. Soit T la matrice de floutage définie par :

$$t_{ij} = C_i \times e^{-\frac{(i-j)^2}{10}} \tag{IV.6}$$

avec $C_i = \sum_{j=1}^{256}$ permettant de normaliser T tel que la somme de chaque ligne soit égale à 1.

Construisons le second signal bruité $w = v + \eta$ où η correspond à une perturbation de u par un bruit blanc de $\frac{1}{100}$:

On pourrait naïvement essayer de débruiter le signal en $\tilde{u} = T^{-1} w$ mais T n'est pas inversible : $|T| = 0$.

En effet on appliquant la SVD (IV.3) à T on a en passant au déterminant :

$$\det T = \det U \times \det \Sigma \times \det V$$

1. Ici nous définissons le taux de compression tel que une image compressée à un taux t occupe t % de la taille originelle

U et V sont orthogonales donc leur déterminant est égale à ± 1 et Σ diagonales donc on a :

$$\det T = \pm \prod_{i=1}^{256} \sigma_i$$

Comme certaines valeurs singulières sont très proches de 0, elles impactent le déterminant de T. On va alors tronquer T en T_k dont le déterminant sera non nul. T_k correspond ainsi à une version compressée mais inversible de T grâce à la SVD.

On a ensuite la pseudo-inverse de T :

$$T_k^+ = \sum_{i=1}^{k} \frac{1}{\sigma_i} V_i U_i^T \tag{IV.7}$$

qui nous donne un signal débruité : $u_k = T_k^+ w$.

L'outil Scilab permettant de calculer la pseudo-inverse d'une matrice est `pinv`.

Pour expérimenter cette technique sur l'image Léna, nous avons le code Scilab suivant :

```
clear;
lena = read("D:\Documents\Cours\TC04 - Printemps
 ↪  2017\MT94\Scilab\TD4-Valeurs_Propres\lena.csv",512,512)';
n = 512;
x = 1:n;        // Map de pixels pour l'affichage
y = n:-1:1;        // inverser car grayplot inverse

[U,S,V] = svd(lena);
sig = diag(S);

eps = 0.002;
for k=1:n-1
    if sig(k+1) < eps*sig(1) then
        break;
    end
end

// Matrice de floutage T
T = zeros(n,n);
for i=1:n
    for j=1:n
        T(i,j) = exp(-1/10 * (i-j)^2);
    end
    C(i) = sum(T(i,:));        // Correction pour somme de la ligne = 1
    T(i,:) = T(i,:)/C(i);
end

// Bruit blanc
eta = rand(n,n);
eta = eta/norm(eta);
eta = eta*norm(lena)*0.001;

v = T*lena*T;    // Floutage
w = v+eta;        // Bruitage
Ab = w;            // Image bruitée

// SVD de T
[UT, ST, VT] = svd(T)
sigT = diag(ST)

Tk = zeros(T');
for i=1:k
    Tk = Tk + sigT(i)*UT(:,i)*VT(:,i)'
end
Ar = pinv(Tk)*w*pinv(Tk);

// Écart
db = norm(lena-Ab, 'fro');
dr = norm(lena-Ar, 'fro');

clf; xset('colormap', graycolormap(256));
subplot(1,3,1); isoview(0,n,0,n);
grayplot(x,y,lena);
title(sprintf('$\\huge\\text{Image originale}$'));

subplot(1,3,2); isoview(0,n,0,n);
grayplot(x,y,Ab);
title(sprintf('$\\huge\\text{Image bruitée : écart }= %f$', db));

subplot(1,3,3); isoview(0,n,0,n);
grayplot(x,y,Ar);
title(sprintf('$\\huge\\text{Image débruitée : écart }= %f$', dr));
```

Code Source IV.2 – Débruitage de Léna

Nous floutons d'abord l'image avec T, puis nous la bruitons avec un bruit blanc η pour obtenir l'image bruitée A_b. Nous créons la matrice restaurée A_r de la manière décrite précedemment avec la pseudo-inverse de T. Nous obtenons les résultats suivants :

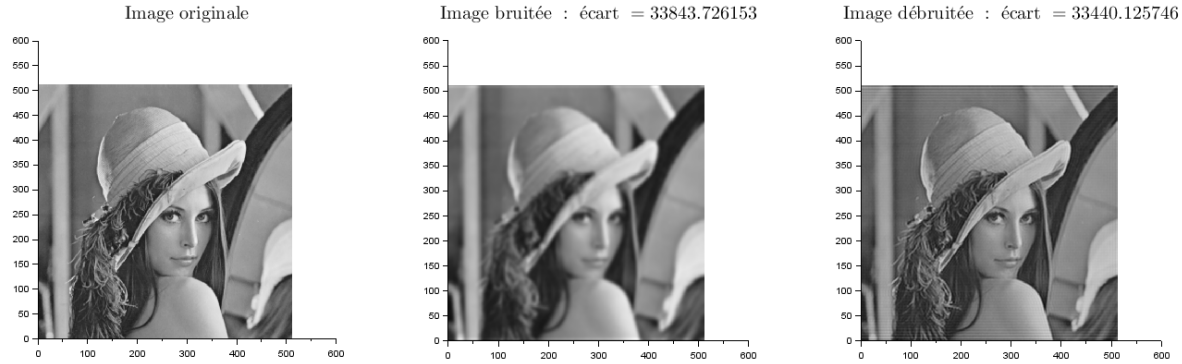


FIGURE IV.2 – Débruitage de Léna

La version bruitée possède un écart de la norme de Frobenius supérieure à la version débruitée, on améliore bien la qualité de l'image après le processus de débruitage. De plus ce résultat ce voit visuellement.

4) Approximation au sens des moindres carrés

La SVD permet aussi d'approcher la solution d'un problème linéaire. Nous verrons plus en détail d'autres techniques dans le chapitre 5 (VI). On cherche ici à résoudre $Ax = b$ avec $A \in \mathcal{M}_{mn}, x \in \mathcal{M}_{n1}, b \in \mathcal{M}_{m1}$. On a plusieurs cas :

— $b \in \text{Im } A$:
  — Ker $A = 0$ → A est inversible, l'unique solution est $x = A^{-1}b$
  — Ker $A \neq 0$ → il existe une infinité de solutions de la forme $x = x_{\text{particulier}} + x_{\text{Ker}}$

— $b \notin \text{Im } A$: Dans ce cas là, qui arrive souvent en pratique, on doit approcher la valeur de x par x^*

On cherche donc x^* l'approximation de x telle que $\|Ax^* - b\|^2$ soit le plus petit possible.

Grâce à la SVD, on a : $A = U\Sigma V^T$

$$\begin{aligned}\|Ax - b\|^2 &= \|U\Sigma V^T x - b\|^2 = \|U\Sigma V^T x - UU^T b\|^2 \\ &= \|U(\Sigma V^T x - U^T b)\|^2 = \|\Sigma V^T x - U^T b\|^2 \qquad \text{car } U \text{ est orthogonale donc il y a isométrie}\end{aligned}$$

On pose $z = V^T x$ et $c = U^T b$. On a alors :

$$\begin{aligned}\|Ax - b\|^2 &= \|\Sigma z - c\|^2 \\ &= \|[\sigma_1 z_1 - c_1, ..., \sigma_r z_r - c_r, 0 - c_{r+1}, ..., 0 - c_m]^T\|^2 \\ &= \sum_{i=1}^{r} (\sigma_i z_i - c_i)^2 + \sum_{i=r+1}^{m} c_i^2\end{aligned}$$

Il suffit donc de poser $\sigma_i z_i - c_i = 0 \forall i \in \{1, ..., r\}$

De cette façon, on a :

$$z_i = \begin{cases} \frac{c_i}{\sigma_i} & i \in \{1, ..., r\} \\ \text{arbitraire} & i \in \{r+1, ..., m\} \end{cases}$$

On a enfin le résultat approché au sens des moindres carrés : $x^* = Vz$. Cette méthode fonctionne aussi si A est de rang plein et donne la solution unique.

Dans le Big Data, on a de très grandes matrices dont les colonnes correspondent à différentes caractéristiques. La SVD permet alors de dégager les k caractéristiques les plus importantes par rapport aux valeurs singulières correspondantes : les σ_i sont triés par ordre décroissant et donc les caractéristiques aussi.

II) Méthode de la puissance

Soit $A \in \mathcal{M}_{nn}$ admettant $\lambda_1, ..., \lambda_n$ valeurs propres telles que $|\lambda_1| > |\lambda_2| \geq ... \geq |\lambda_n|$. On appelle λ_1 la valeur propre dominante (IV.2).

Le but de cette méthode est d'approcher λ_1 et v_1 son vecteur propre associé.

On cherche alors à exprimer la suite : $x_k = Ax_{k-1}$ On part de $x_0 \in \mathbb{R}$ tel que $x_0 = \sum_{i=1}^{n} \alpha_i v_i$ avec $\alpha_i \neq 0 \forall i \in \{1, ..., n\}$. $\{v_1, ..., v_n\}$ forme une base de vecteurs propres de A.

On a :

$$x_1 = Ax_0 = A\left(\sum_{i=1}^{n} \alpha_i v_i\right) = \sum_{i=1}^{n} \alpha_i A v_i = \sum_{i=1}^{n} \alpha_i \lambda_i v_i$$
$$x_2 = Ax_1 = A\left(\sum_{i=1}^{n} \alpha_i \lambda_i v_i\right) = \sum_{i=1}^{n} \alpha_i \lambda_i A v_i = \sum_{i=1}^{n} \alpha_i \lambda_i^2 v_i$$

Par récurrence on obtient :

$$\begin{aligned} x_k = Ax_{k-1} &= \sum_{i=1}^{n} \alpha_i \lambda_i^k v_i \\ &= \alpha_1 \lambda_1^k v_1 + \sum_{i=2}^{n} \alpha_i \lambda_i^k v_i \\ &= \alpha_1 \lambda_1^k v_1 \left(v_1 + \sum_{i=2}^{n} \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_i}{\lambda_1}\right)^k v_i\right) \end{aligned}$$

Si $k >> 1$, comme $\lambda_1 > \lambda_i \forall i > 1$, alors $x_k \sim \lambda_1^k \alpha_1 v_1$. Ainsi x_k a quasiment la même direction que v_1.

Cependant il y a un problème : si $|\lambda_1| > 1$ alors λ_i^k diverge. Pour remédier à cela, nous allons normaliser les x_i en y_i.

Algorithme IV.1 : Méthode de la puissance

Data : $y_0 \in \mathbb{R}^n$ donné
k la puissance désirée
Result : $y_k \in \mathbb{R}^n$
for *i allant de* 1 *à* k **do**
  $x_i = Ay_{i-1}$
  $y_i = \frac{x_k}{\|x_k\|}$
end for

Enfin nous povons récupérer la valeur propre dominante avec cette méthode :

$$y_n^T A x_n = \langle x_n, Ax_n \rangle = \langle \pm v_1, \pm Ac_1, \lambda \rangle_1^k = y_k^T A y_k$$

On peut montrer que :

$$|\lambda^k - \lambda_1| \leq C\left(\frac{\lambda_2}{\lambda_1}\right)^k$$

Le code suivant est l'implémentation de cette méthode sous Scilab :

```
function [lambda, y, k] = methodePuissance(A, x, TOL, ITE_MAX)
    y = x/norm(x);

    for k=1:ITE_MAX
        x = A*y;                    // Puissance
        x = x/norm(x);                // Normalisation
        if norm(x-y) < TOL then     // Arrêt si l'écart est inférieur au seuil de tolérance
            break;
        end
        y = x;
    end

    lambda = x'*A*x;                // Valeur propre dominante
endfunction
```

Code Source IV.3 – Méthode de la puissance

Testons cette méthode sur la matrice suivante :

$$A = \begin{pmatrix} 0.5172 & 0.5473 & -1.224 & 0.8012 \\ 0.5473 & 1.388 & 1.353 & -1.112 \\ -1.224 & 1.353 & 0.03642 & 2.893 \\ 0.8012 & -1.112 & 2.893 & 0.05827 \end{pmatrix} \tag{IV.8}$$

Nous utilisons de code suivant pour tester la méthode IV.3 :

```
exec("D:\Documents\Cours\TC04 - Printemps
 ↪  2017\MT94\Scilab\TD4-Valeurs_Propres\2-Methode_Puissance.sce");

A = [    0.5172,    0.5473,    -1.224,         0.8012;
        0.5473,    1.388,    1.353,         -1.112;
        -1.224,    1.353,    0.03642,     2.893;
        0.8012,    -1.112,    2.893,         0.05827 ]

vps = spec(A);
disp(vps, 'Valeurs propres de A');

x0 = [1; 0; 0; 0];

[lambda, vp, nbIte] = methodePuissance(A, x0, 10^-2, 1000);

disp(lambda, 'Valeur propre dominante de A');
disp(vp, 'Vecteur propre de A associé à la valeur propre dominante');
disp(nbIte, 'Nombre ditération');
```

Code Source IV.4 – Test de la méthode de la puissance

On obtient la valeur propre dominante $\lambda = -3.9956707$.
Grâce à `spec(A)`, nous obtenons les valeurs propres de la matrice :

$$-3.99567071.00052011.99274573.0022949$$

On remarque que λ est bien la valeur propre de valeur absolue la plus grande, elle est bien dominante.

III) Théorème de Perron-Frobenius

Soit $A \in \mathcal{M}_{nn}$.

DÉFINITION IV.1 :
Si $\forall (i,j) a_{ij} \geq 0$ alors la matrice A est dite positive et on note $A \geq 0$. Respectivement, si l'inégalité est stricte, la matrice $A > 0$ est strictement positive.

DÉFINITION IV.2 :
Une valeur propre λ^* est dite dominante si $\forall \lambda \neq \lambda^* : \quad |\lambda^*| > |\lambda|$

DÉFINITION IV.3 :
Une matrice A est dite primitive si elle est positive et si $\exists k \in \mathbb{N}$ tel que $A^k > 0$.

DÉFINITION IV.4 :
Une matrice A est dite irréductible si $\forall (i,j) \exists k \in \mathbb{N}$ tel que $(A^k)_{ij} > 0$.

Une matrice primitive est irréductible, cependant la réciproque est fausse.

THÉORÈME IV.2 : Théorème de Perron
Si A est une matrice primitive, alors λ^* est une valeur propre simple et dominante et il existe un unique vecteur $x \in \mathcal{M}_{n1}$ positif et normalisé [1] tel que $Ax = \lambda^* x$.

THÉORÈME IV.3 : Théorème de Frobenius
Si A est une matrice irréductible, alors λ^* est une valeur propre simple et il existe un unique vecteur $x \in \mathcal{M}_{n1}$ positif et normalisé tel que $Ax = \lambda^* x$.

La seule différence entre ces deux théorèmes est que la valeur propre n'est pas dominante si la matrice n'est que irréductible dans le cas du théorème de Frobenius (IV.3).

1. On définit x comme normalisé si la somme de ses composantes est égale à 1 : $\sum_{i=1}^{n} x_i = 1$

IV) PageRank

L'algorithme *PageRank* a été conçu par Larry Page et Serguey Brin, les fondateurs de Google, dans le cadre d'une thèse à Stanford en 1996. L'idée principale est que la popularité et l'importance d'une page web se mesure avec les liens qui lui font référence. D'autres paramètres rentrent bien sûr en compte et nous les verrons au fur et à mesure de l'élabolartion de l'algorithme.

Cette idée peut être appliquée sur n'importe quel graphe orienté tel qu'un réseau ferroviaire ou bien un ensemble d'articles scientifques se faisant référence entre eux; dans chaque cas on mesure l'importance de chaque sommet.

1) Construction de la méthode

Nous allons ici détaillé la construction de l'algorithme *PageRank* dans le cas initial d'un réseau de pages web. On part d'un ensemble de n pages P_i ($i \in I = \{1,...,n\}$) reliées entre elles ou non selon la matrice d'adjacence $A \in \mathcal{M}_{nn}$:

$$a_{ij} = \begin{cases} 1 & \text{si } j \text{ fait référence à } i \text{ (on a l'arc } P_j \to P_i) \\ 0 & \text{sinon} \end{cases}$$

Les colonnes A_j présentent donc les successeurs de P_j.

En modélisant l'action d'un internaute lambda sur ce réseau, le but est de prédire quelles seront les pages les plus fréquentées et ainsi de les classer. On définit E_k^i l'événement "*être à la page P_i après k clics*" de probabilité $p_k^i = \mathbb{P}(E_k^i)$. On suppose que le choix de la page de départ est équiprobable : $p_0^i = \frac{1}{n} \quad \forall i \in I$. A chaque itération k, l'internaute change de page.

D'après la formule des probabilités totales, on a $\forall i \in I, \forall k \in \mathbb{N}$:

$$p_{k+1}^i = \sum_{j \in I} \mathbb{P}(E_{k+1}^i | E_k^j) \times \mathbb{P}(E_k^j)$$

On note le vecteur de probabilité $U_k \in \mathcal{M}_{n1}$:

$$U_k = \begin{pmatrix} p_k^1 \\ \vdots \\ p_k^n \end{pmatrix} \tag{IV.9}$$

qui correspond à la probabilité d'être sur chaque page à l'itération k.

On pose la matrice de transfert d'importance relative $H \in \mathcal{M}_{nn}$:

$$H_i = \begin{cases} A_i \times \dfrac{1}{\sum\limits_{i \in I} a_{ij}} & \text{si } A_i \neq 0 \\ \\ \frac{1}{n} \times e & \text{sinon} \end{cases} \tag{IV.10}$$

Où $e \in \mathcal{M}_{n1}$ est le vecteur rempli de 1.

Dans le cas où la page P_i n'a aucun lien qui pointe vers elle, A_i est nulle. Cependant nous voulons que la somme de chaque colonne H_i soit égale à 1 et éviter les bloquage sur des pages sans successeurs, c'est pourquoi nous donnons $h_{ij} = \frac{1}{n}$ quand $A_i = 0$. Dans le cas contraire, on a $h_{ij} = \mathbb{P}(E_{k+1}^i | E_k^j)$

On a donc la chaîne de Markov suivante :

$$\begin{cases} U_{k+1} = HU_k \\ U_0 = \frac{1}{n} \times e \end{cases} \tag{IV.11}$$

L'importance des pages est alors indiquée par la convergence de la suite U_k vers $r \in \mathbb{R}^n$. Cependant si on s'arrête ici, on ne garantit pas la convergence dans le cas de circuits.

Pour palier à ce problème et mieux représenter l'attitude d'un internaute, on suppose qu'à n'importe quel moment ce dernier peut choisir de réinitialiser sa navigation, c'est-à-dire quitter la page courante et choisir n'importe quelle page. On note (1-α) la probabilité d'un tel événement R. La probabilité de choisir n'importe quelle page après réinitialisation est équiprobable et de probabilité $\frac{1}{n}$.

On a finalement la matrice Google :

$$G = \alpha H + (1-\alpha) \times \frac{1}{n} \times ee^T \tag{IV.12}$$

Google utilise $\alpha = 0.85$: cela permet d'avoir un comportement sans trop de réinitialisations de la navigation mais permettant tout de même d'éviter les problèmes empêchant la convergence.

On pose la somme de la colonne j : $c_j = \sum_{i \in I} a_{ij}$. En résumant nous avons donc :

$$g_{ij} = \begin{cases} \alpha \frac{a_{ij}}{c_j} + (1-\alpha)\frac{1}{n} ee^\top & \text{si } c_j \neq 0 \\ \frac{1}{n} & \text{si } c_j = 0 \end{cases} \tag{IV.13}$$

La matrice Google G est primitive car $G > 0$ (ses composantes sont des probabilités). Elle est ainsi construite afin de remplir les conditions du théorème de Perron (IV.2).

On a : $Ge = \sum_{i=1}^{n} \underline{G}_i = e$ et e normalisé Ainsi $\lambda^* = 1$ est la valeur propre dominante de G.

On a alors l'unique vecteur propre r positif et normalisé associé à la valeur propre dominante $\lambda_1 = 1$:

$$r = Gr \tag{IV.14}$$

L'importance des pages $r \in \mathcal{M}_{n1}$ est ainsi donnée par r.

On peut calculer le rank r à partir de (IV.14) avec la méthode du point fixe. Comme la matrice G est énorme, il est grandement préférable de faire les opérations pour des matrices creuses. Ici nous utilison la méthode de la puissance vu en IV.3 sur G pour récupérer le rank R. Nous avons le code Scilab suivant :

```
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD4-Valeurs_Propres\2-Methode_Puissance.sce");
function [G, R] = Rank(fileName)
    alpha = 0.85;                // Proportion magique pour cette recette

    // Lecture la matrice d'adjacence
    A = fscanfMat(fileName);

    [n p] = size(A);
    if (n <> p) then
        disp("La matrice nest pas carrée");
        abort;
    end

    // Construction de la matrice Google
    for j=1:n
        s = sum(A(:,j));            // Somme de la colonne
        if (s == 0) then
            G(:,j) = 1/n;
        else
            G(:,j) = alpha*A(:,j)./s + (1-alpha)/n;
        end
    end

    x0 = ones(n,1)./n;         // Equiprobabilité de départ
    [lambda, R, k] = methodePuissance(G, x0, 10^-10, 100);

    R = R/sum(R);
endfunction
```

Code Source IV.5 – Rank d'une matrice

Il suffit juste de créer la matrice en format csv et de la passer à la fonction pour récuperer le rank. Cette technique n'est pas adaptée pour de grandes matrices mais ici nous n'utiliserons que de petites matrices carrées de taille inférieure à 12. Pour adapter cette fonction à de grandes matrices, il faudrait passer une matrice creuse construites préalablement.

2) Applications

a) PageRank d'un réseau de 8 pages

Nous allons appliquer l'algorithme du *PageRank* dans le réseau de pages suivant :

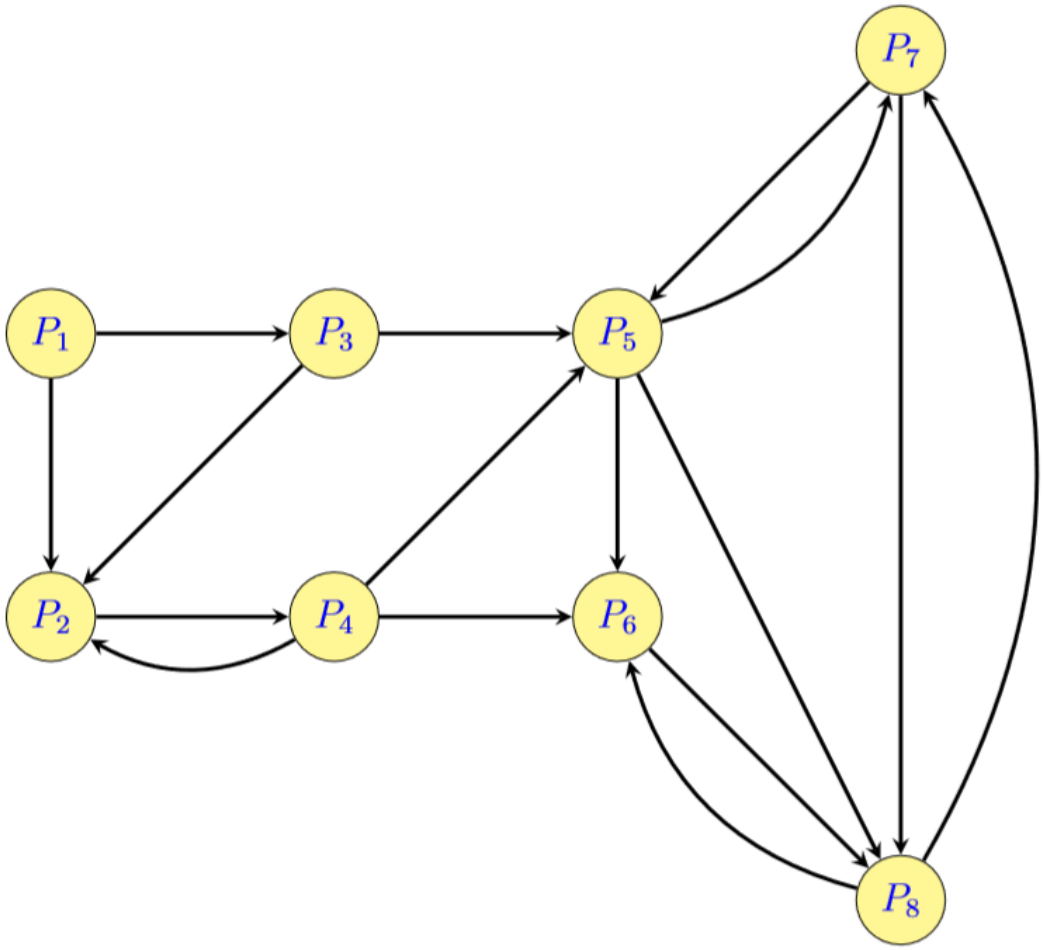


FIGURE IV.3 – Réseau de 8 pages

Nous avons la matrice d'adjacence A suivante :

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Puis la matrice H :

$$H = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 1/2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1/2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 & 0 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 1/2 & 0 & 0 & 1/2 \\ 0 & 0 & 0 & 0 & 1/2 & 1 & 1/2 & 0 \end{pmatrix}$$

On a bien la somme de chaque colonne H_i égale à 1 : $\sum_{i=1}^{8} h_{ij} = 1,\ \forall j$.

On obtient ici la matrice stochastique G :

$$G = 0.85 \times H + 0.15 \times \frac{1}{8}$$

Appliquons l'algorithme IV.5 avec le code suivant :

```
clear;
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD4-Valeurs_Propres\3-Rank.sce");

[G R] = Rank('D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD4-Valeurs_Propres\web1.csv');

// Affichage du rank trié
[Rs,s] = gsort(R);

printf("\n");
printf("  %8s | %-10s\n","Rank","Page");
printf(" ----------|------------\n");
printf("  %8.6f | %-10d\n", Rs, s);
printf("\n");
```

Code Source IV.6 – PageRank du réseau de 8 pages

Après tri du rank nous obtenons :

Rank	Page
0.322208	8
0.213505	7
0.182237	6
0.136039	5
0.062469	4
0.038074	2
0.026719	3
0.018750	1

TABLE IV.2 – Rank du réseau de 8 pages

La page P_8 est donc la plus importante.

b) Rank d'un réseau ferroviaire

L'algorithme peut s'appliquer à n'importe quel réseau, y compris au réseau ferroviaire suivant :

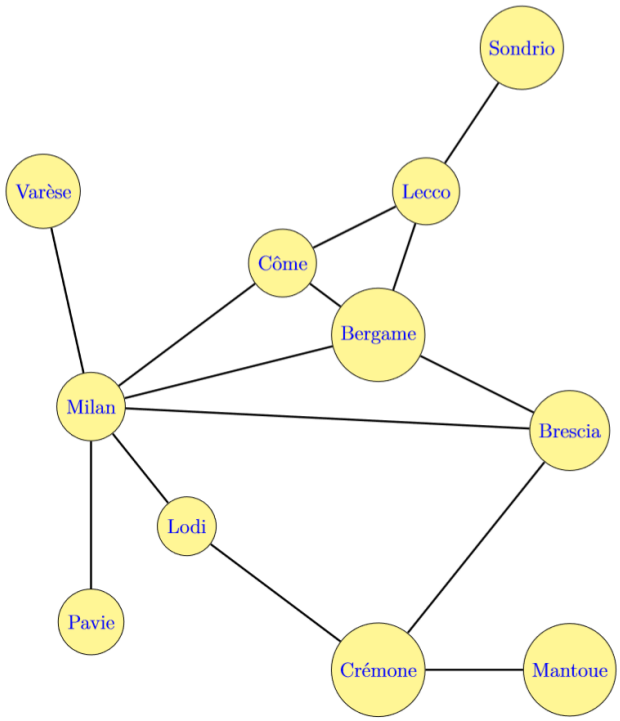


FIGURE IV.4 – Réseau ferroviaire entre 11 villes

Nous avons la matrice d'adjacence A suivante :

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Appliquons pareillement l'algorithme IV.5 avec le code suivant :

```
clear;
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD4-Valeurs_Propres\3-Rank.sce");

Villes = ["Varèse"; "Milan"; "Pavie"; "Lodi"; "Côme"; "Bergame"; "Crémone"; "Lecco"; "Sondrio";
 ↪  "Brescia"; "Mantoue"];
[G R] = Rank('D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD4-Valeurs_Propres\villes.csv');

// Affichage du rank trié
[Rs,s] = gsort(R);
printf("  %8s | %-20s\n","Rank","Villes");
printf(" ----------|------------\n");
printf("  %8.6f | %2d - %-16s\n", Rs, s, Villes(s));
printf("\n");
```

Code Source IV.7 – Rank du réseau ferroviaire

Après tri du rank nous obtenons :

Rank	Villes
0.201485	2 - Milan
0.129563	6 - Bergame
0.112639	7 - Crémone
0.106809	8 - Lecco
0.101626	10 - Brescia
0.099975	5 - Côme
0.074094	4 - Lodi
0.045551	11 - Mantoue
0.043899	9 - Sondrio
0.042180	1 - Varèse
0.042180	3 - Pavie

TABLE IV.3 – Rank du réseau de 8 pages

Sans surprise, Milan est la ville la plus importante du réseau. Cet algorithme peut donc servir dans la prise de décision du nombre de trains passant sur chaque ligne par exemple.

CHAPITRE V

Optimisation

Un problème d'optimisation dans $\mathbb{R}^n$ peut se présenter de deux manière différentes. Soit nous cherchons à minimiser (respectivement maximiser) une fonction f, c'est-à-dire trouver x^* tel que $f(x^*) \leq f(x)\ \forall x$ (respectivement $f(x^*) \geq f(x)$). Soit nous cherchons à trouver un modèle f qui corresponde le mieux à des données. Ces deux problèmes, bien qu'initialement différents, se résolvent quasiment de la même manière. Nous verrons cela au long du chapitre.

I) Optimisation

1) Cadre statistique

A partir de n données (x_i, y_i) on cherche un modèle $f(x,\theta)$ paramétré par θ.

Ainsi $x \in \mathbb{R}$ est la variable indépendante, $y \in \mathbb{R}^n$ la variable dépendante de x qui correspond aux résultats observés selon les conditions x, et $\theta \in \mathbb{R}^p$ les p paramètres du modèle f. Il faut d'abord choisir un modèle pertinent et ensuite trouver les paramètres qui permettent de minimiser les écarts entre le modèle théorique et les résultats obtenus expérimentalement afin d'obtenir le modèle paramétré le plus vraisemblable face aux données.

Comme les y_i sont des mesures, il y a forcément des incertitudes : ce sont donc des variables alétoires que l'on suppose statistiquement indépendantes telles que :

$$y_i = f(x_i,\theta) + \epsilon_i \tag{V.1}$$

Où toute la partie aléatoire est contenue dans la variable aléatoire ϵ_i d'espérance nulle (il y a autant de probablité de sur-estimer que de sous-estimer y_i) et de variance σ^2. ϵ représente ainsi les erreurs de mesures.

On pose g la densité de ϵ. La densité de probabilité de y_i est alors :

$$\phi_i(y_i,\theta) = g(y - f(x_i,\theta))$$

Et on a :

$$E[y_i|\theta] = f(x_i,\theta)$$

Les $(y_i)_{i=1..n}$ sont indépendants, la densité conjointe du vecteur $Y = (y_1, ..., y_n)$ est alors :

$$\phi(Y,\theta) = \prod_{i=1}^{n} \phi_i(y_i,\theta) \tag{V.2}$$

Ainsi la probabilité que les données expérimentales se trouvent dans un certain domaine $D \in \mathbb{R}^n$ est :

$$\mathbb{P}(Y \in D|\theta) = \int_D \phi(Y,\theta)\, dY$$

On définit maintenant $L(\theta, Y) = \phi(Y,\theta)$ la fonction de vraisemblance (*Likelihood function* en anglais, d'où le L). Sa différence avec ϕ la densité de Y où θ est fixé et Y est la variable aléatoire ; est que cette fois Y est fixé par les données obtenues expérimentalement et les paramètres θ constituent la variable, de sorte que la plus haute vraisemblance est atteinte pour $\hat{\theta}$ tel que :

$$\hat{\theta} = \arg\max_{\theta \in \mathbb{R}^p} L(\theta, Y)$$

A partir de là, on peut effectuer différentes hypothèses sur la densité de ϵ qui donneront des méthodes différentes.

2) Méthode des moindres carrés

Pour cette méthode, on suppose que ϵ suit une loi normale $\mathcal{N}(0,\sigma^2)$:

$$g(\epsilon) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2\sigma^2}\times\epsilon^2} \tag{V.3}$$

On a donc :

$$\phi_i(y_i,\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2\sigma^2}\times(y-f(x_i,\theta))^2}$$

A partir de (V.2), on obtient ϕ puis directement L :

$$\begin{aligned} L(\theta,Y) &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2\sigma^2}\times(y-f(x_i,\theta))^2} \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp^{-\frac{1}{2\sigma^2}\times\sum_{i=1}^{n}(y-f(x_i,\theta))^2} \end{aligned}$$

Maximiser L, et donc une exponentielle, revient à minimiser la somme des carrés.

$$\hat{\theta} = \arg\max_{\theta\in\mathbb{R}^p} L(\theta,Y) = \arg\min_{\theta\in\mathbb{R}^p} S(\theta)$$

Ainsi nous ramenons le problème de maximisation de $L(\theta,Y)$ à la minimisation de $S(\theta)$ définie par :

$$S(\theta) = \sum_{i=1}^{n} \big(y_i - f(x_i,\theta)\big)^2 = \sum_{i=1}^{n} (r_i(\theta))^2 = \|r(\theta)\|^2 \tag{V.4}$$

On appelle $r(\theta)$ le vecteur des résidus. En partant de l'hypothèse que la distribution de ϵ est Gaussienne (V.3), on obtient ainsi la méthode des moindres carrés (V.4).

3) Méthode de la moindre déviation absolue

Pourquoi utiliser la somme des normes au carré et non seulement la somme des normes On suppose ici que ϵ suit une distribution de Laplace :

$$g(\epsilon) = \frac{1}{\sqrt{2\sigma^2}} \exp^{-\frac{\sqrt{2}}{\sigma}\times|\epsilon|} \tag{V.5}$$

Ce qui nous donne :

$$\begin{aligned} L(\theta,Y) &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\sigma^2}} \exp^{-\frac{\sqrt{2}}{\sigma}\times|y-f(x_i,\theta)|} \\ &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\sigma^2}} \exp^{-\frac{\sqrt{2}}{\sigma}\times\sum_{i=1}^{n}|y-f(x_i,\theta)|} \end{aligned}$$

Ainsi nous obtenons :

$$S(\theta) = \sum_{i=1}^{n} |y - f(x_i,\theta)| \tag{V.6}$$

Cependant cette hypothèse pose déjà un problème, S n'est pas différentiable. C'est pourquoi on préfère généralement utiliser la méthode des moindres carrés (V.4) qui fournit un bon support si l'on ne connaît pas la distribution des incertitudes ϵ pour une expérience.

On voit bien que la méthodes des moindres carrés et celle de la moindre déviation absolue ont des distributions différentes, ainsi les paramètres obtenus avec les même données pour chaque méthode seront différents.

4) Sélection des modèles

De plus on se peut se demander quel modèle choisir par ceux entraînés : par exemple dans le cas d'une régression polynomiale quel degré choisir ? En effet plus le degré du polynome augmente, plus l'erreur diminue.

Le principe est le suivant : on sépare les données en deux sets, un set de validation V pour calculer l'erreur et son complémentaire T qui sert à former le modèle. On forme d'abord les paramètres θ avec T, il s'agit de la phase d'apprentissage. On choisit alors le modèle réalisant l'erreur minimale sur le set de validation $S_V(\theta_k)$, on valide donc le bon modèle sur des données sur lesquelles il n'a pas été entraîné.

Cependant le choix des sets impacte la qualité de la validation : par exemple si on sépare les données en deux à partir d'une certaine valeur, on observera une extrapolation assez brutale qui ne coïncidera que très peut avec le set V. Pour éviter ce genre de biais, on préfèrera utiliser chaque donnée dans les deux types de sets, en réalisant les tests sur plusieurs sets T et V. Cette validation croisée est utile notamment quand le set T n'est pas très équilibré.

II) Problème linéaires

1) Régression polynomiale

On a $y = \sum_{k=0}^{d} \theta_k x^k = \theta_0 + \theta_1 x + ... + \theta_d x^d$ un modèle polynomiale de degré d. En utilisant la méthode des moindres carrés, on a :

$$S(\theta) = \sum_{i=1}^{n} \left(\sum_{k=0}^{d} \theta_k x^k - y_i \right) = \| r(\theta) \|^2$$

Le vecteur résiduel r s'écrit alors : Que l'on décompose :

$$r(\theta) = A\theta - y \tag{V.7}$$

où

$$r_i(\theta) = \begin{pmatrix} 1 & x_i & \dots & x_i^d \end{pmatrix} \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{pmatrix}$$

Avec A la matrice de Vandermonde suivante :

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 & ... & x_1^d \\ 1 & x_2 & x_2^2 & ... & x_2^d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & ... & x_n^d \end{pmatrix} \tag{V.8}$$

Cette matrice particulière est inversible si et seulement si les x_i sont tous distincts.
Ainsi on ramène le problème initial à trouver le minimum de S, $\hat{\theta}$ tel que :

$$\nabla S(\theta) = \nabla \| A\hat{\theta} - Y \|^2 = 0$$

On trouve le gradient grâce au développement limité suivant :

$$\begin{aligned} S(\theta + h) &= \| A(\theta + h) - Y \|^2 = \| A\theta - Y + Ah \|^2 \\ &= (A\theta - Y + Ah)^T (A\theta - Y + \alpha) \\ &= (A\theta - Y)^T (A\theta - Y) + (A\theta - Y)^T Ah + (Ah)^T (A\theta - Y)(Ah)^T Ah \\ &= \| A\theta - Y \|^2 + 2(A\theta - Y)^T Ah + \| Ah \|^2 \\ &= S(\theta) + \nabla S(\theta)^T h + \| Ah \|^2 \end{aligned}$$

Par identification $\| Ah \|^2 \xrightarrow[h \to 0]{} 0$ est le reste et on a le gradient :

$$\nabla S(\theta) = 2A^T (A\theta - y) \tag{V.9}$$

La solution $\hat{\theta}$ est donnée par :

$$\begin{aligned} \nabla S(\hat{\theta}) = 0 &\iff 2A^T (A\hat{\theta} - y) = 0 \\ &\iff A^T A\hat{\theta} - A^T y = 0 \\ &\iff A^T A\hat{\theta} = A^T y \end{aligned}$$

Cette solution est unique si le rang de A est p : $A^T A\hat{\theta} = A^T y \implies A\hat{\theta} = y$. Il n'y a plus qu'à résoudre le système linéaire par la méthode de Gauss [1].

Démonstration. $\forall \theta \in \mathbb{R}^p$:

$$\begin{aligned} S(\theta) = S(\hat{\theta} + \theta - \hat{\theta}) &= S(\hat{\theta}) + \nabla S(\hat{\theta})^T (\theta - \hat{\theta}) + \| A(\theta - \hat{\theta}) \|^2 \\ &= S(\hat{\theta}) + \| A(\theta - \hat{\theta}) \|^2 \\ &\geq S(\hat{\theta}) \end{aligned}$$

Donc $\hat{\theta}$ est solution minimale de S.

1. Simplement faire $\theta = A \backslash y$ sous Scilab

$$\begin{aligned} S(\hat{\theta}) = S(\theta) &\iff \|A(\theta - \hat{\theta})\|^2 = 0 \\ &\iff A(\theta - \hat{\theta}) = 0 \qquad \text{or A est injective car son rang est plein} \\ &\iff \theta = \hat{\theta} \end{aligned}$$

Ainsi $\hat{\theta}$ est unique si A est de rang p. □

EXEMPLE : Régression linéaire
On cherche θ_1 et θ_2 tels que :

$$y = f(\theta) = \theta_1 + \theta_2 x$$

Avec la méthode des moindres carrés, on a :

$$S(\theta) = \sum_{i=1}^{n} \big(y_i - f(x_i, \theta)\big)^2$$

Le but est alors de chercher θ^* minimisant S *i.e.* :

$$\theta^* = \arg\min_{\theta \in \mathbb{R}^p} S(\theta)$$

Nous allons voir ici l'application de la régression polynomiale sur un set de données. Nous avons le code suivant :

```
clear;
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\1.0-Data.sce");

DEG_POLY_MAX = 10;
NB_SETS = 3;
myColor = rainbowcolormap(DEG_POLY_MAX+1);

// Degré k => k+1 coeff
polys = zeros(DEG_POLY_MAX+1, DEG_POLY_MAX+1);
erreur = zeros(NB_SETS, DEG_POLY_MAX+1);
errMoy = zeros(DEG_POLY_MAX+1);

// Verification Sets
Va = find(abs(t) <=1);
Vb = find(abs(t) <=0.5);
Vc = 1:2:length(t);
Vall(1, 1:length(Va)) = Va;
Vall(2, 1:length(Vb)) = Vb;
Vall(3, 1:length(Vc)) = Vc;

tailleSet = [length(Va), length(Vb), length(Vc)];
clear Va Vb Vc;

for s=1:NB_SETS
    V = Vall(s, 1:tailleSet(s));    // Validation Set
    T = setdiff(1:length(t), V);    // Training Set

    // Calcul des polynomes
    A = [];
    for p=0:DEG_POLY_MAX
        A = [A t.^p];                // Concaténation avec degré supérieur
        theta = A(T,:)\y(T);        // Résolution des coefficient du polynome avec les données de
         ↪  Training
        yPoly = A*theta;            // y du polynome

        polys(p+1, 1:length(theta)) = polys(p+1, 1:length(theta)) + theta';    // Stockage des
         ↪  coefficients pour moyenne après
        // Ligne p = ploy de degré p, col = coeff

        erreur(s, p+1) = sum((yPoly(V)-y(V)).^2);  // Erreur de validation pour set s et poly p
    end
end
polys = polys./NB_SETS;                    // Moyenne de la somme déjà calculée

clf; subplot(1,2,1);
plot(t,y, "ok");
title("$\huge\text{Polynomes moyens sur les différents sets}$");
leg = legend("Données");
A =[];
for p=0:DEG_POLY_MAX        // Affichage des résultats des sets
    errMoy(p+1) = mean(erreur(:,p+1));    // Erreur moyenne du polynome sur tous sets
    A = [A t.^p];
    yPoly = A*polys(p+1,1:p+1)';
    plot(t, yPoly, 'Foreground', myColor(p+1,:));
    leg = legend(leg.text, sprintf("Degré %i", p));
end

// Affichage des erreurs
subplot(1,2,2);
bar(0:DEG_POLY_MAX, errMoy);
title("$\huge\text{Erreur moyenne par polynome}$");

// Affichage du meilleur polynome
[m, k] = min(errMoy);                    // k est la ligne du polynome donc deg = k-1
mprintf("\n Après validation croisée avec %i sets différents,\n le polynome optimal est de degré %i
 ↪  :\n", NB_SETS, k-1);
mprintf("\n   %.3f", polys(k,1));
for i=2:k
    mprintf(" + %.3f*x^%d", polys(k,i), i-1);
end
mprintf("\n");
```

Code Source V.1 – Régression Polynomiale

Nous allons construire plusieurs polynomes P_0 à P_{10} du degré $p = 0$ ($y = C \in \mathbb{R}$) au degré $p = 10$ sur les couples de données (t, y). Ici nous allons faire une validation croisée avec 3 sets différents, nous effectuerons ensuite une moyenne des erreurs et des coefficients sur chaque set pour chaque polynome P_p afin de choisir celui avec l'erreur minimale et qui collera donc le mieux aux données.

Pour chaque set, les coefficients des polynomes (θ_p) sont calculés avec un set d'apprentissage T par la méthode de Gauss. On calcule ensuite y_{Poly} pour chaque polynome sur l'ensemble des données pour estimer l'erreur sur le set de validation.

Nous obtenus les résultats suivants :

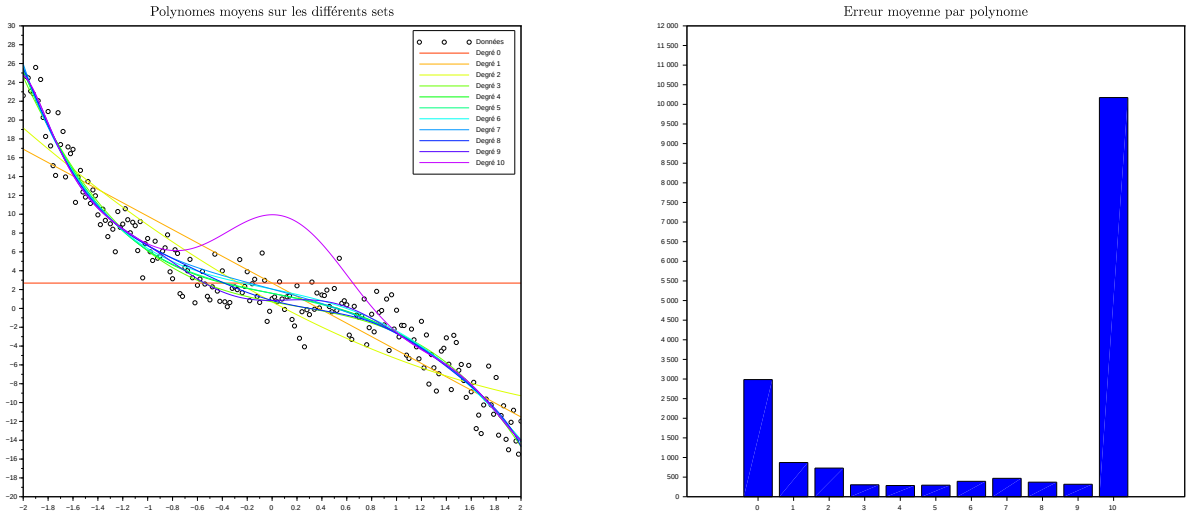


FIGURE V.1 – Résultats de la Régression Polynomiale

Le modèle le plus approprié serait alors :

$$P_4(x) = 1.599 - 2.521x^1 - 0.060x^2 - 1.825x^3 + 0.266x^4$$

Cependant on remarque qu'il y a relativement peu d'écart entre P_3, P_4 et P_5, les trois sont donc acceptables mais nous préfèrons P_4.

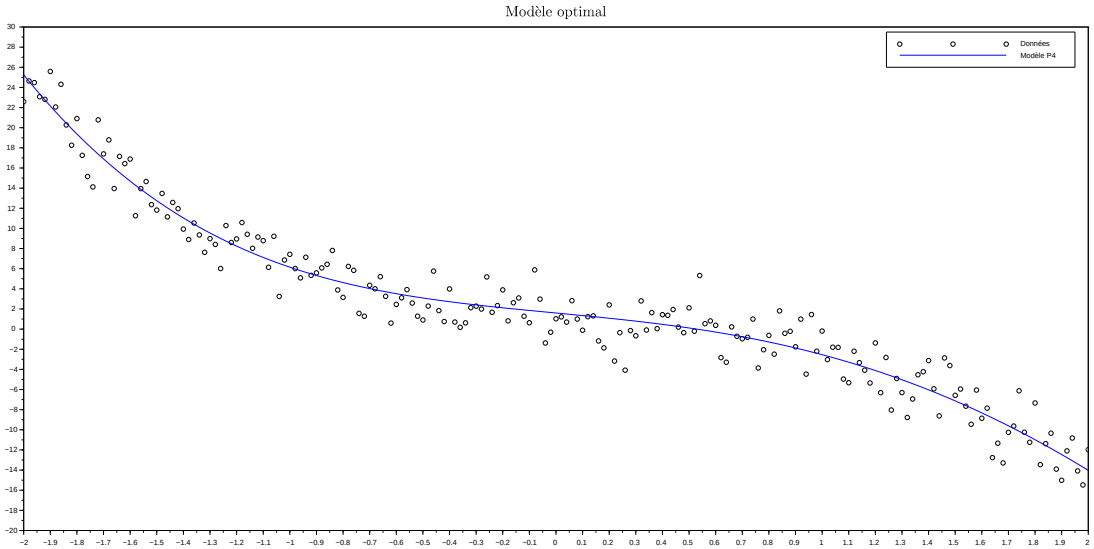


FIGURE V.2 – Polynome Optimal P_4

2) Régression polynomiale d'un cercle

Un cas particulier de l'utilisation de la régression polynomiale est si la fonction recherchée est celle d'un cercle. On cherche alors à minimiser la distance :

$$d(a,b,R) = \sum_{i=1}^{n} \left((x_i - a)^2 + (y_i - b)^2 - R^2\right)^2 = \|r\|^2 \tag{V.10}$$

Cependant cette forme n'est pas linéaire. On pose alors :

$$\begin{aligned} r_i &= R^2 - a^2 - b^2 + 2ax - i + 2by - i - (x_i^2 + y_i^2) \\ &= \begin{pmatrix} 2x_i & 2y_i & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ R^2 - a^2 - b^2 \end{pmatrix} - (x_i^2 + y_i^2) \end{aligned}$$

Cette forme est linéaire : $\theta = \begin{pmatrix} a & b & R^2 - a^2 - b^2 \end{pmatrix}^T$

Ainsi on pose :

$$A = \begin{pmatrix} 2x_1 & 2y_1 & 1 \\ 2x_2 & 2y_2 & 1 \\ \vdots & \vdots & \vdots \\ 2x_n & 2y_n & 1 \end{pmatrix} \quad \text{et} \quad z = \begin{pmatrix} x_1^2 + y_1^2 \\ x_2^2 + y_2^2 \\ \vdots \\ x_n^2 + y_n^2 \end{pmatrix}$$

Et on obtient le problème linéaire

$$S(\theta) = d(a,b,R) = \|A\theta - z\|^2 = \|r(\theta)\|^2 \tag{V.11}$$

Effectuons cette régression avec le code suivant :

```
clear;
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\1.0-Data.sce");

n = 100         // Nombre de données
RAND_MIN = 0.95;
RAND_ECART = .1;

// Génération des données 'expérimentales'
t = linspace(0,2*%pi,n)
rayon = 1;
centre = [1.5; 1.5];
bruit = RAND_MIN + rand(1,n)*RAND_ECART;

xData = centre(1)+rayon*cos(t).*bruit;
yData = centre(1)+rayon*sin(t).*bruit;

// Construction des matrices
A = [2*xData', 2*yData', ones(n,1)];
z = [xData'.^2 + yData'.^2];

x0 = zeros(3,1);    // Estimation initiale [a0;b0;c0]

theta = A\z;
a = theta(1);
b = theta(2);
R = sqrt(theta(3) + a^2 + b^2);

scf(0); clf;
plot(xData,yData,"ko");
set(gca(), 'isoview','on', 'data_bounds', [0 3 0 3]);
xSoluce = a + R*cos(t);
ySoluce = b + R*sin(t);
plot(xSoluce, ySoluce, 'r');
legend("Données bruitées", "Solution trouvée");
printf("\nOn obtient le cercle de centre (%.3f; %.3f) et de rayon %.3f", a,b,R);
```

Code Source V.2 – Régression d'un cercle

On construit d'abord un cercle bruité qui nous servira de données 'expérimentales'. On résout ensuite le problème de moindre carré linéaire (V.2). On obtient le résultat suivant :

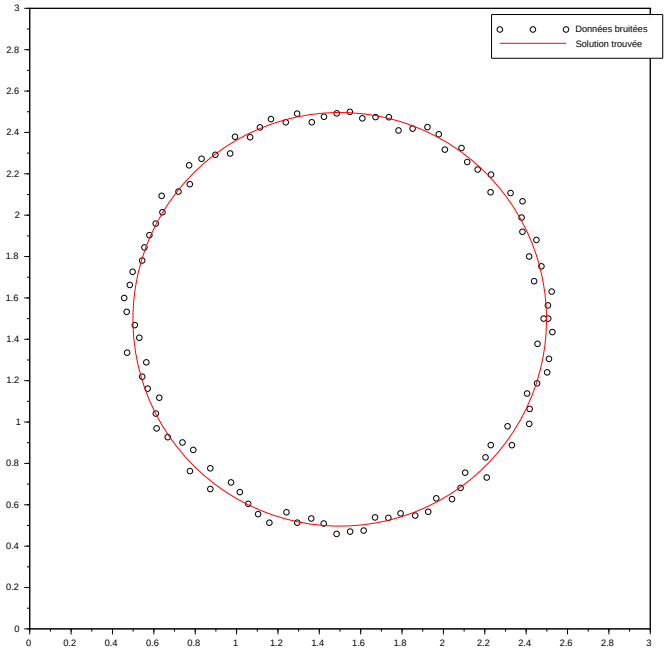


FIGURE V.3 – Régression d'un cercle

On obtient le cercle de centre $(a;b) = (1.498701;1.496565)$ et de rayon $R = 0.9999299$. On est donc très proche de la solution malgré le bruit, témoignant ainsi de l'efficacité de la méthode.

III) Problèmes non-linéaires

1) Erreurs possibles

Prenons comme example le modèle non-linéaire suivant :

$$y = f(x,\theta) = e^{\theta_0+\theta_1 x}$$

On pourrait penser qu'utiliser le logarithme permettrait de se ramener au problème linéaire :

$$S_{log} = \sum_{i=1}^{n} \left(\ln y_i - (\theta_0 + \theta_1 x)\right)^2 \tag{V.12}$$

Cependant cela n'est pas correct car si $y_i - f(x_i,\theta)$ a une distribution normale, ce n'est pas le cas pour $\log y_i - \log f(x_i,\theta)$.

On pourrait aussi essayer de calculer le gradient de S et l'annuler avec la méthode de Newton mais il faudrait calculer la jacobienne de $r = y_i - f(x,\theta) - y$ et cela n'assurerait pas un minimum mais donnerait peut-être un maximum ou un point selle.

Une solution viable est la suivante, on pose le développement limité de θ en θ_k :

$$r(\theta) = r(\theta_k) + J_r(\theta)(\theta - \theta_k) + \|\theta - \theta_k\|\epsilon(\theta - \theta_k)$$

Ainsi, trouver θ_{k+1} minimisant $S_k(\theta) = \|r(\theta_k) + J_r(\theta_k)(\theta - \theta_k)\|^2$ est un problème des moindres carrés linéaires que l'on peut résoudre avec la méthode de Newton : .

Ainsi pour résoudre des problèmes de moindre carré non-linéaires nous avons les méthodes suivantes, qui sont aussi valables pour dans le cas linéaire mais plus lourdes.

2) Méthode du gradient

La méthode du gradient est une méthode de descente : on choisit une direction $d_k = -\nabla f(x_k)$ et un pas p_k tel que $x_{k+1} = x_k + p_k \times d_k$ 'descende' par rapport à x_k, c'est-à-dire $f(x_{k+1}) < f(x_k)$.

Concernant le pas, on peut choisir un pas p_k constant ou bien variable selon x_k.

Un pas constant pose divers problèmes : s'il est mal choisi la méthode converge très lentement (pas trop petit) ou bien diverge (trop grand). En revanche nous pouvons pour chaque itération calculer un pas optimal.

Algorithme V.1 : Méthode du gradient

Data : $x_0 \in \mathbb{R}^n$ donné
Result : $\hat{x} \in \mathbb{R}^n$ minimum local de f approchée à ϵ près
while $(\|f(x_k)\| > \epsilon$ *et* $J_f(x_k)$ *inversible)* **do**
  $x_{k+1} = x_k - \rho_k \times \nabla f(x_k)$
end while

On a l'implémentation Scilab suivante :

```
function [x, k] = methodeGradient(x, f, gradf, pas, ITE_MAX, EPS, plotIsoCurve)
    if plotIsoCurve==1 then
        exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\trace_iso.sce");
        plotIso(x);
    end
    grad = gradf(x);
    gradIni = norm(grad);

    for k=1:ITE_MAX
        if norm(grad) < gradIni*EPS then
            break;
        end

        p = pas(x, grad);
        oldx = x;
        x = x - p*grad;

        if plotIsoCurve==1 then
            plotIso(x);
        end
        plot([oldx(1) x(1)], [oldx(2), x(2)], 'r', 'linewidth',2);

        grad = gradf(x);
    end
endfunction
```

Code Source V.3 – Méthode du Gradient

Nous allons utiliser cette méthode sur la forme quadratique suivante :

$$f(x) = \frac{1}{2}x^T Ax - b^T x \tag{V.13}$$

avec A une matrice symétrique définie positive. On choisit :

$$A = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} \qquad b = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

```scilab
clear; scf(0); clf;
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\2-Methode_Gradient.sce");
function y = fQuad(x)            // Forme quadratique à minimiser
    y = .5*x'*A*x-b'*x;
endfunction
function grad = gradQuad(x)        // Trouvé avec développement f(x+h)
    grad = A*x - b;
endfunction
function p = pasQuad(x, grad)    // Pas Optimal pour forme quadratique
    p = grad'*grad/(grad'*A*grad);
endfunction

// Inputs
x0 = [1; 0];
A = [2 -1; -1 2];
b = [1; 1];
global A, b;

clf;
[soluce, ite] = methodeGradient(x0, fQuad, gradQuad, pasQuad, 100, 10^-8, 1);
ecart = norm(soluce-[1; 1]);
printf("Après %d itérations on obtient un écart de %f", ite, ecart);
```

Code Source V.4 – Méthode du Gradient pour une fonction quadratique

Nous choisissons d'abord un pas fixe $p = 0.2$ L'algorithme converge alors en 79 itérations.

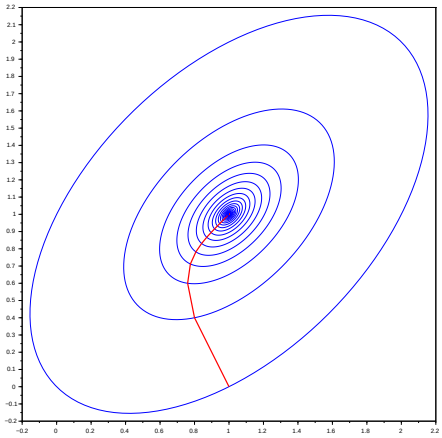


FIGURE V.4 – Méthode du Gradient pour une forme quadratique avec un pas fixe

Dans le cas de f sous forme quadratique, le pas optimal est le suivant :

$$p_k = \frac{\| \nabla f \|^2}{\nabla f' \times A \times \nabla f} \tag{V.14}$$

Avec ce pas optimal, on s'aperçoit que la convergence est plus rapide avec désormais 18 itérations.

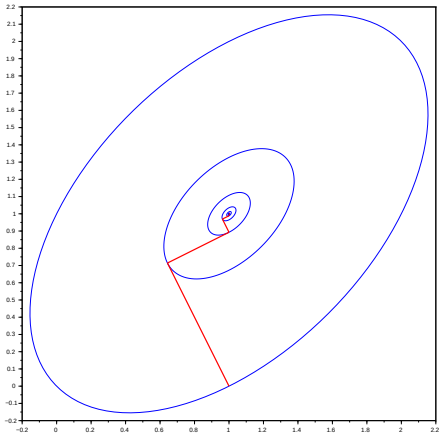


FIGURE V.5 – Méthode du Gradient pour une forme quadratique avec un pas optimal

Cependant cette méthode a des limites. Prenons la fonction de Rosenbrock :

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2; \tag{V.15}$$

Cette fonction particulière est souvent utilisée pour mettre à l'épreuve les algorithmes d'optimisation. On s'aperçoit facilement qu'elle admet un minimum global en $(1,1)$.

Nous allons appliquer la méthode du gradient. Concernant le pas optimal pour cette fonction, on utilise la méthode de la section dorée fournie par Stéphane Mottelet qui fonctionne quelque soit la fonction à minimiser.

```
clear; scf(0); clf;
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\2-Methode_Gradient.sce");
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\rosen.sce");
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\pasOptimal.sci");

function y = fRsbrck(x)
    y = (1-x(1))^2 + 100*(x(2)-x(1)^2)^2;
endfunction
function grad = gradRsbrck(x)
    a = -2*(1-x(1)) -400*x(1)*(x(2) - (x(1))^2);
    b = 200*(x(2) - (x(1))^2);
    grad = [a; b];
endfunction
function p = pasOpti(x,grad)    // Pas Optimal pour fonction quelconque
    p = pasOptimal(x, -grad, list(fRsbrck));
endfunction

x0 = [-0.5; 0.5];
[soluce, ite] = methodeGradient(x0, fRsbrck, gradRsbrck, pasOpti, 10^-8, 1000, 0);
ecart = norm(soluce-[1;1]);
printf("Après %d itérations on obtient un écart de %f", ite, ecart);
```

Code Source V.5 – Méthode du Gradient pour la fonction de Rosenbrock

Ici on ne trace pas les courbes iso-valeur à chaque itération mais l'allure de la surface (plus les couleurs tendent vers le bleu, plus la fonction est "basse").

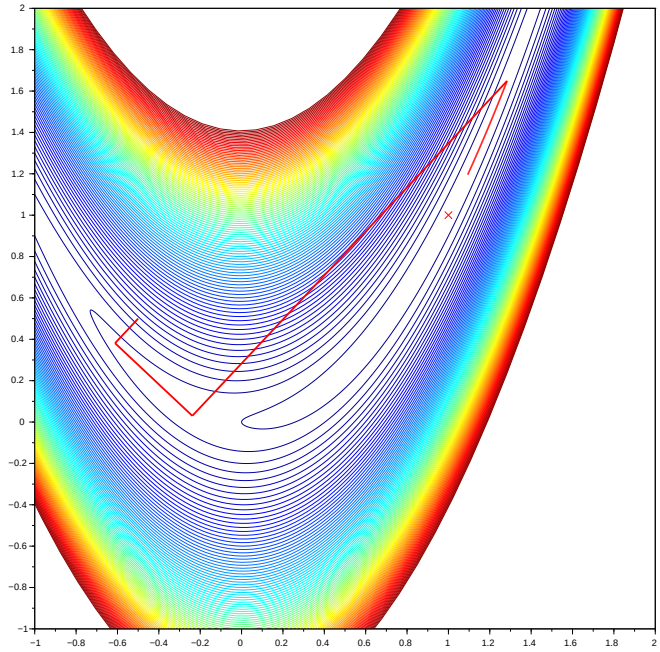


FIGURE V.6 – Méthode du Gradient sur la fonction de Rosenbrock

Après 1000 itérations, la solution n'a toujours pas convergé et on obtient un écart en norme de 0.2173970 par rapport à la solution $(1,1)$.

On s'aperçoit que les itérations successives s'approche très lentement du minimum et restent bloquées dans une "vallée". On trouve donc une limite à la méthode du gradient, nous avons alors d'autres méthodes plus efficaces.

3) Méthode de Gauss-Newton

L'idée de cette méthode est la suivante. On part de la méthode des moindre carré (V.4) : $S(\theta) = \|r(\theta)\|^2$ où nous cherchons à minimiser S. Effectuons le développement limité de $r(\theta)$ à l'itération θ_k :

$$r(\theta) = r(\theta_k) + J_r(\theta_k)(\theta - \theta_k) + \|\theta - \theta_k\|\epsilon(\theta - \theta_k)$$

Ainsi il suffit de trouver θ_{k+1} minimisant :

$$S_k(\theta) = \|r(\theta_k) + J_r(\theta_k)(\theta - \theta_k)\|^2 = \|r_k(\theta)\|^2$$

qui est un problème de moindre carré linéaire. On a donc :

$$\begin{aligned}\theta_{k+1} &= \theta_k - J_r(\theta_k)^{-1} r(\theta_k) \\ &= \theta_k - \left(J_r(\theta_k)^T J_r(\theta_k)\right)^{-1} J_r(\theta_k)^T r(\theta_k) \\ &= \theta_k - \frac{1}{2}\left(J_r(\theta_k)^T J_r(\theta_k)\right)^{-1} \nabla S(\theta_k)\end{aligned}$$

Le problème est quand le rang de la jacobienne en θ_k est dégénéré. Pour remédier à cela, on empêche que θ_{k+1} soit trop éloigné de θ_k La méthode suivante permet d'éviter ce problème.

4) Méthode de Levenberg-Marquardt

Nous avons la méthode de Levenberg-Marquardt :

$$\theta_{k+1} = \theta_k - \frac{1}{2}\left(J_r(\theta_k)^T J_r(\theta_k) + \lambda I\right)^{-1} \nabla S(\theta_k) \tag{V.16}$$

On peut choisir de régler cette méthode entre :

— la rapidité quand $\lambda \to 0$: on se rapproche de la méthode de Gauss-Newton.

— la sureté $\lambda \to \infty$: on est plus proche de la méthode du gradient.

Scilab implémente aussi l'outil `lsqrsolve` qui utilise cette méthode pour résoudre un problème non-linéaire au sens des moindres carrés.

Implémentons la sous Scilab de la manière suivante :

```
function [x, i] = methodeLM(x, gradf, Jf, lambda, ITE_MAX, EPS)
    J = Jf(x);
    grad = gradf(x)

    [a,b] = size(J);
    gradIni = norm(grad);

    for i = 1:ITE_MAX
        if norm(grad) < gradIni*EPS then
            break;
        end

        oldx = x;
        x = x - (J'*J + lambda*eye(a,a)) \ (J'*grad);

        plot([oldx(1) x(1)], [oldx(2), x(2)], 'r', 'linewidth',2);
        J = Jf(x);
        grad = gradf(x);
    end
endfunction
```

Code Source V.6 – Méthode de Levenberg-Marquardt

Testons la sur la fonction de Rosenbrock (V.15). On a le code suivant :

```
clear; scf(0); clf;
function y = gradf(x)
    y = [1-x(1); 10*(x(2)-x(1)^2)];
endfunction
function J = Jf(x)
    J = [-1, 0; -20*x(1), 10];
endfunction

exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\3-Methode_LM.sce");
x0 = [0; 1];
lambda =[0, .25, .75, 1];
for i=1:4
    subplot(1,4,i);
    exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD5-Optimisation\rosen.sce");
    [soluce, ite] = methodeLM(x0, gradf, Jf, lambda(i), 100, 10^-8);
    title(sprintf("$\\huge\\lambda=%1.2f \\text{ et convergence en }%d \\text{ itérations}$",
     ↪  lambda(i),ite));
    printf("Avec lambda = %1.2f, la méthode converge après %d itérations.\n", lambda(i), ite);
end
```

Code Source V.7 – Méthode de Levenberg-Marquardt pour la fonction de Rosenbrock

On applique l'algorithme pour plusieurs valeur de λ afin de voir l'impact de ce paramètre sur la vitesse de convergence de la méthode. On obtient les résultats suivants :

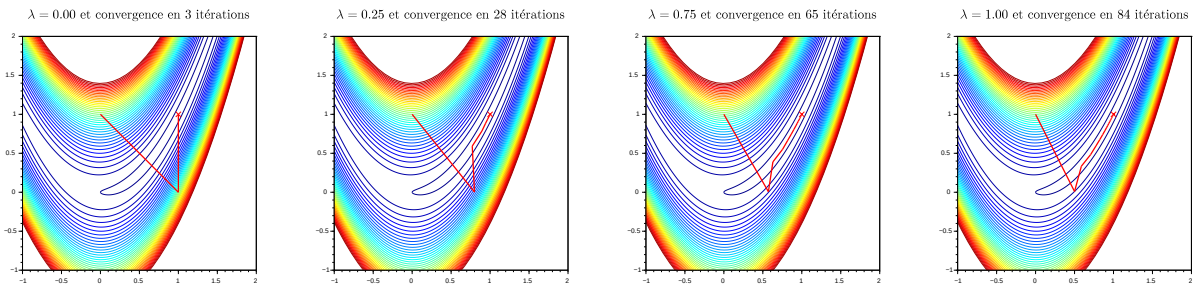


FIGURE V.7 – Méthode de Levenberg-Marquardt avec différentes valeurs de λ

On remarque bien que plus λ est petit, plus la convergence est rapide, allant même jusqu'à atteindre le minimum de la fonction de Rosenbrock en 3 itérations ! Cette méthode est donc bien plus efficace que celle du gradient.

CHAPITRE VI

Séries de Fourier

Dans ce chapitre nous aborderons les séries de Fourier. Cet outil permet d'approcher n'importe quelle fonction périodique par une somme de fonctions sinusoïdales. Historiquement le mathématicien français Josepeh Fourier (1768-1830) a construit ce concept en étudiant la propagation de la chaleur. Commençons par étudier le même problème que lui au travers de son expérience suivante.

I) L'expérience de Fourier

i) Description et modélisation

L'expérience de Fourier sur la propagation de la chaleur dans un matériau est un des exemples historiques d'équation aux dérivées partielles.

Un anneau métallique est chauffé à blanc sur une partie, puis il est plongé dans du sable, un matériau isolant. Nous cherchons à étudier ce qu'il se passe entre la temps initial et l'infini où la chaleur devrait s'être diffusée dans tout l'anneau.

On suppose le rayon de l'anneau suffisamment grand devant la taille de la section afin de n'avoir qu'une propagation longitudinale selon l'anneau. On repère alors la progression de la chaleur selon l'angle θ avec $\theta_0 = 0$ l'endroit où l'anneau est chauffé initialement.

On note $u(\theta, t)$ la température de l'anneau à l'angle $\theta \in]-\pi;\pi[$ à l'instant $t \geq 0$. On a l'équation aux dérivées partielles suivantes :

$$c \times \rho \times \frac{\partial u}{\partial t} - \lambda \frac{\partial^2 u}{\partial \theta^2} = 0$$

où c est la capacité calorifique, ρ la masse linéique et λ la conductivité thermique.

En posant $d = \frac{\lambda}{c\rho}$ on a :

$$\frac{\partial u}{\partial t} - d \frac{\partial^2 u}{\partial \theta^2} = 0 \tag{VI.1}$$

Pour des raisons pratiques, on supppose une symétrie de la chaleur dans l'anneau :

$$\forall \theta \in [-\pi;\pi] : \ u(\theta, t) = u(-\theta, t)$$

u est alors paire et on travaillera donc sur l'intervalle $I = [0;\pi]$. On suppose aussi $u(\theta, t)$ dérivable par rapport à θ.

ii) Conditions

On ajoute les conditions aux limites : $\forall\, t \geq 0$

$$\frac{\partial u}{\partial \theta}(0, t) = \frac{\partial u}{\partial \theta}(\pi, t) = 0 \tag{VI.2}$$

et la condition initiale :

$$u(\theta, 0) = f(\theta) \qquad \forall \theta \in I \tag{VI.3}$$

iii) Construction de u_n

On cherche u sous la forme à variables séparées : $u(\theta,t) = g(\theta)h(t)$.
On a :

$$\begin{aligned}(\text{VI.1}) &\implies g(\theta)h'(t) - dg''(\theta)h(t) = 0 \\ &\iff \frac{g''(\theta)}{g(\theta)} = \frac{h'(t)}{d \times h(t)} = C \end{aligned} \tag{VI.4}$$

Les deux parts de l'équation (VI.4) dépendent de variables différentes donc elles sont égales à la même constante $C \in \mathbb{R}$. On trouve alors h et g :

$$\begin{aligned}(\text{VI.4}) &\implies h'(t) = Cdh(t) \\ &\implies h(t) = \gamma e^{Cdt} \qquad \gamma \in \mathbb{R}\end{aligned}$$

De plus $d > 0$ donc si $C > 0$ alors $h(t) \xrightarrow[t\to\infty]{} \infty$. Or cela n'est pas plausible physiquement, ainsi $C \leq 0$.

$$(\text{VI.4}) \implies g''(\theta) - Cg(\theta) = 0$$

On pose $C = -\omega^2$

$$g''(\theta) + \omega^2 g(\theta) = 0 \implies g(\theta) = \alpha\cos(\omega\theta) + \beta\sin(\omega\theta)$$

Comme u (et donc g) est paire par rapport à θ, $\beta = 0$. De plus on a :

$$\begin{aligned}(\text{VI.2}) &\implies -\alpha\omega\sin(\omega\pi) \times ke^{Cdt} = 0 \\ &\implies \sin(\omega\pi) = 0 \\ &\implies \lambda = n \in \mathbb{N} \\ g(\theta) &= \alpha\cos(n\theta) \qquad \forall n \in \mathbb{N}\end{aligned}$$

Ainsi :

$$u_n(\theta,t) = \alpha_n \cos(n\theta)e^{-dn^2t} \tag{VI.5}$$

est solution de (VI.1) et des deux conditions limites (VI.2) mais pas encore de la condition initiale (VI.3).

iv) Construction de la Série

Toute combinaison linéaire de u_n est solution. On pose alors :

$$u(\theta,t) = \sum_{n\geq 0} \alpha_n \cos(n\theta)e^{-dn^2t} \tag{VI.6}$$

vérifiant les conditions (VI.2) et (VI.3) par construction à partir (VI.5).

$$(\text{VI.3}) \implies u(\theta,0) = \sum_{n\geq 0} \alpha_n \cos(n\theta) = f(\theta) \quad \forall\theta \in [0.\pi]$$

D'une part, on a $\forall n > 0$:

$$\begin{aligned}\int_0^\pi \cos(n\theta)\,\mathrm{d}\theta = 0 \implies \int_0^\pi \alpha_n\cos(n\theta)\,\mathrm{d}\theta &= \int_0^\pi f(\theta)\,\mathrm{d}\theta \\ \iff \pi\alpha_0 &= \int_0^\pi f(\theta)\,\mathrm{d}\theta \\ \iff \alpha_0 &= \frac{1}{\pi}\int_0^\pi f(\theta)\,\mathrm{d}\theta\end{aligned}$$

Ainsi α_0 est la chaleur moyenne.

D'autre part :

$$\int_0^\pi cos(n\theta)cos(k\theta)\,\mathrm{d}\theta = 0 \quad \text{si } k \neq n \quad \text{et} \quad \int_0^\pi cos^2(n\theta)\,\mathrm{d}\theta = \frac{\pi}{2}$$

Donc :

$$\begin{aligned}\int_0^\pi \sum_{n\geq 0} cos(n\theta)cos(k\theta)\,\mathrm{d}\theta = \int_0^\pi f(\theta)cos(k\theta)\,\mathrm{d}\theta &\iff \alpha_k\int_0^\pi cos^2(k\theta)\,\mathrm{d}\theta = \int_0^\pi f(\theta)cos(k\theta)\,\mathrm{d}\theta \\ &\iff \alpha_k = \frac{2}{\pi}\int_0^\pi f(\theta)cos(k\theta)\,\mathrm{d}\theta\end{aligned}$$

On a ainsi entièrement défini la série de Fourier (VI.6).

Si on néglige les termes dont $n \geq 2$ (ils sont négligeables car alors e^{-dn^2t} est proche de zéro), $u(\theta,t)$ peut être approché par :

$$u(\theta,t) \simeq \alpha_0 + \alpha_1\cos\theta e^{-dt}$$

II) Séries de Fourier

DÉFINITION VI.1 : Polynôme trigonométrique
Toute fonction de la forme

$$P_n(x) = \frac{a_0}{2} + \sum_{k=1}^{n} a_k \cos\left(\frac{2k\pi}{T}x\right) + b_k \sin\left(\frac{2k\pi}{T}x\right) \tag{VI.7}$$

avec T la période et la fréquence $\omega = \frac{2\pi}{T}$.

PROPRIÉTÉ VI.1 :
Si $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ sont décroissants et tendent vers 0 alors $P_n(x)$ converge pour tout x.

DÉFINITION VI.2 : Série trigonométrique
Toute fonction de la forme :

$$f(x) = \lim_{n\to\infty} P_n(x) \tag{VI.8}$$

On suppose que la série converge pour tout x.

DÉFINITION VI.3 : Convergence simple
On dit que P_n converge simplement vers f si :

$$\forall \epsilon > 0, \forall x \in [0;T], \exists N \in \mathbb{N} \text{ tel que } \forall n \in \mathbb{N} : n > N \implies |P_n(x) - f(x)| < \epsilon$$

DÉFINITION VI.4 : Convergence uniforme
On dit que P_n converge uniformément vers f si :

$$\forall \epsilon > 0, \exists N \in \mathbb{N} \text{ tel que } \forall x \in [0;T], \forall n \in \mathbb{N} : n > N \implies |P_n(x) - f(x)| < \epsilon$$

Si P_n converge vers f uniformément alors on peut montrer que :

$$a_n = \frac{2}{T}\int_0^T f(x)\cos(n\omega x)\,\mathrm{d}x$$
$$b_n = \frac{2}{T}\int_0^T f(x)\sin(n\omega x)\,\mathrm{d}x$$

avec la pulsation $\omega = \frac{2\pi}{T}$. L'intégrale peut être centrée sur $[-\frac{T}{2};\frac{T}{2}]$, l'essentiel étant qu'elle soit de longueur T.

Définition VI.1 : Un point de discontinuité de première espèce a une limite à gauche et à droite.

$\sin(\frac{1}{x})$ possède une discontinuité de seconde espèce en 0 (pas de limites).

THÉORÈME VI.1 : Théorème de Dirichlet
Soit f une fonction périodique telle que :
— les discontinuités de f dans un période sont en nombre fini et de première espèce
— f admet une dérivée à gauche et à droite
Alors $S(x) = P_n(x)$ converge et on a :

$$S(x) = \begin{cases} \dfrac{f(x^+) + f(x^-)}{2} & \text{si } f \text{ est discontinue en } x \\ f(x) & \text{sinon} \end{cases} \tag{VI.9}$$

avec $f(x^+)$ la limite à droite et $f(x^-)$ celle à gauche. La convergeance est uniforme sur tout intervalle où f est continue.

Pour calculer n'importe quel série de Fourier réelle dans Scilab, j'ai créé le code suivant :

```
// Coefficients an et bn avec en paramètres x, n, T
function y = a(x,n,T)
    // y = ....
endfunction
function y = b(x,n,T)
    // y = ....
endfunction

function S = serie(a0, a, b, x, T, nbIte)
    S = zeros(1,length(x)) + a0/2;

    w = 2*%pi/T;
    for n=1:nbIte
        if (a == 0) then
            an = 0;
        else
            an = a(x,n,T);
        end
        if (b == 0) then
            bn =  0;
        else
            bn = b(x,n,T)
        end

        S = S + an*cos(n*w*x) + bn*sin(n*w*x);
    end
endfunction

// Paramètres
NB_POINTS = 400;
NB_ITE = 500;
XMIN = 0;
XMAX = 1/2;
a0 = 1/3;
T = 1/2;

x = linspace(XMIN,XMAX, NB_POINTS);
S = serie(a0, a, b, x, T, NB_ITE);
clf; plot(x, S);
```

Code Source VI.1 – Exemple du calcul d'une Série de Fourier

Les différents paramètres sont :
— `a0`
— `a(x,n,T)` et `b(x,n,T)` les fonctions permettant de calculer les coefficients a_n et b_n, on peut aussi passer 0 si le coefficent est nul afin d'éviter des appels inutiles
— `x` le vecteur des abscisses discrétisé pour lesquelles on calcule la série.
— `T` la période
— `NB_ITE` le nombre d'itérations n pour calculer la série
Cela permet de calculer n'importe quelle série simplement en renseignant ces quelques paramètres.

Dans le cas complexe, la série est définie de la façon suivante :

$$S(x) = \sum_{n\in\mathbb{Z}} c_n e^{i\times n\omega x} \tag{VI.10}$$

Avec les coefficients complexes c_n tels que :

$$c_n = \frac{1}{T}\int_{-T/2}^{T/2} f(t)e^{-i\times n\omega t}\,\mathrm{d}t \tag{VI.11}$$

III) Applications

1) Phénomène de Gibbs

Prenons la fonction f de période 2π telle que :

$$f(x) = \begin{cases} -1 & \forall x \in [-\pi;0[\\ 1 & \forall x \in [0;\pi[\end{cases} \tag{VI.12}$$

On a les termes a_n et b_n :

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(nx)\,\mathrm{d}x = 0 \quad \text{car impaire} \tag{VI.13}$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(nx)\,\mathrm{d}x = \frac{1}{\pi}\int_{0}^{\pi} \sin(nx)\,\mathrm{d}x = \frac{2}{n\pi}(1-\cos(n\pi)) = \frac{2}{n\pi}(1-(-1)^n) \tag{VI.14}$$

Ainsi seules les termes avec n impair ne sont pas nuls, on pose $n = 2p+1$ et on obtien la série de Fourier correspondante :

$$S(x) = \frac{4}{\pi}\sum_{p\geq 0}\frac{\sin((2p+1)x)}{2p+1} \tag{VI.15}$$

Comme le prévoit le théorème de Dirichlet (VI.1), on n'a pas égalité entre $f(x)$ et $S(x)$ aux points de discontinuité (ici de la forme $x = k\pi$ avec k impair) mais bien une moyenne des deux limites, ici $\frac{-\pi+\pi}{2} = 0$. Que se passe-t-il donc à ces points de discontinuité?

Découvrons le avec le code suivant :

```
function y = carre(x)
for i=1:length(x)
    if x(i) < 0 then
        if abs(modulo(x(i)-%pi,2*%pi))<=%pi then
           y(i) = 1
       else
          y(i) = -1
        end
    else
        if modulo(x(i),2*%pi)<=%pi then
           y(i) = 1
       else
          y(i) = -1
        end
    end
end
endfunction

NB_PERIODE = 2;
NB_POINTS = 400;
NB_ITE = 50;

x = linspace(-NB_PERIODE*%pi,NB_PERIODE*%pi, NB_POINTS);
S = zeros(1,length(x));

for p=0:NB_ITE
    n = 2*p+1;
    S = S + 4*sin(n*x)/(n*%pi);
end

clf;
plot(x,carre(x)','k', x,S,'r');
```

Code Source VI.2 – Phénomène de Gibbs

Ici nous allons calculer la série de Fourier d'un signal carré de période $T = 2\pi$ défini par :

$$\forall x \in [-\pi;0[: f(x) = -1$$
$$\forall x \in [0;\pi] : f(x) = 1$$

Nous obtenons le rendu suivant :

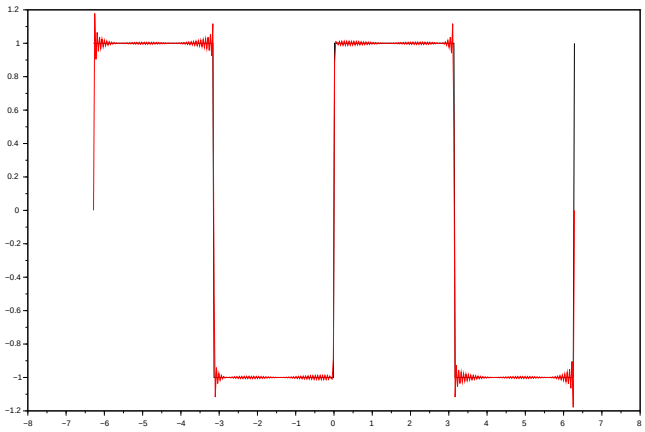


FIGURE VI.1 – Phénomène de Gibbs

On observe des oscillations aux voisinage des points de discontinuité. Ces erreurs d'approximation sont dues au fait que S ne converge pas uniformément vers f aux points de discontinuité. Ceci est appelé le phénomène de Gibbs.

2) Régularité et décroissance des coefficients

Nous allons ici comparer trois séries de Fourier approchant la fonction f suivante sur $[0;\frac{1}{2}]$:

$$f(x) = x(x-1) \tag{VI.16}$$

Nous avons les trois fonctions suivantes qui vont nous servir pour les séries de Fourier :
— f_1 de période $T_1 = \frac{1}{2}$: $\forall x \in [0;\frac{1}{2}]$: $f_1(x) = f(x)$
— f_1 paire de période $T_2 = 1$: $\forall x \in [0;\frac{1}{2}]$: $f_2(x) = f(x)$
— f_1 impaire de période $T_3 = 2$: $\forall x \in [0;1]$: $f_3(x) = f_2(x)$

On calcule les différents coefficents a_n et b_n $\forall n > 0$:

— Pour $f_1 : \omega = 4\pi$

$$\left|\begin{array}{l} a_n = 4\displaystyle\int_0^{\frac{1}{2}} x(x-1)\cos(4\pi nx)\,\mathrm{d}x = \ldots = \frac{1}{4\pi^2 n^2} \\ b_n = 4\displaystyle\int_0^{\frac{1}{2}} x(x-1)\sin(4\pi nx)\,\mathrm{d}x = \ldots = \frac{1}{4\pi n} \\ a_0 = \dfrac{1}{3} \end{array}\right.$$

La vitesse de décroissance des coefficients de f_1 est 1 (on prend le minimum des deux coefficients). Cela signifie que la fonction n'est pas régulière en tous points, en effet au point de discontinuités on a $S \neq f_1$ d'après le théorème VI.1. Cela est dû au fait que la fonction f_1 est discontinue.

— Pour $f_2 : \omega = 2\pi$

$$\left|\begin{array}{ll} a_n = 2\displaystyle\int_{-\frac{1}{2}}^{\frac{1}{2}} f_2(x)\cos(2\pi nx)\,\mathrm{d}x = 4\int_0^{\frac{1}{2}} x(x-1)\cos(2\pi nx)\,\mathrm{d}x = \ldots = \frac{1}{\pi^2 n^2} & \\ b_n = 0 & \text{car } f_2 \text{ est paire??} \\ a_0 = \dfrac{1}{3} & \end{array}\right.$$

La vitesse de décroissance des coefficients de f_2 est 2. On a donc $S = f_2$ car la fonction est continue.

— Pour $f_2 : \omega = \pi$

$$\left|\begin{array}{ll} a_n = 0 & \text{car } f_2 \text{ est impaire??} \\ b_n = \ldots = 4\dfrac{(-1)^n - 1}{\pi^3 n^3} & \\ a_0 = 0 & \end{array}\right.$$

La vitesse de décroissance des coefficients de f_3 est 3. Ici, en plus d'être continue et $S = f_3$, la fonction est également continûment dérivable.

Ainsi le choix de la fonction périodique f_i pour la série impacte la convergence de celle-ci vers f. Plus les coefficients décroissent vite, plus la convergence est élevée.

Vérfions cela expérimentalement avec Scilab :

```
exec("D:\Documents\Cours\TC04 - Printemps 2017\MT94\Scilab\TD6-Equations_Differentielles_Partielles\0-Calcul_Serie.sce");
// Différents coefficients des fonctions fi
function y = a1(x,n,T)
    y = 1/(4*%pi^2 * n^2);
endfunction
function y = b1(x,n,T)
    y = 1/(4*%pi*n);
endfunction

function y = a2(x,n,T)
    y = 1/(%pi^2 *n^2);
endfunction

function y = b3(x,n,T)
    y = 4*((-1)^n -1)/(%pi^3 * n^3);
endfunction

NB_POINTS = 400;
NB_ITE = 5;
XMIN = 0;
XMAX = 1/2;

x = linspace(XMIN,XMAX, NB_POINTS);
f = x.*(x-1);

// Calcul des séries
S1 = serie(-1/3, a1, b1, x, 1/2, NB_ITE);
S2 = serie(-1/3, a2, 0, x, 1, NB_ITE);
S3 = serie(0, 0, b3, x, 2, NB_ITE);

// Calcul des écarts
ecarts = zeros(3, NB_POINTS);
ecarts(1,:) = abs(f-S1);
ecarts(2,:) = abs(f-S2);
ecarts(3,:) = abs(f-S3);

clf; subplot(2,1,1);
plot(x', [S1; S2; S3; f]');
legend("Série f1", "Série f2", "Série f3", "Fonction originale");
title("Séries");

subplot(2,1,2);
plot(x', ecarts');
legend("Série f1", "Série f2", "Série f3");
title("Écarts entre la fonction f et les séries");

erreurMoy = [
    mean(ecarts(1,:));
    mean(ecarts(2,:));
    mean(ecarts(3,:));
]
disp(erreurMoy, "L erreur moyenne de chaque série est : ");
```

Code Source VI.3 – Comparaison de séries

On calcule d'abord les trois séries avec la fonction VI.1 pour $n = 400$ points, puis on affiche les séries et la fonction d'origine f sur le même graphe. Ensuite on calcule l'écart entre la fonction f et les différentes séries, on l'affiche.

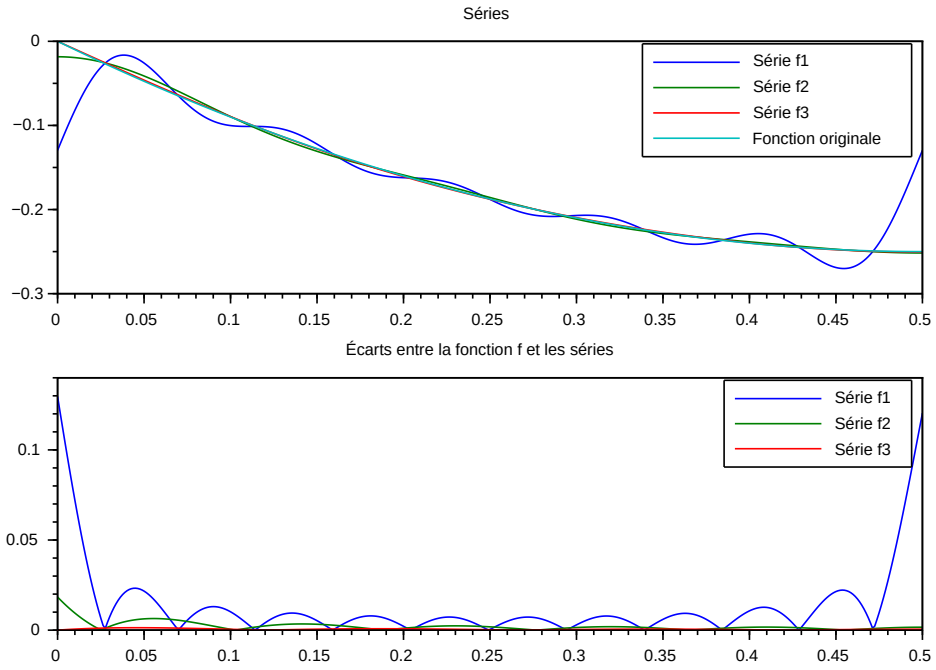


FIGURE VI.2 – Différences entre les séries

On remarque sur les deux graphiques que S_3 est bien plus proche de f que S_2 et encore plus S_1. Cela confirme donc bien que plus les coefficients décroissent, plus la série converge et est proche de f.

3) Propagation de la chaleur

Nous allons simuler ici l'expérience de Fourier. Nous prenons $d = 1$, la fonction de départ $f(\theta)$ est définie par $\lambda \in]0;\pi[$:

$$f(\theta) = \begin{cases} 1 & \text{si } x \in [\pi - \lambda; \pi + \lambda] \\ 0 & \text{sinon} \end{cases}$$

On prend ici $\lambda = \frac{\pi}{2}$.

Nous calculons les coefficients de la série de Fourier correspondante :

$$T = 2\pi \implies \omega = \frac{2\pi}{T} = 1$$

f est paire donc $b_n = 0$

$$\begin{aligned} a_n &= \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(n\omega x)\,\mathrm{d}x = \frac{2}{\pi}\int_{0}^{\pi} f(x)\cos(nx)\,\mathrm{d}x \\ &= \frac{2}{\pi}\int_{\pi-\lambda}^{\pi} \cos(nx)\,\mathrm{d}x = \frac{2}{n\pi}\left(\sin(n\pi) - \sin(n(\pi-\lambda))\right) \\ &= \frac{-2}{n\pi}\sin(n(\pi-\lambda)) \\ a_0 &= \frac{1}{2\pi}\int_{0}^{2\pi} f(x)\,\mathrm{d}x = 2\frac{\lambda}{\pi} \end{aligned}$$

Il suffit ensuite de calculer la série correspondante avec le programme suivant :

```
NB_POINTS = 50;         // Discrétisation de la fonction
NB_TEMPS = 100;         // Discrétisation du temps
NB_COLOR = 128;
MAX_TEMPS = 6;
x = linspace(0, 2*%pi, NB_POINTS);
t = linspace(0.01, MAX_TEMPS, NB_TEMPS);       // t >0 pour éviter
LAMBDA = %pi/2;
a0 = 2*LAMBDA/%pi;

// Calcul de la série de Fourier s
scf(0); clf;
s = zeros(NB_TEMPS, NB_POINTS);            //    Ligne par ligne on a un instant t, col par col un point
 ↪  => s(t,x)
for i =1:NB_TEMPS
    s(i,:) = a0/2;
    for n = 1:NB_POINTS
        an = 2/(n*%pi) * sin(n*(%pi - LAMBDA));
        // an = rand();        // TEST random pour vérifier la convergence
        s(i,:) = s(i,:) - an* cos(n*x)* exp(-n^2 * t(i));
    end

    // Animation vague
    T = t(i)*ones(1,NB_POINTS);
    sleep(20)
    param3d(x,T,s(i,:));
    set(gca(), 'auto_scale', 'off', 'data_bounds', [0,2*%pi, 0,MAX_TEMPS, 0,1]);
end
xtitle('', 'x', 't', 'u(x,t)');

// Affichage de l'évolution de la solution
scf(1); clf;
surf(x,t,s);
set(gca(), 'auto_scale', 'off', 'data_bounds', [0,2*%pi, 0,MAX_TEMPS, 0,1]);
set(gcf(), 'color_map', hotcolormap(128));
set(gce(), 'color_flag',3, 'color_mode', -1);
xtitle('', 'x', 't', 'u(x,t)');

// Animation de la diffusion de la chaleur dans un anneau
r = [1 1.5]; // r1 r2
[R, THETA] = meshgrid(r, x);
X = R.*cos(THETA);
Y = R.*sin(THETA);

scf(2); clf;
for k=1:NB_TEMPS
    Z = [s(k,:)' s(k,:)'];
    sleep(20)
    drawlater;
    clf;
    Color = floor((1-0)*(NB_COLOR-1)*Z)+1;
    // Transformation linéaire de Z avec la plage et la taille de la colormap, chaque valeur de C
     ↪  associée à une valeur de Z correspond à l'indice de la colormap
    surf(X,Y,Z,Color);
    set(gca(), 'view', '2d', 'isoview', 'on', 'auto_scale', 'off', 'data_bounds',
     ↪  [-1.5,1.5,-1.5,1.5,0,1]);
    set(gcf(), 'color_map', hotcolormap(NB_COLOR));
    set(gce(), 'color_flag', 3, 'color_mode', -1, 'cdata_mapping', 'direct');
    drawnow;
end
colorbar(0, 1);
```

Code Source VI.4 – Chaleur

La série n'est alors pas définie pour $t = 0$ car nous avons $u(\theta, 0) = f(\theta)$. Nous choisissons t_0 légèrement supérieur à 0 pour éviter le phénomène de Gibbs.

J'ai fait diverses visualisations animées mais sur la suivante nous permet de voir l'évolution de la chaleur en fonction du temps et de θ.

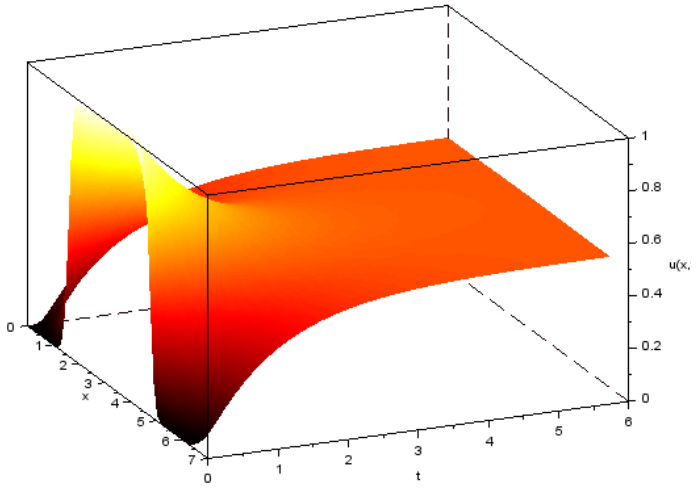
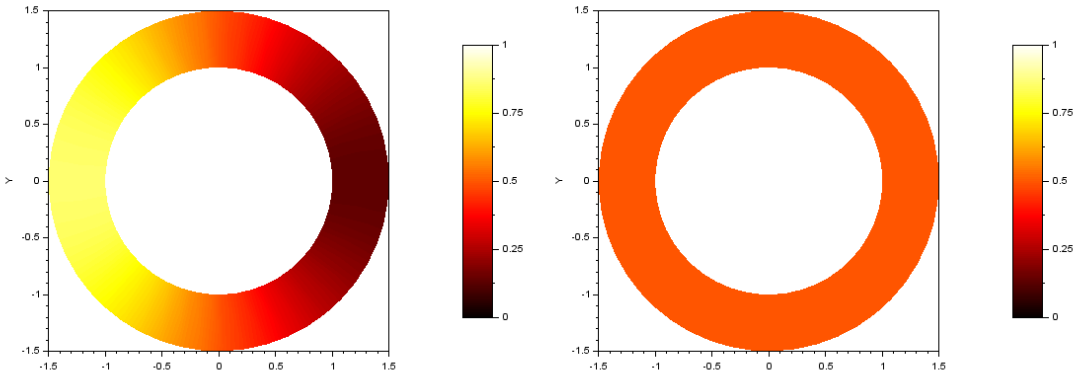


FIGURE VI.3 – Évolution de la chaleur

Les points blancs correspondent aux points les plus chauds et les noirs aux plus froids.

Nous pouvons aussi visualiser la propagation de la chaleur dans un anneau avec l'animation. Prenons deux temps on observons :



(a) Chaleur pour $t = 0.5$ (b) Chaleur pour $t = 6$

FIGURE VI.4 – Propagation de la chaleur dans l'anneau

Liste des codes sources Scilab

Table des figures