

$$x \cdot \cot g x = 1$$

$$2) a^2 = b^2 + c^2 - 2bc \cos \alpha$$

$$\cot \frac{\alpha}{2} = \frac{1 - \cos \alpha}{\sin \alpha}$$

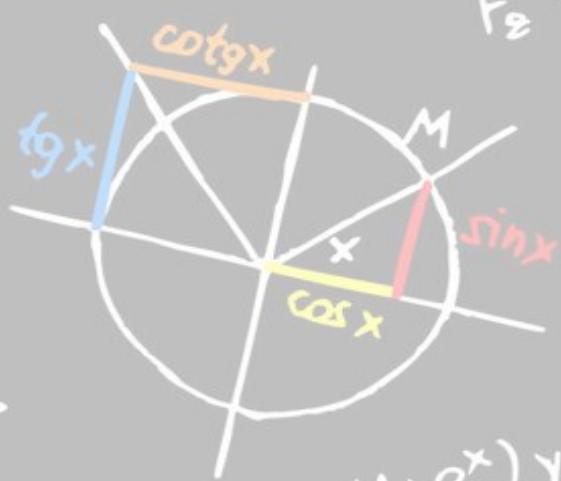
$$+z=1$$

$$y+z=\lambda$$

$$y+\lambda z=\lambda^2$$

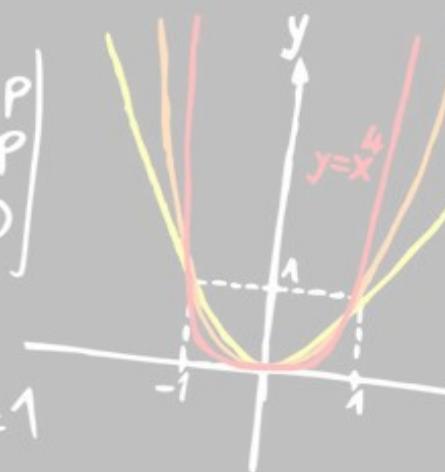
$$\tan \beta = \frac{c}{\sin \beta}$$

$$\sqrt{1+x^2}, x = \tan t$$



$$F_2 = 2 \times y^2 - 1 = 1$$

$$X_1 = \begin{pmatrix} 2\rho \\ -\rho \\ 0 \end{pmatrix}$$



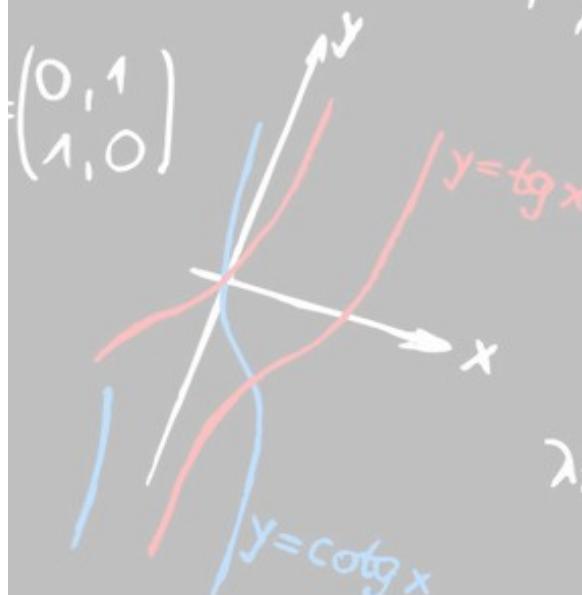
Cahier d'intégration MT94

$$X_1 = \begin{pmatrix} \alpha + \beta x_1 \\ \beta \\ 0 \end{pmatrix}$$

Marlow Justine

$$\cos 2x = \cos^2 x - \sin^2 x$$

$$(0,1)$$



$$\sin^2 x + \cos^2 x = 1$$

$$\begin{aligned} A+B+C &= \\ -3A-7B+2C &= \\ -18A+6B &= \end{aligned}$$

$$\int R(x, \sqrt{\frac{ax+b}{cx+d}}) dx$$

$$\frac{\sin x}{x} \leq$$

$$e_i 1]$$

$$\frac{2x}{x^2+2y^2} = 2 \quad z = \frac{1}{x} \arctan \frac{\sqrt{2}}{2}$$

$$\eta_1 = \lambda_1^2 - 3\lambda_1 + 1$$

$$x \neq 0$$

$$\sin(x+y) = \sin x \cos y + \cos x \sin y$$

$$x=0, y=1, z=2$$

$$y' - \frac{\sqrt{y}}{x+2} = 0 \quad ; \quad y(0) = 1$$

$$(1,0), \left(\frac{1}{2\sqrt{3}}, \frac{1}{4\sqrt{3}}\right)$$

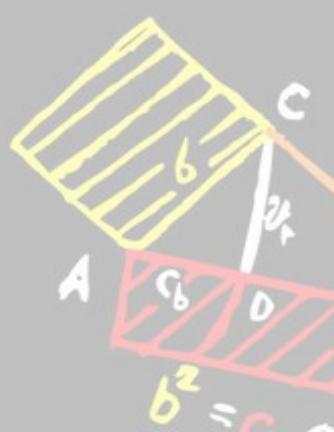


TABLE DES MATIÈRES

Table des matières	1
Table des figures	3
Table des codes	5
Résumé	7
1 Problèmes non linéaires I	8
1.1 Introduction	8
1.2 La dichotomie ou bisection	9
1.3 La méthode de point fixe	11
1.4 La méthode de Newton appliquée à $n = 1$	13
1.5 La méthode de la sécante	15
1.6 Pour $n > 1$: La méthode de Newton	17
1.6.1 Théorie	17
1.6.2 Fonction <i>fsolve</i>	17
1.6.3 Application 1 : le GPS	18
1.6.4 Application 2 : la cinématique inversée	21
2 Fractales	24
2.1 Introduction	24
2.2 Dimension de Hausdorff	25
2.3 Ensemble de Cantor	25
2.3.1 Théorie	25
2.3.2 Application dans <i>Scilab</i> (méthode récursive)	26
2.3.3 Application dans <i>Scilab</i> (méthode itérative)	27
2.4 Triangle de Sierpinski	29
2.4.1 Théorie	29
2.4.2 Application dans <i>Scilab</i> (méthode récursive)	30
2.4.3 Application dans <i>Scilab</i> (méthode itérative)	31
2.4.4 Variante : le tapis de Sierpinski	32
2.5 Flocon de neige de Von Koch	35
2.5.1 Théorie	35
2.5.2 Application dans <i>Scilab</i> (méthode récursive)	37
2.5.3 Application dans <i>Scilab</i> (méthode itérative)	38
2.6 Fougère de Barnsley	40
2.6.1 Théorie	40
2.6.2 Application dans <i>Scilab</i>	41

2.7 Ensemble de Julia	42
2.7.1 Théorie	42
2.7.2 Application dans <i>Scilab</i>	43
2.8 Ensemble de Mandelbrot	44
2.8.1 Théorie	44
2.8.2 Application dans <i>Scilab</i>	44
3 Équations différentielles	46
3.1 Introduction	46
3.2 Idée, stabilité, consistance et ordre d'un schéma numérique	47
3.2.1 Idée	47
3.2.2 Stabilité d'un schéma numérique	47
3.2.3 Consistance d'un schéma numérique	47
3.2.4 Ordre d'un schéma numérique	47
3.3 Schéma d'Euler	48
3.3.1 Théorie	48
3.3.2 Application dans <i>Scilab</i>	49
3.4 Schéma du point milieu	50
3.4.1 Théorie	50
3.4.2 Application dans <i>Scilab</i>	51
3.5 Schéma d'Euler-Cauchy	53
3.5.1 Théorie	53
3.5.2 Application dans <i>Scilab</i>	54
3.6 Schéma de Runge-Kutta	55
3.6.1 Théorie	55
3.6.2 Application dans <i>Scilab</i>	55
3.7 Application : mise en évidence de l'ordre de chaque schéma	57
3.8 Fonction <i>ode</i>	61
3.9 Application concrète : le pendule	61
3.10 Application concrète : les systèmes proies / prédateurs	63
3.11 Application concrète : la mécanique céleste	65
4 Valeurs propres	68
4.1 Introduction	68
4.2 Base théorique	69
4.2.1 Rappels d'algèbre linéaire	69
4.2.2 La décomposition SVD (<i>Singular Value Decomposition</i>)	69
4.2.3 Rang et noyau de $A \in \mathcal{M}_{m,n}(\mathbb{R})$	69
4.3 Application 1 : La compression d'images	70
4.3.1 Présentation du problème	70
4.3.2 Théorie de résolution (théorème et corollaire)	71
4.3.3 Résolution du problème	71
4.4 Application 2 : le <i>PageRank</i>	74
4.4.1 Présentation du problème	74
4.4.2 Théorie de résolution (modélisation et méthode de la puissance)	74
4.4.3 Résolution du problème	76
4.4.4 Extension de la méthode à d'autres problèmes	79
5 Problèmes non linéaires II	81
5.1 Introduction	81
5.2 Approche statistique	82
5.3 Problèmes des moindres carrés linéaires	82
5.3.1 Théorie	82
5.3.2 Application : Régression polynomiale avec validation	83
5.4 Problèmes des moindres carrés non linéaires	88

TABLE DES FIGURES

1.1	Erreurs en fonction de l'itération	10
1.2	Diminution relative de l'erreur en fonction de l'itération	10
1.3	Erreurs en fonction de l'itération	12
1.4	Diminution relative de l'erreur en fonction de l'itération	12
1.5	Erreurs en fonction de l'itération	14
1.6	Diminution relative de l'erreur en fonction de l'itération	14
1.7	Erreurs en fonction de l'itération	16
1.8	Diminution relative de l'erreur en fonction de l'itération	16
1.9	Définitions affichages de la cinématique inversée en fonction de n	23
2.1	Exemple de fractales : fractale de Mandelbrot	24
2.2	Ensemble de Cantor	25
2.3	Cantor 1845-1918	25
2.4	Figure pour $n=7$	27
2.5	Ensemble de Cantor itératif pour 1000 points	28
2.6	Triangle de Sierpinski	29
2.7	Sierpinski 1882-1924	29
2.8	Triangle de Sierpinski (récuratif) pour $n=5$	30
2.9	Triangle de Sierpinski itératif pour 1000 points	32
2.10	Tapis de Sierpinski	32
2.11	Tapis de Sierpinski (récuratif) pour $n=2$	33
2.12	Tapis de Sierpinski (itératif) pour $n=100000$ points	35
2.13	Flocon de neige de Von Koch	35
2.14	Von Koch 1870-1924	35
2.15	Courbe de Von Koch	37
2.16	Figure pour $n=4$	38
2.17	Flocon de Von Koch itératif pour 10000 points	39
2.18	Barnsley 1946-...	40
2.19	Fougère de Barnsley	40
2.20	Définitions affichages de la fougère en fonction de N	41
2.21	Julia 1893-1978	42
2.22	Ensemble de Julia (exemple parmi beaucoup d'autres)	42
2.23	Définitions affichages de l'ensemble de Julia en fonction de c	43
2.24	Ensemble de Mandelbrot	44
2.25	Mandelbrot 1924-2010	44
2.26	Ensemble de Mandelbrot	45
2.27	Définitions affichages de l'ensemble de Julia en fonction de c (dans ou hors la frontière de Mandelbrot)	45

3.1	Affichage (schéma d'Euler)	50
3.2	Affichage (schéma du point milieu)	52
3.3	Affichage (schéma d'Euler-Cauchy)	55
3.4	Affichage (schéma de Runge Kutta)	57
3.5	Affichage (ordres des différents schémas)	60
3.6	Schéma du pendule considéré	61
3.7	Superposition des solutions approchées pour différentes valeurs de θ_0	63
3.8	Évolution des proportions proies-prédateurs	65
3.9	Mécanique céleste : trajectoires des corps	67
4.1	Image originale Lena	70
4.2	Image originale Lena en niveaux de gris (obtenue grâce à la matrice)	70
4.3	Valeurs propres de Σ	72
4.4	Différentes compressions de l'image et erreur associée	72
4.5	Résultat du débruitage	73
4.6	PageRank : configuration 1	74
4.7	PageRank : configurations 2 à 5	76
5.1	Ensemble de données	84
5.2	Affichage des différentes approximations	85
5.3	Évolution de l'erreur en fonction du degré du polynôme	85
5.4	Affichage des différentes approximations sur T et V	87
5.5	Évolution de l'erreur sur T et sur V en fonction du degré du polynôme	88

TABLE DES CODES

1.1	Dichotomie	9
1.2	Point fixe	11
1.3	Newton (appliquée à $n = 1$)	13
1.4	Sécante	15
1.5	GPS (Méthode 1)	18
1.6	GPS (Méthode 2)	19
1.7	GPS (Méthode 3)	20
1.8	Cinématique inversée (code principal)	22
1.9	Cinématique inversée (code graphique)	22
2.1	Ensemble de Cantor (méthode récursive)	26
2.2	Ensemble de Cantor (méthode itérative)	28
2.3	Triangle de Sierpinski (méthode récursive)	30
2.4	Triangle de Sierpinski (méthode itérative)	31
2.5	Tapis de Sierpinski (méthode récursive)	33
2.6	Tapis de Sierpinski (méthode itérative)	34
2.7	Courbe de Von Koch (méthode récursive)	37
2.8	Flocon de Von Koch (méthode itérative)	39
2.9	Fougère de Barnsley	41
2.10	Ensemble de Julia	43
2.11	Ensemble de Mandelbrot	44
3.1	Schéma d'Euler	49
3.2	Schéma du point milieu	52
3.3	Schéma d'Euler-Cauchy	54
3.4	Schéma de Runge-Kutta	56
3.5	Ordre des schémas	58
3.6	Résolution du problème du pendule	62
3.7	Résolution du problème proies / prédateurs	64
3.8	Résolution du modèle de mécanique céleste	66
4.1	Lena : Récupération et affichage de l'image originale	70
4.2	Lena : Compression de l'image	71
4.3	Lena : Bruitage puis débruitage de l'image	73
4.4	Détermination du PageRank d'un ensemble de page internet	77
4.5	Détermination de l'importance d'un ville dans un réseau ferroviaire	79
5.1	Récupération des données	83
5.2	Approximation par différents polynômes	84

5.3 Approximation par différents polynômes et méthode de validation	86
---	----

RÉSUMÉ

Blabla résumé, MT94

CHAPITRE 1

PROBLÈMES NON LINÉAIRES I

1.1 Introduction

Les problèmes non linéaires constituent un ensemble de problèmes mathématiques qui sont, pour la plupart, insolubles de manière analytique. Ces problèmes peuvent en effet se ramener à la recherche des solutions de $f(x) = 0$. Or la recherche de racines devient problématique lorsque le degré du polynôme f augmente, impossible de déterminer des solutions analytiquement pour un degré supérieur ou égal à 4. Il existe donc des méthodes numériques afin d'approcher ces solutions. Nous allons étudier quelques une de ces méthodes au sein de ce chapitre.

Les trois premières méthodes étudiées, à savoir *la dichotomie*, *la méthode des points fixes* et *la méthode de Newton*, concernent les problèmes à une inconnue : $f : \mathbb{R} \rightarrow \mathbb{R}$. Nous verrons également l'application de *la méthode de la sécante*, qui se trouve être une méthode dérivée de *la méthode de Newton*. La dernière méthode étudiée, qui est également *la méthode de Newton*, permet de traiter des problèmes à n ($n > 1$) inconnues : $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$.

Pour chacune de ces méthodes, nous étudierons tout d'abord leur aspect théorique avant de les appliquer dans *Scilab*. Cette application suivra toujours la même démarche :

- Résoudre le problème par exécution du code
- Tracer la courbe de l'erreur en fonction de l'itération avec la commande :

```
--> plot(erreur, 'o')
```

- Tracer la courbe de régression linéaire avec la commande :

```
--> plot(log(erreur(1:k-1)), log(erreur(2:k)))
```

et connaître son coefficient directeur avec la commande :

```
--> reglin(log(erreur(1:k-1)), log(erreur(2:k)))
```

1.2 La dichotomie ou bisection

Théorie

La dichotomie est une méthode de résolution numérique répandue et enseignée dès le lycée. Son principal avantage est qu'elle ne nécessite qu'une seule hypothèse : la continuité de f sur un intervalle $I \subset \mathbb{R}$.

La fonction $f : \mathbb{R} \rightarrow \mathbb{R}$ est donc continue sur un intervalle I . On va par ailleurs supposer que l'on travaille sur un intervalle $[a, b] \subset I$ tel que $f(a)f(b) < 0$. La continuité de f permet d'appliquer le théorème des valeurs intermédiaires : il existe $x^* \in [a, b]$ tel que $f(x^*) = 0$.

Le principe de l'algorithme est alors le suivant : on définit :

$(a_n), n \geq 0, (b_n), n \geq 0, a_0 = a, b_0 = 0, (x_n), n \geq 0, x_n = \frac{a_n + b_n}{2}, a_{n+1}$ et b_{n+1} par:

Tant que $|f(x_n)| > \varepsilon$ (ε désigne la précision recherchée) **faire**

Si $f(a_n)f(b_n) > 0$ **alors**

$a_{n+1} = x_n$

Sinon

$b_{n+1} = x_n$

fin si

fin tant que

Pour juger de l'efficacité de cette méthode, on s'intéresse à la convergence de la suite. Par construction, on a $(b_n - a_n) = (\frac{1}{2})n(b_0 - a_0)$.

Ainsi $|x_n - x^*| \leq \frac{1}{2}(b_n - a_n)$ donc $|x_n - x^*| \leq \frac{1}{2}^{n+1}(b_0 - a_0)$.

Application

Pour cette application, on étudie $f : \mathbb{R} \rightarrow \mathbb{R}$ définie telle que $f(x) = x^2 - 2$. On connaît évidemment la solution de cette équation, il s'agit de $\pm\sqrt{2}$. On choisit donc un intervalle $[a, b]$ qui encadre seulement une solution, $\sqrt{2}$, on prendra ici $[a, b] = [1, 2]$. On implémente donc dans *Scilab* le code 1.2.

Code 1.1: Dichotomie

```

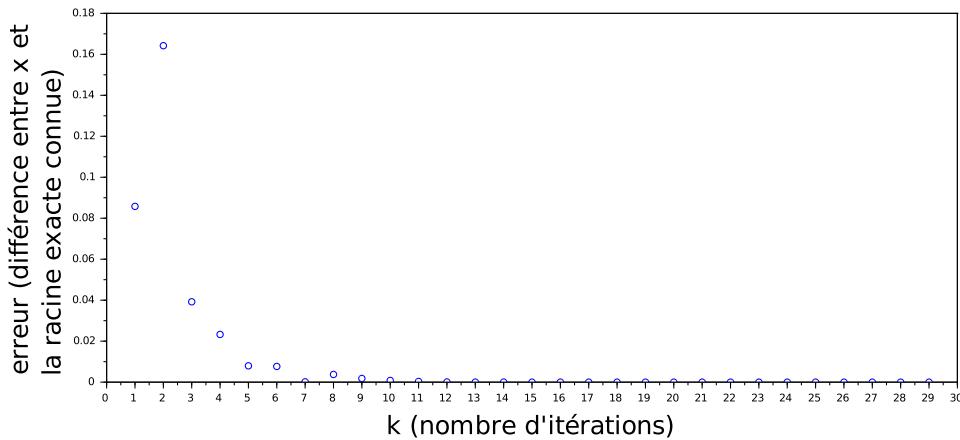
1  function y=f(x)
2      y=x^2-2;
3  endfunction
4
5 // initialisation
6 a=1; b=2;
7 ITMAX=1000;
8 precision=1e-10;
9 erreur=zeros(ITMAX,1);
10
11 for k=1:ITMAX
12     x=(a+b)/2;
13     erreur(k)=abs(x-sqrt(2));
14     if abs(f(x))<precision
15         break;
16     end
17     if f(a)*f(x)>0
18         a=x;
19     else b=x;
20     end
21 end
22
23 erreur=erreur(1:k);
24 // affichage
25 disp(x); disp(k); disp(erreur);

```

On trace l'évolution de l'erreur en fonction de l'itération (figure 1.1).

Figure 1.1: Erreur en fonction de l'itération

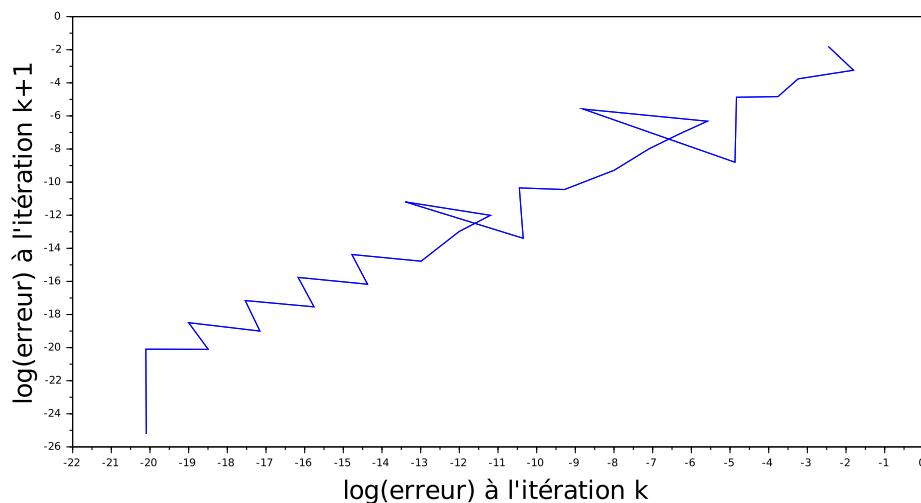
Variation de l'erreur en fonction du nombre d'itérations



Puis on s'intéresse à l'évolution de l'erreur relative (figure 1.2).

Figure 1.2: Diminution relative de l'erreur en fonction de l'itération

Diminution relative de l'erreur au fil des itérations



La solution est donc approchée en 29 itérations. Une remarque toute particulière pour la dichotomie : l'erreur n'est pas réduit de manière uniforme au fil des itérations (comme le montre la figure 1.2). Si on linéarise cette courbe (figure 1.2) grâce à *Scilab*, on obtient un coefficient proche de 1 (environ 1,0188), il s'agit effectivement d'une convergence linéaire, ou d'ordre 1.

1.3 La méthode de point fixe

Théorie

Cette méthode, également répandue, nécessite toutefois plus d'hypothèses que la dichotomie, en effet il est nécessaire que f soit une fonction dérivable.

L'idée de la méthode est de rechercher une fonction $g : \mathbb{R} \rightarrow \mathbb{R}$ telle que $f(x) = 0 \Leftrightarrow g(x) = x$. Cette fonction g (supposée dérivable) admet ainsi un point fixe, notons le x^* tel que $|g'(x^*)| < 1$. Alors, en appliquant le théorème des accroissements finis, il existe $[a, b]$ tel que $x^* \in [a, b]$ et la suite :

$$\begin{cases} x_0 \in [a, b] \\ x_{n+1} = g(x_n), n \geq 0 \end{cases} \text{ converge vers } x^* \text{ (la démonstration est admise ici, détaillée en MT90).}$$

Comme avec la dichotomie, s'intéresser à la convergence de la suite nous indique sur l'efficacité de la méthode. Si $g'(x^*) \neq 0$ alors $\frac{|x_{n+1} - x^*|}{|x_n - x^*|} < k$ avec $0 < k < 1$. Par récurrence, $|x_{n+1} - x^*| < k^n |x_0 - x^*|$.

Application

A nouveau, on étudie $f : \mathbb{R} \rightarrow \mathbb{R}$ définie telle que $f(x) = x^2 - 2$. Comme précédemment, on choisit l'intervalle $[a, b] = [1, 2]$, et on choisit ici $x_0 = \frac{3}{2}$ (on se place au milieu de l'intervalle d'étude). On implémente donc dans *Scilab* le code 1.2.

Code 1.2: Point fixe

```

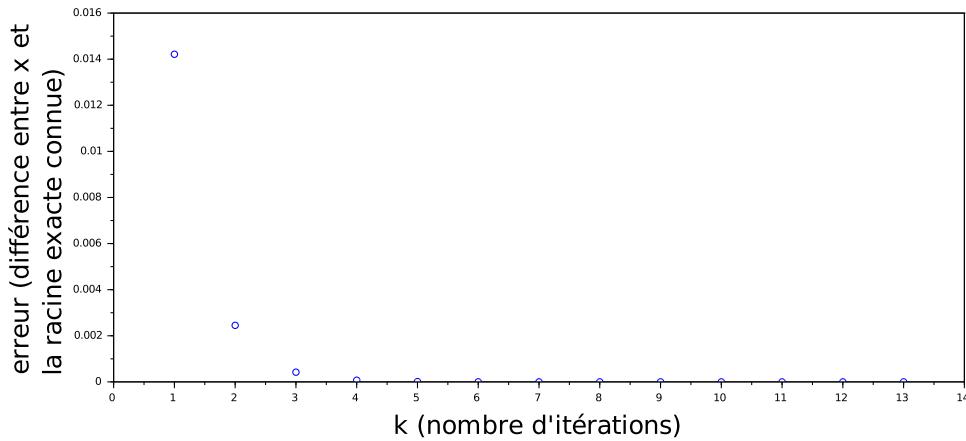
1 function y=f(x)
2     y=x^2-2;
3 endfunction
4
5 function y=g(x)
6     y=(x+2)/(x+1);
7 endfunction
8
9 a=1;
10 b=2;
11 x=3/2;
12 ITMAX=1000;
13 precision=1e-10;
14 erreur=zeros(ITMAX, 1);
15
16 for k=1:ITMAX
17     x=g(x);
18     erreur(k)=abs(x-sqrt(2));
19     if abs(f(x))<precision
20         break;
21     end
22 end
23
24 erreur=erreur(1:k);
25 disp(x); disp(k); disp(erreur);

```

On trace l'évolution de l'erreur en fonction de l'itération (figure 1.3).

Figure 1.3: Erreur en fonction de l'itération

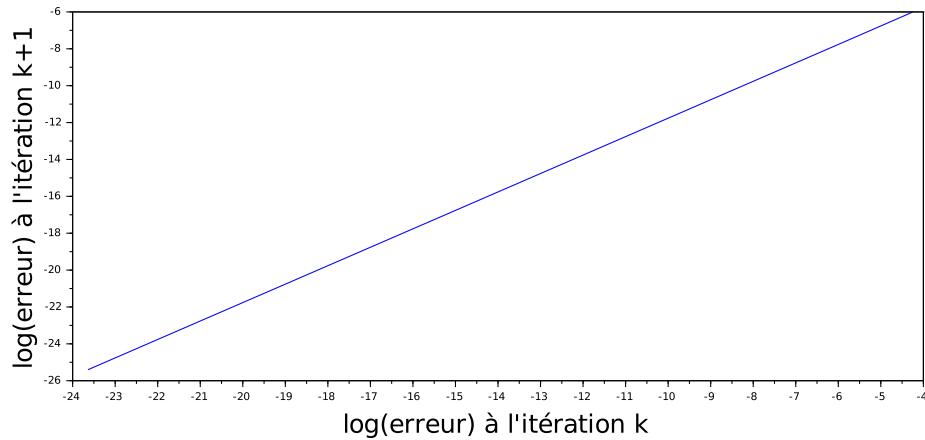
Variation de l'erreur en fonction du nombre d'itérations



Puis on s'intéresse à l'évolution de l'erreur relative (figure 1.4).

Figure 1.4: Diminution relative de l'erreur en fonction de l'itération

Diminution relative de l'erreur au fil des itérations



La solution est donc approchée en 13 itérations. Si on linéarise cette courbe (figure 1.4) grâce à *Scilab*, on obtient un coefficient très proche de 1 (environ 1,0001), effectivement, il s'agit à nouveau d'une convergence d'ordre 1.

1.4 La méthode de Newton appliquée à $n = 1$

Théorie

La méthode de Newton nécessite encore plus d'hypothèses que les deux méthodes que nous venons d'étudier. En effet, on suppose ici que la fonction f est deux fois continûment dérivable. On écrit ensuite le développement de Taylor Lagrange sur f : soit $x_0 \in \mathbb{R}$, il existe $\theta \in]0, 1[$,

$$f(x_0 + h) = f(x_0) + f'(x_0)h + \frac{h^2}{2}f''(x_0 + \theta h)$$

si on pose $x = x_0 + h$, le développement devient :

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

On effectue l'approximation affine de $f(x)$: $T_{x_0}(x) = f(x_0) + f'(x_0)(x - x_0)$

et on définit x_1 par $T_{x_0}(x_1) = 0 \Leftrightarrow f(x_0) + f'(x_0)(x_1 - x_0) = 0$

$$\text{si } f'(x_0) \neq 0 \text{ alors } x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Ainsi, graphiquement, la méthode de Newton consiste à tracer une droite tangente à $y = f(x)$ en x_n , x_{n+1} est alors la racine de cette tangente.

L'algorithme est donc le suivant : pour x_0 donné, et ε la précision à atteindre,

```

Tant que | $f(x_n)$ | >  $\varepsilon$  faire
     $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ 
fin tant que

```

On peut remarquer que la méthode de Newton est une méthode de point fixe particulière. En effet $x_{n+1} = x_n - \frac{f'(x_0)}{f'(x_0)} \Leftrightarrow x_{n+1} = g(x_n)$ avec $g(x) = x - \frac{f(x)}{f'(x)}$. Ainsi $g'(x) = 1 - \frac{f'(x)^2 - f(x)f''(x)}{f'(x)^2}$, on émet l'hypothèse que $f'(x^*) \neq 0$, donc $g'(x) = 1 - \frac{f'(x^*)^2}{f'(x^*)^2} = 0$, d'où $|g'(x^*)| < 1$.

On s'intéresse maintenant à la convergence de cette suite, toujours pour juger de l'efficacité de la méthode. Si on suppose que f est trois fois continûment dérivable, on a $x_{n+1} - x^* = g(x_n) - g(x^*)$, $g(x_n) = g(x^*) + (x_n - x^*)g'(x^*) + \frac{(x_n - x^*)^2}{2}g''(\xi) \Rightarrow |x_{n+1} - x^*| \leq C|x_n - x^*|^2$ avec $C = \frac{1}{2}\max|g''(\xi)|$.

Application

Encore une fois, on étudie $f : \mathbb{R} \rightarrow \mathbb{R}$ définie telle que $f(x) = x^2 - 2$. Comme précédemment, on choisit l'intervalle $[a, b] = [1, 2]$ et $x_0 = \frac{3}{2}$. On implémente donc dans *Scilab* le code 1.3.

Code 1.3: Newton (appliquée à $n = 1$)

```

1 function y=f(x)
2     y=x^2-2;
3 endfunction

4
5 function y=g(x)
6     y=x/2+1/x;
7 endfunction

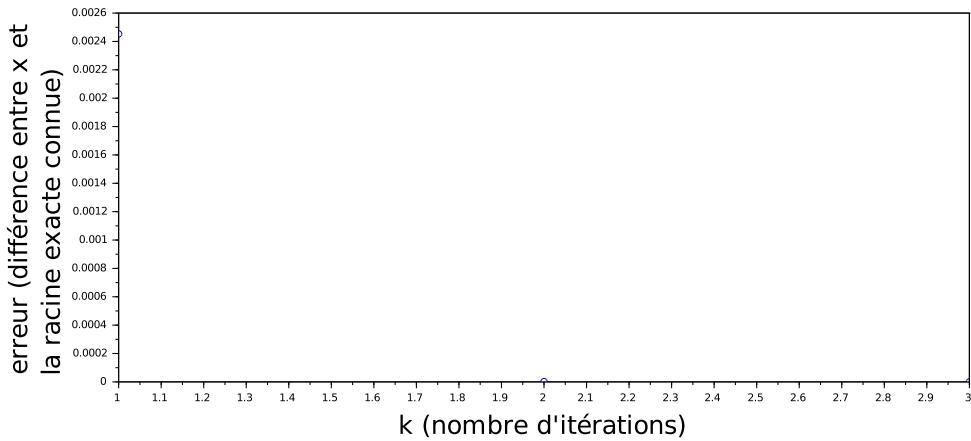
8
9 a=1; b=2; x=3/2;
10 ITMAX=1000; precision=1e-10; erreur=zeros(ITMAX,1); // initialisation
11
12 for k=1:ITMAX
13     x=g(x);
14     erreur(k)=abs(x-sqrt(2));
15     if abs(f(x))<precision
16         break;
17     end
18 end
19
20 erreur=erreur(1:k);
21 disp(x); disp(k); disp(erreur); // affichage

```

On trace l'évolution de l'erreur en fonction de l'itération (figure 1.5).

Figure 1.5: Erreur en fonction de l'itération

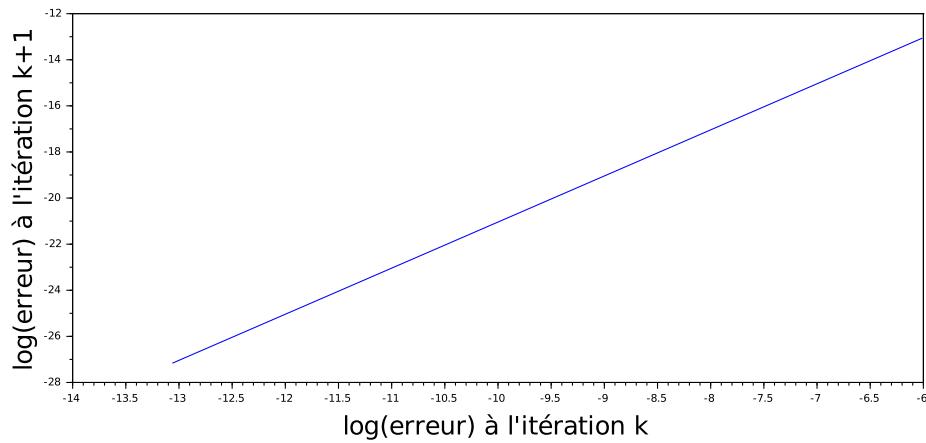
Variation de l'erreur en fonction du nombre d'itérations



Puis on s'intéresse à l'évolution de l'erreur relative (figure 1.6).

Figure 1.6: Diminution relative de l'erreur en fonction de l'itération

Diminution relative de l'erreur au fil des itérations



La solution est donc approchée en 3 itérations. Si on linéarise cette courbe (figure 1.6) grâce à *Scilab*, on obtient un coefficient très proche de 2 (environ 1,9998), il s'agit effectivement d'une convergence quadratique ou d'ordre 2.

1.5 La méthode de la sécante

Théorie

Comme nous l'avons évoqué précédemment, la méthode de la sécante est directement dérivée de la méthode de Newton. En effet, dans l'algorithme, on prenait $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$. Pour la méthode de la sécante, il suffit d'approcher $f'(x_n)$ par le taux d'accroissement, à savoir $\frac{f(x_n) - f(x_{n+1})}{x_n - x_{n+1}}$.

L'algorithme devient donc, pour x_0 et x_1 donnés :

```
Tant que |f(xn)| > ε faire
    xn+1 = xn -  $\frac{f(x_n)}{f(x_n) - f(x_{n+1})}(x_n - x_{n+1})$ 
fin tant que
```

Application

On étudie une dernière fois $f : \mathbb{R} \rightarrow \mathbb{R}$ telle que $f(x) = x^2 - 2$. Comme précédemment, on choisit l'intervalle $[a, b] = [x_0, x_1] = [1, 2]$. On implémente donc dans *Scilab* le code 1.4.

Code 1.4: Sécante

```

1 function y=f(x)
2     y=x^2-2;
3 endfunction
4
5 x=2; y=1;
6 ITMAX=1000; precision=1e-10; erreur=zeros(ITMAX,1); // initialisation
7
8 for k=1:ITMAX
9     y=x-f(x)*(x-y)/(f(x)-f(y));
10    // passage par une variable temporaire pour inverser les valeurs de x et y
11    temp=x; x=y; y=temp;
12    erreur(k)=abs(y-sqrt(2));
13    if abs(f(y))<precision
14        break;
15    end
16 end
17
18 erreur=erreur(1:k); disp(x); disp(k); disp(erreur); // affichage

```

Figure 1.7: Erreur en fonction de l'itération

Variation de l'erreur en fonction du nombre d'itérations

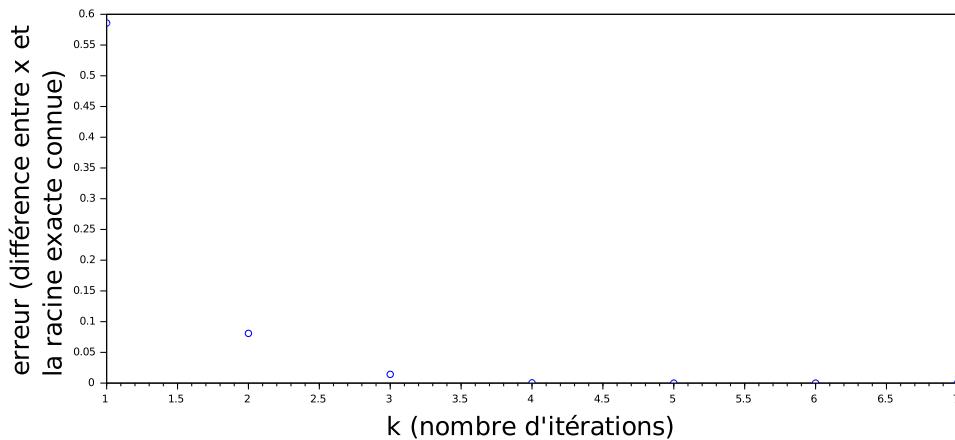
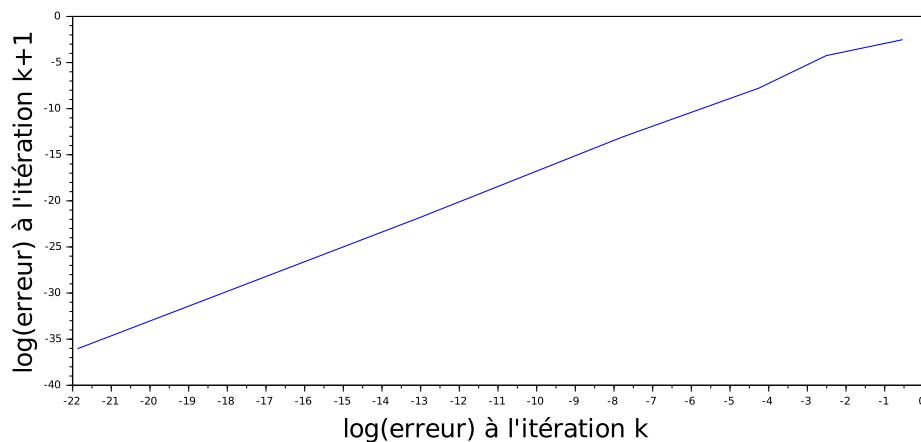


Figure 1.8: Diminution relative de l'erreur en fonction de l'itération

Diminution relative de l'erreur au fil des itérations



La solution est donc approchée en 7 itérations. Si on linéarise cette courbe (figure 1.8) grâce à *Scilab*, on obtient un coefficient d'environ 1,6004, il s'agit effectivement d'une convergence d'ordre $\frac{1+\sqrt{5}}{2}$ (le nombre d'or).

1.6 Pour $n > 1$: La méthode de Newton

Il existe bien d'autres méthodes afin d'approcher numériquement la solution d'une équations à plusieurs inconnues sous la forme de $f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_n(x) \end{pmatrix} = \vec{0}$ avec $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$. La seule méthode dont nous étudierons la partie théorique est la méthode de Newton, appliquée donc dans un cas multidimensionnel.

1.6.1 Théorie

Pour comprendre cette méthode, il apparaît judicieux de revenir sur la notion de différentiabilité. Soit $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ avec $f(x) = \begin{pmatrix} f_1(x) \\ \vdots \\ f_n(x) \end{pmatrix}$ et $x_0 \in \mathbb{R}^n$.

On dit que f est différentiable en x_0 s'il existe une matrice $A \in \mathcal{M}_{n,n}$ telle que

$\forall h \in \mathbb{R}^n, f(x_0 + h) = f(x_0) + Ah + ||h||\varepsilon(h)$

où $\lim_{h \rightarrow \vec{0}} \varepsilon(h) = \vec{0}$ et $||h|| \stackrel{\text{def}}{=} \left(\sum_{i=1}^n h_i^2 \right)^{\frac{1}{2}}$.

Les coefficients de A seront alors $a_{ij} = \frac{\partial f_i}{\partial x_j}(x_0)$.

On note $A \stackrel{\text{def}}{=} J_f(x_0)$, A est la matrice Jacobienne de f en x_0 .

De manière analogue au développement de la partie unidimensionnelle, pour x_0 donné et $x \in \mathbb{R}$, on a :

$$f(x) = f(x_0) + J_f(x_0)(x - x_0) + ||x - x_0||\varepsilon(x - x_0)$$

On effectue l'approximation affine de $f(x)$: $T_{x_0}(x) = f(x_0) + J_f(x_0)(x - x_0)$

On définit x_1 par $T_{x_0}(x_1) = \vec{0}$ (il s'agit d'un système linéaire de n équations à n inconnues)

$$\Leftrightarrow f(x_0) + J_f(x_0)(x_1 - x_0) = \vec{0} \Leftrightarrow J_f(x_0)(x_1 - x_0) = -f(x_0)$$

d'où $x_1 = x_0 - (J_f(x_0))^{-1}f(x_0)$

L'idée de la méthode de Newton est alors la suivante (notons que dans la pratique, on n'inverse pas la matrice, on résout le système d'équation linéaire), pour x_0 donné,

```

Tant que ||f(x_n)|| > ε et J_f(x_n) est inversible faire
    résoudre J_f(x_n)h_n = -f(x_n)
    x_{n+1} = x_n + h_n
fin tant que

```

1.6.2 Fonction *fsolve*

Avant de commencer l'application de la méthode de Newton, il apparaît judicieux de présenter la macro *fsolve* de *Scilab*. En effet, *Scilab* possède déjà une méthode de résolution des problèmes non linéaires inspirée de la méthode de Newton. *fsolve* a différents prototypes, les arguments qui nous intéressent ici sont :

- le x_0 donné
- la fonction f
- (facultativement) sa dérivée (*resp.* sa Jacobienne) df

Si le dernier paramètre, nécessaire à l'application de la méthode de Newton, n'est pas renseigné, *Scilab* ne pouvant la calculer directement, il va l'approcher grâce au développement de Taylor Lagrange.

Ainsi, la résolution d'un problème non linéaires avec *Scilab* peut se ramener à la mise en forme du problème sous sa forme standard, $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$, $f(x) = \vec{0}$ puis à l'utilisation de la macro *fsolve*.

1.6.3 Application 1 : le GPS

Énoncé et passage en forme standard

Le sujet de ce problème est le suivant :

”Le GPS est un système de positionnement basé sur la connaissance de la distance du récepteur R à trois satellites (situés à des orbites de l'ordre de 28000km). On suppose que les trois satellites au moment du calcul de distance ont les positions suivantes dans un repère cartésien d'origine le centre de la terre:

$$S_1 = (-11716.227778, -10118.754628, 21741.083973) \text{ (unité=km)}$$

$$S_2 = (-12082.643974, -20428.242179, 11741.374154)$$

$$S_3 = (14373.286650, -10448.439349, 19596.404858)$$

Sachant que les trois distances respectives au récepteur ont été calculées et valent : $(d_1, d_2, d_3) = (22163.847742, 21492.777482, 21492.469326)$,

Déterminer avec *fsolve* la position du récepteur (et vérifier que celui-ci se trouve bien à la surface de la terre...)."

Notons X la position du récepteur avec $X = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$.
Le problème nous donne donc le système suivant : $\begin{cases} \|S_1 - X\| = d_1 \\ \|S_2 - X\| = d_2 \\ \|S_3 - X\| = d_3 \end{cases}$

Avec $\|X\| = \sqrt{x^2 + y^2 + z^2}$ (il s'agit de la norme euclidienne). La racine carrée pose problème, car la fonction $f : \mathbb{R} \rightarrow \mathbb{R}$, $f(x) = \sqrt{x}$ n'est pas dérivable sur \mathbb{R} . On élève donc chacune des équations du système au carré. La fonction que l'on cherche donc à annuler dans notre étude est :

$$f : \mathbb{R}^3 \rightarrow \mathbb{R}^3, f(X) = \begin{pmatrix} \|S_1 - X\|^2 - d_1^2 \\ \|S_2 - X\|^2 - d_2^2 \\ \|S_3 - X\|^2 - d_3^2 \end{pmatrix}.$$

Avec *Scilab* comme outils, plusieurs solutions s'offrent donc à nous pour résoudre le problème :

- utiliser la fonction *fsolve* sans renseigner la Jacobienne
- calculer la Jacobienne et utiliser la fonction *fsolve* en la renseignant
- calculer la Jacobienne et appliquer la méthode de Newton (sans utiliser *fsolve*)

Il peut être intéressant de comparer ces différentes méthodes.

Méthode 1

Pour la première méthode, il suffit de renseigner la fonction et de faire appel à *fsolve*. On implémente donc dans *Scilab* le code 1.5.

Code 1.5: GPS (Méthode 1)

```

1 function out=f(X)
2     out=[norm(X-S1)^2-d(1)^2
3         norm(X-S2)^2-d(2)^2
4         norm(X-S3)^2-d(3)^2];
5 endfunction
6
7 S1 = [-11716.227778, -10118.754628, 21741.083973]';
8 S2 = [-12082.643974, -20428.242179, 11741.374154]';
9 S3 = [14373.286650, -10448.439349, 19596.404858]';
10 d=[22163.847742, 21492.777482, 21492.469326];
11 X=zeros(3,1);
12
13 X=fsolve(X,f);
14 disp(X);

```

On obtient l'affichage suivant :

```
-->exec('/home/marlow/latex/GPS_1.sce', -1)
595.02505
-
- 4856.0251
4078.33
```

Méthode 2

Cette méthode ne se différencie pas beaucoup dans la première, la seule différence est que l'on renseigne la Jacobienne en argument de *fsolve*.

Calcul de la Jacobienne :

Pour simplifier le calcul, on s'intéresse à la première composante de $f(X)$ (à la constante près):
 $f_1(X) : \mathbb{R}^3 \rightarrow \mathbb{R}$ avec $f_1(X) = \|X - S_1\|^2$

$$\begin{aligned} f_1(X + h) &= \|(X + h) - S_1\|^2 \\ &= ((X - S_1) + h)^T((X - S_1) + h) \\ &= (X - S_1)^T(X - S_1) + h^T(X - S_1) + (X - S_1)^Th + h^Th \\ &= \|X - S_1\|^2 + 2(X - S_1)^Th + \|h\|^2 \\ &= f_1(X) + J_{f_1}(X) + \|h\|\varepsilon(h) \end{aligned}$$

On peut raisonner de la même façon pour les autres composantes de $f(X)$, on obtient donc :

$$J_f(X) = \begin{bmatrix} (X - S_1)^T \\ (X - S_2)^T \\ (X - S_3)^T \end{bmatrix}$$

Ainsi, on peut implémenter la Jacobienne dans notre précédent programme *Scilab* et la renseigner dans les arguments de *fsolve*, on obtient le code 1.6.

Code 1.6: GPS (Méthode 2)

```

1 function out=f(X)
2     out=[norm(X-S1)^2-d(1)^2
3           norm(X-S2)^2-d(2)^2
4           norm(X-S3)^2-d(3)^2];
5 endfunction
6
7 function out=Jf(X)
8     out=2*[(X-S1)'
9                 (X-S2)'
10                (X-S3)' ];
11 endfunction
12
13
14 S1 = [-11716.227778, -10118.754628, 21741.083973]';
15 S2 = [-12082.643974, -20428.242179, 11741.374154]';
16 S3 = [14373.286650, -10448.439349, 19596.404858]';
17 d=[22163.847742, 21492.777482, 21492.469326];
18 X=zeros(3,1);
19
20 X=fsolve(X,f,Jf);
21 disp(X);
```

On obtient l'affichage suivant :

```
-->exec('/home/marlow/latex/GPS_2.sce', -1)
595.02505
-
- 4856.0251
4078.33
```

Méthode 3

Enfin, pour tirer parti au maximum de cet exemple, nous allons résoudre le problème sans faire appel à la macro *fsolve* (car *A vaincre sans péril, on triomphe sans gloire*).

La Jacobienne a été calculée pour appliquer la deuxième méthode, on a donc : $f(X) = \begin{pmatrix} \|S_1 - X\|^2 - d_1^2 \\ \|S_2 - X\|^2 - d_2^2 \\ \|S_3 - X\|^2 - d_3^2 \end{pmatrix}$

et $J_f(X) = \begin{bmatrix} (X - S_1)^T \\ (X - S_2)^T \\ (X - S_3)^T \end{bmatrix}$

On implémente donc la méthode de Newton dans *Scilab* le code 1.7, suivant l'algorithme que nous en avions donné dans la partie théorique.

Code 1.7: GPS (Méthode 3)

```

1  function out=f(X)
2      out=[norm(X-S1)^2-d(1)^2
3          norm(X-S2)^2-d(2)^2
4          norm(X-S3)^2-d(3)^2];
5  endfunction
6
7  function out=Jf(X)
8      out=2*[(X-S1),
9              (X-S2),
10             (X-S3)]';
11 endfunction
12
13
14 S1 = [-11716.227778, -10118.754628, 21741.083973]';
15 S2 = [-12082.643974, -20428.242179, 11741.374154]';
16 S3 = [14373.286650, -10448.439349, 19596.404858]';
17 d=[22163.847742, 21492.777482, 21492.469326];
18 ITMAX=1000;
19 precision=10e-10;
20
21 X=[1000 1000 1000]';
22 //choix arbitraire d'une position initiale proche de la solution
23
24 for k=1:ITMAX
25     if abs(norm(f(X)))<precision
26         break;
27     end
28     X = X-Jf(X)\f(X);
29 end
30 disp(X);

```

On obtient l'affichage suivant :

```
-->exec('/home/marlow/latex/GPS_3.sce', -1)
      595.02505
      - 4856.0251
      4078.33
```

Cette méthode nous permet de connaître précisément le nombre d'itérations nécessaire pour approcher la solution : $k = 10$ après exécution du programme.

Bilan

Les trois méthodes nous offrent exactement la même solution suivant l'affichage standard de Scilab. On pourrait chercher des différences à quelques décimales près entre les différentes méthodes, comme on pourrait chercher à connaître le nombre d'itérations (ou d'appels aux fonctions) nécessaires à chaque algorithme avant d'approcher la solution. Toutefois, ce n'est pas vraiment l'objectif qui était fixé, nous nous en tiendrons donc là pour le GPS.

Petite remarque : vérifions tout de même que la position que nous avons correspond à une position qui a du sens. En effet, le sujet nous invitait à vérifier que le récepteur se trouve bien à la surface de la Terre : on calcule donc la norme euclidienne de X (notre solution) :

```
-->norm(X)
ans =
6369.2864
```

Si on suppose un rayon de la Terre égal à 6 371 km (réponse de *Google*), on obtient bien un récepteur GPS à la surface de la Terre.

1.6.4 Application 2 : la cinématique inversée

Énoncé et passage en forme standard

Le sujet de ce problème est le suivant :

"On considère un bras robot articulé dans le plan (x_1, x_2) , d'origine O , avec un premier segment de longueur l_1 et faisant un angle θ_1 avec $(0, x_1)$ et un deuxième segment de longueur l_2 faisant un angle θ_2 avec le premier segment. L'extrémité du bras a pour coordonnées :

$$M(\theta) = (l_1 \cos \theta_1 + l_2 \cos(\theta_1 + \theta_2), l_1 \sin \theta_1 + l_2 \sin(\theta_1 + \theta_2)).$$

1. Écrire une macro *Scilab* déterminant θ tel que $M(\theta) = A$ où $A = (x_A, y_A)$ est un point du plan.
2. On prend $l_1 = l_2 = 1$. Écrire un programme *Scilab* utilisant *fsolve* et représentant les positions successives du bras lorsque le point $A(t)$ est défini par une courbe paramétrique, par exemple :

$$A(t) = \begin{cases} x_1(t) = 1 + \frac{1}{2} \cos(t) \\ x_2(t) = 1 + \frac{1}{2} \sin(t) \end{cases}$$

Le problème nous donne donc le système suivant :

$$M(\theta) = A \Leftrightarrow \begin{cases} l_1 \cos \theta_1 + l_2 \cos(\theta_1 + \theta_2) = x_A \\ l_1 \sin \theta_1 + l_2 \sin(\theta_1 + \theta_2) = y_A \end{cases}$$

La fonction que l'on cherche donc à annuler dans notre étude est :

$$f : \mathbb{R}^2 \rightarrow \mathbb{R}^2, f(\theta) = \begin{pmatrix} l_1 \cos \theta_1 + l_2 \cos(\theta_1 + \theta_2) - x_A \\ l_1 \sin \theta_1 + l_2 \sin(\theta_1 + \theta_2) - y_A \end{pmatrix}.$$

Tout comme avec l'application du GPS, plusieurs solutions s'offrent à nous pour résoudre le problème. Toutefois, se contenter d'utiliser la macro *fsolve* ne permet pas d'apprécier l'exécution de la méthode de Newton, itération après itération.

Calcul de la Jacobienne

Pour calculer la Jacobienne de la fonction, nous allons utiliser sa définition précédemment énoncée, c'est à dire :

$$Jf(\theta) = A \text{ avec } A \text{ une matrice dont les coefficients sont } a_{ij} = \frac{\partial f_i}{\partial \theta_j}(\theta)$$

Par simple calcul de dérivées partielles, on obtient donc :

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

Résolution

On retranscrit donc à nouveau l'algorithme de la méthode de Newton appliquée à notre exemple dans *Scilab*. On obtient donc le code 1.8.

Code 1.8: Cinématique inversée (code principal)

```

1 function out=f(theta)
2     M=[11*cos(theta(1))+12*cos(sum(theta))
3         11*sin(theta(1))+12*sin(sum(theta))];
4     out=M-[xA;yA];
5 endfunction
6
7 function out=Jf(theta)
8     out=[-11*sin(theta(1))-12*sin(sum(theta)) -12*sin(sum(theta))
9         11*cos(theta(1))+12*cos(sum(theta)) 12*cos(sum(theta))];
10 endfunction
11
12 l1=1; l2=1;
13 ITMAX=1000; precision=1e-10;
14 theta=[0,%pi/2]'; //choix arbitraire d'une position initiale
15
16 for t=linspace(0,2*pi,100) //on choisit de tracer 100 bras
17     xA = 1 + (1/2)*cos(t); yA = 1 + (1/2)*sin(t);
18     for k=1:ITMAX
19         if abs(norm(f(theta)))<precision
20             break;
21         end
22         theta = theta-Jf(theta)\f(theta);
23     end
24     dessine_bras(theta)
25 end

```

Code fourni pour un rendu graphique

Le code *dessine_bras* appelé pour le rendu graphique est le code 1.9.

Code 1.9: Cinématique inversée (code graphique)

```

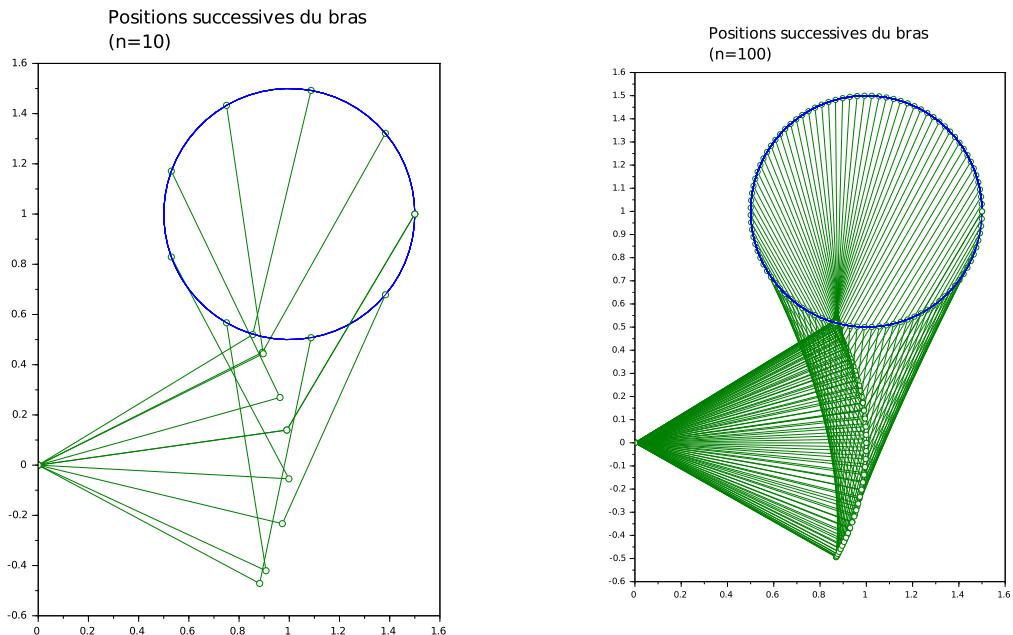
1 function dessine_bras(theta)
2     l1=1;l2=1;
3     t=linspace(0,2*pi,64);
4     x=[0
5         11*cos(theta(1))
6         11*cos(theta(1))+12*cos(sum(theta))];
7     y=[0
8         11*sin(theta(1))
9         11*sin(theta(1))+12*sin(sum(theta))];
10    drawlater
11    plot(1+.5*cos(t),1+.5*sin(t),x,y,"-o");
12    set(gca(),"isoview","on")
13    drawnow
14 endfunction

```

Affichage

On peut ainsi avoir accès après exécution du programme principal à un affichage dynamique de la position du bras à chaque itération. On peut par ailleurs faire varier le nombre d'itérations pour construire plus de bras (figure 1.9 : affichages pour différentes valeurs de n).

Figure 1.9: Différents affichages de la cinématique inversée en fonction de n



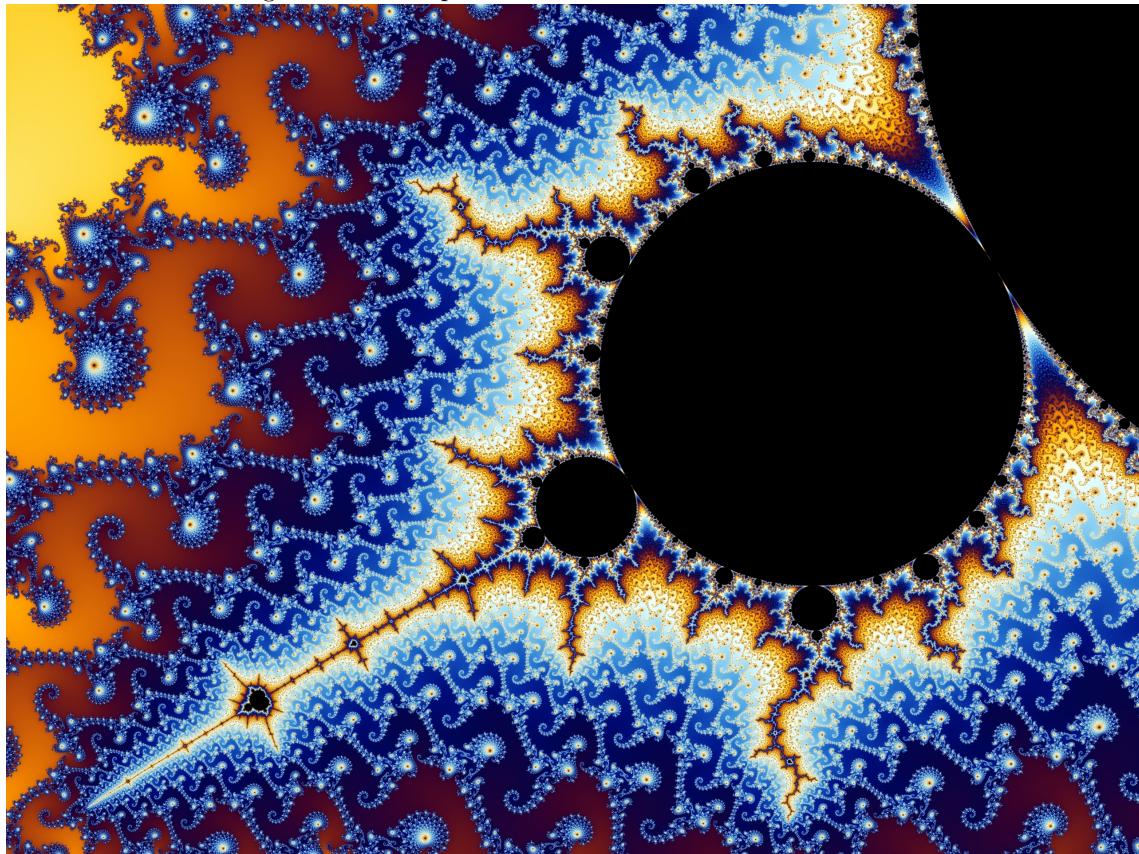
CHAPITRE 2

FRACTALES

2.1 Introduction

L'existence des fractales part d'une interrogation assez surprenante : comment construire des figures parfaitement irrégulières, peu importe l'échelle choisie pour les observer (c'est à dire des courbes de fonctions continues mais non dérivables) ? Comment simuler grâce à des outils mathématiques des motifs présent dans la nature comme l'écume des vagues, les nuages,... ? Comment représenter *l'infini* sur une surface finie ? La solution à ces interrogations est d'aller au delà de la géométrie classique, c'est à dire euclidienne.

Figure 2.1: Exemple de fractales : fractale de Mandelbrot



2.2 Dimension de Hausdorff

Avant d'étudier des fractales particuliers, il apparaît pertinent de commencer par définir *la dimension de Hausdorff*, c'est à dire la notion de dimension que l'on peut appliquer à un fractal. La dimension de Van Hausdorff d'une fractale K est définie ainsi :

- on note $N(\varepsilon)$ le nombre de carrés de longueur (ou de disques de rayon) ε recouvrant K
- on a alors $d = \lim_{\varepsilon \rightarrow 0} \frac{\ln(N(\varepsilon))}{\ln(\frac{1}{\varepsilon})}$, $d \notin \mathbb{N}$

Avec cette définition, nous calculerons donc la *dimension de Hausdorff* de chacune des fractales que nous étudierons.

2.3 Ensemble de Cantor

2.3.1 Théorie

Présentation

L'étude de tels objets n'a pas attendu l'apparition du terme "fractale" par Benoît Mandelbrot en 1974. Certains mathématiciens se sont intéressés à ces objets avant même de pouvoir les nommer. On considère souvent comme la première figure fractale l'objet construit par le mathématicien allemand Benoît Cantor : *l'ensemble de Cantor*.

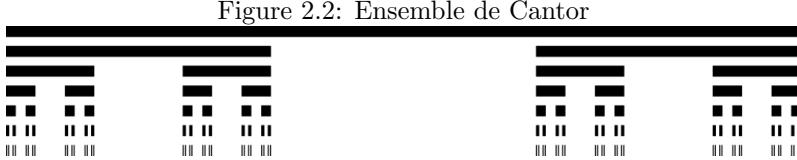


Figure 2.2: Ensemble de Cantor

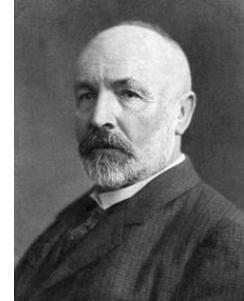


Figure 2.3: Cantor
1845-1918

Définition algorithmique

Initialisation Un segment $[0,1]$

Opération de base : Partager le segment en trois parties égales et ne pas retenir le segment central

Itération : Itérer l'opération sur chacun des segments retenus

Remarques

On appelle $C^{(k)}$ l'ensemble obtenu à l'étape k , on note : $C^{(\infty)} = \bigcap_{k=0}^{\infty} C^{(k)}$
On a par ailleurs les propriétés suivantes :

- i) $|C^{(\infty)}| = 0$
- ii) $C^{(\infty)}$ est dénombrable

On peut démontrer ces propriétés :

démo i)

$$|C^{(0)}| = 1, |C^{(1)}| = \frac{2}{3}, |C^{(2)}| = \frac{4}{9} = \left(\frac{2}{3}\right)^2$$

par récurrence, on obtient $|C^{(k)}| = \left(\frac{2}{3}\right)^k \Rightarrow \lim_{k \rightarrow \infty} |C^{(k)}| = 0 \Rightarrow |C^{(\infty)}| = 0$

démo ii)

$C^{(\infty)}$ est en bijection avec $[0, 1] \Rightarrow C^{(\infty)}$ est dénombrable

Or on a le **théorème** :

$\forall x \in C^{(\infty)}, x = (0, x_1 x_2 \cdots x_k \cdots)_3$ avec $(x)_3$ désigne la base 3 de x et $x_i = 0$ ou 2

exemple :

$$\begin{aligned}\frac{1}{3} &= (0, 1)_3 = (0, 0222 \cdots 2 \cdots)_3 = \frac{0}{3} + \frac{2}{3^2} + \frac{2}{3^3} + \cdots + \frac{2}{3^k} + \cdots \\ &= \frac{2}{3^2} \left(1 + \frac{1}{3} + \cdots + \frac{1}{3^k} + \cdots \right) = \frac{2}{9} \times \frac{1}{1 - \frac{1}{3}} = \frac{2}{9} \times \frac{3}{2} = \frac{1}{3}\end{aligned}$$

Soit donc la **bijection** $x \rightarrow \frac{x}{2} = 0, \frac{x_1}{2} \frac{x_2}{2} \cdots \frac{x_k}{2} \cdots$

donc $\forall x \in C^{(\infty)}$ on associe en base 2 le nombre $(0, x_1 x_2 \cdots x_k \cdots)_2$

Dimension de l'ensemble de Cantor

Pour calculer la dimension de l'ensemble de Cantor au sens de la *dimension de Hausdorff*, on compte le nombre de carrés nécessaires à chaque étape pour recouvrir la figure :

- à l'initialisation, 1 seul carré de côté 1
- à l'itération 1, 2 carrés de côté $\frac{1}{3}$ sont nécessaires pour recouvrir les segments
- à l'itération 2, $4 = 2^2$ carrés de côté $\frac{1}{9} = \frac{1}{3^2}$ sont nécessaires pour recouvrir les segments
- à l'itération k, 2^k carrés de côté $\frac{1}{3^k}$ sont nécessaires pour recouvrir les segments

On a donc $d = \lim_{k \rightarrow \infty} \frac{\ln(2^k)}{\ln(3^k)} = \frac{\ln(2)}{\ln(3)}$ la dimension de l'ensemble de Cantor.

2.3.2 Application dans Scilab (méthode récursive)

La méthode récursive s'inspire directement de l'algorithme de définition de l'ensemble de Cantor. On implémente donc dans *Scilab* le code 2.1.

Code 2.1: Ensemble de Cantor (méthode récursive)

```

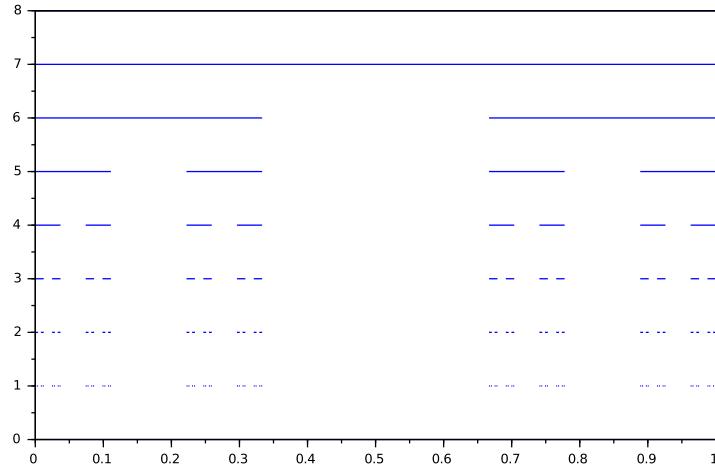
1 function cantor(n,a,b)
2   if n<1
3     break;
4   else
5     I = (b-a)/3;
6     plot([a,a+I],[n,n]);
7     plot([b-I,b],[n,n]);
8     cantor(n-1,a,a+I);
9     cantor(n-1,b-I,b);
10    end
11 endfunction
12
13 clf;
14 N=input("Entrez n, le nombre de niveaux a dessiner : ");
15 plot([0,1],[N+1,N+1]); //pour permettre une visualisation du premier niveau
16 plot([0,1],[N,N]);
17 N=N-1;
18 cantor(N,0,1)
19 plot([0,1],[0,0]); //pour permettre une visualisation du dernier niveau

```

L'exécution du code nous donne ainsi la figure 2.4.

Figure 2.4: Figure pour $n=7$

Ensemble de Cantor à 7 niveaux (méthode récursive)



2.3.3 Application dans *Scilab* (méthode itérative)

Même si la méthode récursive appliquée précédemment pour dessiner l'ensemble de Cantor s'est révélée efficace pour $n = 7$ niveaux, elle l'est beaucoup moins pour des valeurs de n supérieures. En effet, la complexité de l'algorithme est de l'ordre de $O(2^n)$ (on réalise 2 appels récursifs pour chaque niveau), on dit qu'elle est exponentielle, un tel algorithme est donc de plus en plus *gourmand* en temps et en mémoire à mesure que n augmente. En outre, la complexité des algorithmes récursifs devient rapidement délirante pour tracer des fractales en 2D, encore d'avantage en 3D. C'est pourquoi il est pertinent de s'intéresser dès à présent à d'autres méthodes de dessin des fractales : les méthodes itératives. Nous allons donc étudier l'une d'entre elles : celles des IFS (*Iterated Functions Systems*), en français *les systèmes de fonctions itérées*.

La théorie des IFS, basée sur l'invariance par changement d'échelle, permet la représentation fonctionnelle d'un fractale. Cette théorie fait appel à la notion de *fonctions contractantes* sur un espace métrique M , (c'est à dire muni d'une notion de distance d entre deux éléments). Graphiquement, une fonction contractante "rapproche les images". Une fonction f est dite *contractante* ou *k-contractante* sur $M(E, d)$ ssi :

$$\exists k, 0 \leq k \leq 1, \forall (x, y) \in E^2, d(f(x), f(y)) \leq kd(x, y) \quad (2.1)$$

Un IFS est un ensemble de fonctions f_1, f_2, \dots, f_n toutes contractantes sur un ensemble de compacts de M munis de la *distance de Hausdorff*. On définit alors la fonction f , avec A un compact, par :

$$f(A) = f_1(A) \cup f_2(A) \cup \dots \cup f_n(A) \quad (2.2)$$

On peut montrer que f est aussi contractante de rapport de contraction $k = \max(k_i)$.

Le *théorème du collage*, démontré par Barnsley en 1985, énonce que tout IFS converge vers un *attracteur* A , c'est à dire un ensemble vers lequel le système évolue irrémédiablement, tel que $f(A) = A$ pour tout donnée initiale choisie, A est donc un point fixe de f . Autrement dit, un IFS converge vers un compact attracteur A qui se révèle être une fractale.

Pour une fractale particulière, on déduit les fonctions f_1, f_2, \dots, f_n à partir de la définition algorithmique de la fractale. On travaille donc sur un ensemble fini de points, et pour chaque point, on choisit (aléatoirement) d'y appliquer une des fonctions f_1, f_2, \dots, f_n .

Pour l'ensemble de Cantor, on a :

$$f(C) = f_0(C) \cup f_1(C) \text{ avec } \begin{cases} f_0, f_1 : [0, 1] \longrightarrow [0, 1] \\ f_0 : x \longrightarrow \frac{x}{3} \\ f_1 : x \longrightarrow \frac{x}{3} + \frac{2}{3} \end{cases} \text{ d'où } \begin{cases} C_0 : [0, 1] \\ C_{n+1} : f(C_n) \end{cases} \quad (2.3)$$

Finalement C_n converge au sens de la distance de Hausdorff vers C^∞ , l'ensemble de Cantor. Pour la sélection de la fonction à appliquer à une point particulier, on applique une méthode équiprobable afin d'obtenir un ensemble uniforme.

On implémente donc dans *Scilab* le code suivant :

Code 2.2: Ensemble de Cantor (méthode itérative)

```

1 function Y=f0 (X)
2     Y=[1/3  0;0  1/3]*X;
3 endfunction

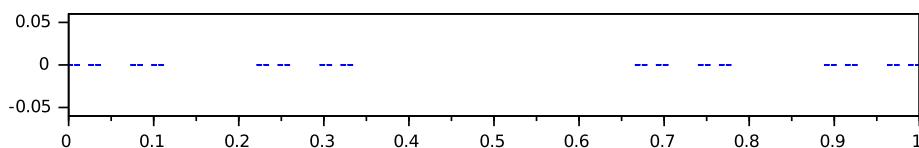
4
5 function Y=f1 (X)
6     Y=[1/3  0;0  1/3]*X+[2/3;  0];
7 endfunction

8
9 N=input("Entrez N, le nombre de points a dessiner : ");
10 C=zeros(2,N); C(:,1)=[0;0]; //initialisation de l'ensemble de N points
11
12 for i=2:N
13     t=floor(2*rand(1)+1); // selection aleatoire de la fonction
14     select t;
15     case 1 then
16         C(:, i)=f0 (C(:, i-1));
17     case 2 then
18         C(:, i)=f1 (C(:, i-1));
19     end
20 end
21
22 clf;
23 isoview(0 ,1 , -0.05 ,0.05);
24 plot(C(1,: ),C(2,: ),".", 'markersize',1); // affichage

```

L'exécution du code nous donne ainsi la figure 2.5.:

Figure 2.5: Ensemble de Cantor itératif pour 1000 points
Ensemble de Cantor pour 1000 points (méthode récursive)



2.4 Triangle de Sierpinski

2.4.1 Théorie

Présentation

Notre analyse continue avec l'étude d'une seconde fractale construite par Waclaw Sierpinski, mathématicien polonais, fractale nommée *le triangle de Sierpinski*.

Figure 2.6: Triangle de Sierpinski

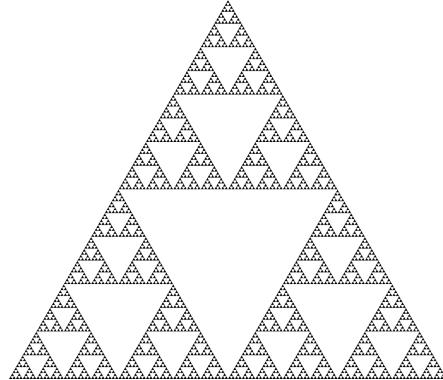


Figure 2.7: Sierpinski 1882-1924

Définition algorithmique

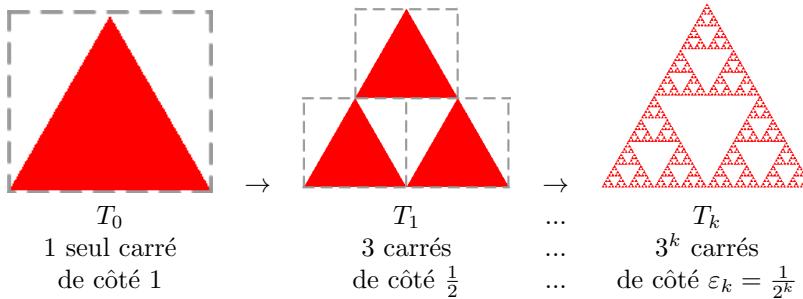
Initialisation Un triangle équilatéral de côté 1

Opération de base : Partager le triangle en 4 triangles équilatéraux égaux et ne pas retenir le triangle central

Itération : Itérer l'opération sur chacun des triangles retenus

Dimension du Triangle de Sierpinski

On appelle T^k la figure obtenue à l'itération k . On applique donc la définition de la dimension de Hausdorff au triangle de Sierpinski :



Donc $d = \lim_{k \rightarrow \infty} \frac{\ln(3^k)}{\ln(2^k)} = \frac{\ln(3)}{\ln(2)}$ est la dimension de Hausdorff du fractal.

2.4.2 Application dans *Scilab* (méthode récursive)

Le programme récursif s'inspire tout autant de la définition même de la fractale que celui pour l'ensemble de Cantor. On implémente donc dans *Scilab* le code 2.3.

Code 2.3: Triangle de Sierpinski (méthode récursive)

```

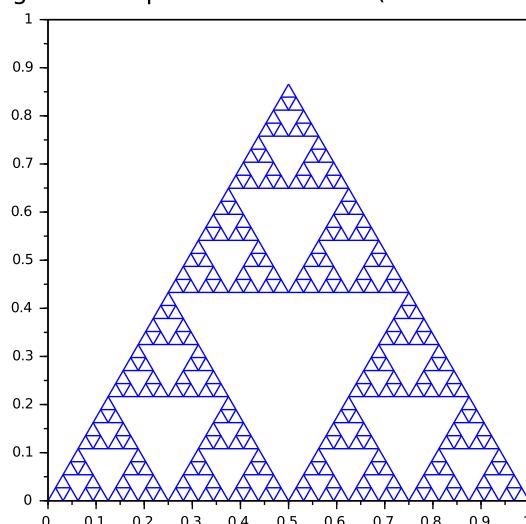
1 function sierpinski(n,a,b,c)
2   if n<1
3     break;
4   else
5     a1=(c+b)/2; b1=(c+a)/2; c1=(b+a)/2; // calcul des nouveaux sommets
6     //trace du triangle central
7     plot ([a1(1),b1(1)], [a1(2),b1(2)]);
8     plot ([b1(1),c1(1)], [b1(2),c1(2)]);
9     plot ([a1(1),c1(1)], [a1(2),c1(2)]);
10    //appel de sierpinski pour les triangles retenus
11    sierpinski(n-1,a,c1,b1);
12    sierpinski(n-1,c1,b,a1);
13    sierpinski(n-1,b1,a1,c);
14  end
15 endfunction
16
17 clf;
18 a=[0,0]'; b=[1/2, sqrt(3)/2]'; c=[1,0]'; //sommets de T0
19 //trace de T0
20 plot ([a(1),b(1)], [a(2),b(2)]);
21 plot ([b(1),c(1)], [b(2),c(2)]);
22 plot ([a(1),c(1)], [a(2),c(2)]);
23
24 N=input("Entrez n, le nombre de niveaux à dessiner : ");
25 sierpinski(N,a,b,c)
26
27 isoview (0,1,0,1);

```

L'exécution du code nous donne ainsi la figure 2.8.

Figure 2.8: Triangle de Sierpinski (récuratif) pour n=5

Triangle de Sierpinski à 5 niveaux (méthode récursive)



2.4.3 Application dans *Scilab* (méthode itérative)

La complexité de l'algorithme récursif est ici en $O(3^n)$ (on réalise 3 appels récursifs pour chaque niveau). Comme pour l'ensemble de Cantor, cette complexité est surmontable pour des valeurs de niveaux relativement faibles ($n \leq 7$), mais pour des valeurs de n importantes, cette méthode n'est plus applicable. On implémente donc également une méthode itérative à l'aide d'un IFS. On a donc :

$$\begin{cases} f_0 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ f_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ 0 \end{bmatrix} \\ f_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{4} \\ \frac{\sqrt{3}}{4} \end{bmatrix} \end{cases} \quad (2.4)$$

d'où $\left\{ \begin{array}{l} T_0 \text{ donné} \\ T_{n+1} : f(T_n) \text{ avec } f(T) = f_0(T) \cup f_1(T) \cup f_2(T) \end{array} \right.$

Finalement T_n converge au sens de la distance de Hausdorff vers T^∞ , le triangle de Sierpinski. A nouveau, la dispersion des points doit être uniforme, on utilise donc des conditions d'équiprobabilité.

On implémente donc dans *Scilab* le code 2.4.

Code 2.4: Triangle de Sierpinski (méthode itérative)

```

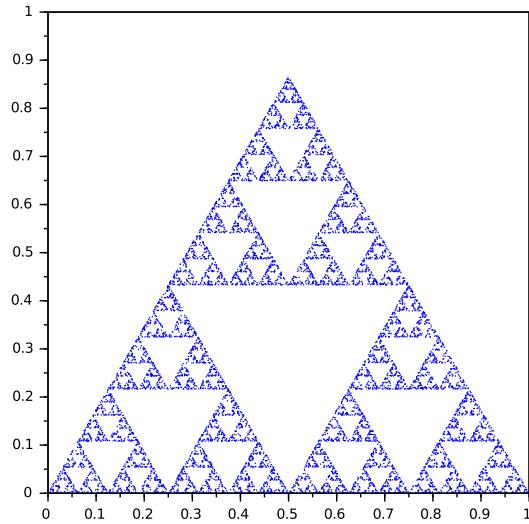
1  function Y=f0(X)
2      Y=[1/2 0;0 1/2]*X;
3  endfunction
4
5  function Y=f1(X)
6      Y=[1/2 0;0 1/2]*X+[1/2;0];
7  endfunction
8
9  function Y=f2(X)
10     Y=[1/2 0;0 1/2]*X+[1/4;sqrt(3)/4];
11 endfunction
12
13 N=input("Entrez n, le nombre de points a dessiner : ");
14 T=zeros(2,N); T(:,1)=[0,0]'; //initialisation de l'ensemble de N points
15
16 for i=2:N
17     t=floor(3*rand(1)+1); //selection aleatoire de la fonction
18     select t
19     case 1 then
20         T(:,i)=f0(T(:,i-1));
21     case 2 then
22         T(:,i)=f1(T(:,i-1));
23     case 3 then
24         T(:,i)=f2(T(:,i-1));
25     end
26 end
27
28 clf;
29 isoview(0,1,0,1);
30 plot(T(1,:),T(2,:),".",'markersize',1); //affichage

```

L'exécution du code nous donne ainsi la figure 2.9.

Figure 2.9: Triangle de Sierpinski itératif pour 1000 points

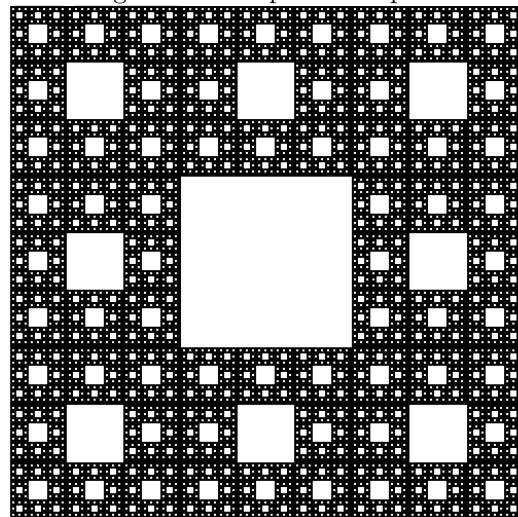
Triangle de Sierpinski pour 10000 points (méthode récursive)



2.4.4 Variante : le tapis de Sierpinski

On a pu travailler sur une variante du triangle de Sierpinski : *Le tapis de Sierpinski*.

Figure 2.10: Tapis de Sierpinski



Largement similaire au triangle, on implémente le code 2.5 (méthode récursive).

Code 2.5: Tapis de Sierpinski (méthode récursive)

```

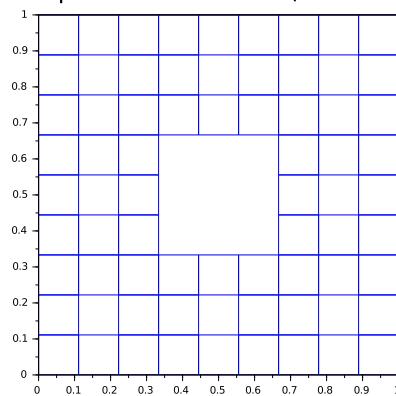
1  function tapis(N,a,b,c,d)
2    if N<1 then
3      plot([a(1),b(1)],[a(2),b(2)]); plot([b(1),c(1)],[b(2),c(2)]);
4      plot([c(1),d(1)],[c(2),d(2)]); plot([d(1),a(1)],[d(2),a(2)]);
5    else
6      // calcul des nouveaux sommets
7      e=[a(1),a(2)+2*(d(2)-a(2))/3];
8      f=[a(1),a(2)+(d(2)-a(2))/3];
9      g=[a(1)+(b(1)-a(1))/3,a(2)];
10     h=[a(1)+2*(b(1)-a(1))/3,a(2)];
11     i=[b(1),a(2)+(c(2)-b(2))/3];
12     j=[b(1),a(2)+2*(c(2)-b(2))/3];
13     k=[a(1)+2*(b(1)-a(1))/3,c(2)];
14     l=[a(1)+(b(1)-a(1))/3,c(2)];
15     m=[a(1)+(b(1)-a(1))/3,a(2)+2*(d(2)-a(2))/3];
16     n=[a(1)+(b(1)-a(1))/3,a(2)+(d(2)-a(2))/3];
17     o=[a(1)+2*(b(1)-a(1))/3,a(2)+(c(2)-b(2))/3];
18     p=[a(1)+2*(b(1)-a(1))/3,a(2)+2*(c(2)-b(2))/3];
19     // appels de tapis pour chacun des carres retenus
20     tapisSierpinski(N-1,a,g,n,f);
21     tapisSierpinski(N-1,f,n,m,e);
22     tapisSierpinski(N-1,e,m,l,d);
23     tapisSierpinski(N-1,l,k,p,m);
24     tapisSierpinski(N-1,p,j,c,k);
25     tapisSierpinski(N-1,j,p,o,i);
26     tapisSierpinski(N-1,b,h,o,i);
27     tapisSierpinski(N-1,n,o,h,g);
28   end
29 endfunction
30
31 A=[0,0]; B=[1,0]; C=[1,1]; D=[0,1]; // points du premier carre
32
33 N=input("Entrez n, le nombre de niveaux à dessiner : ");
34 tapis(N,A,B,C,D)
35 clf isoview(0,1,0,1); // affichage

```

L'exécution du code nous donne ainsi la figure 2.11.

Figure 2.11: Tapis de Sierpinski (récuratif) pour n=2

Tapis de Sierpinski à 2 niveaux (méthode récursive)



On obtient donc un algorithme récursif de complexité en $O(8^n)$, complexité difficilement supportable pour la machine (mon ordinateur ne parvient à dessiner dans un temps raisonnable que le niveau $n = 2\dots$). On implémente donc également la méthode itérative avec le code 2.6 suivant l'IIFS :

$$\left\{ \begin{array}{l} f_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ f_3 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + 2 \times \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} \\ f_5 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + 2 \times \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} \\ f_7 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} + 2 \times \begin{bmatrix} 0 \\ \frac{1}{3} \end{bmatrix} \\ f_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} \\ f_4 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{3} \end{bmatrix} \\ f_6 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} \\ f_8 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + 2 \times \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} + 2 \times \begin{bmatrix} 0 \\ \frac{1}{3} \end{bmatrix} \end{array} \right. \quad (2.5)$$

Code 2.6: Tapis de Sierpinski (méthode itérative)

```

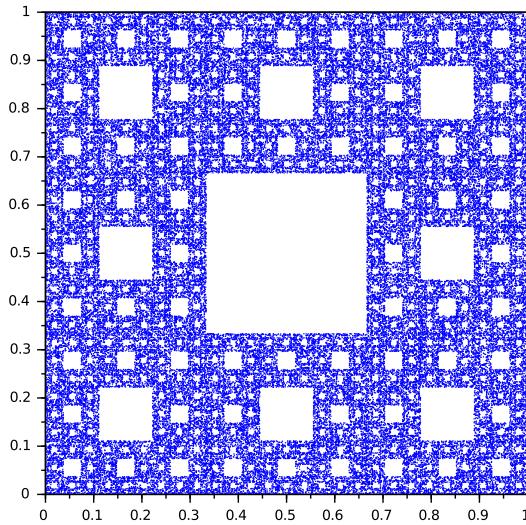
1  function tapis(N,a,b,c,d)
2    if N<1 then
3      plot([a(1),b(1)],[a(2),b(2)]); plot([b(1),c(1)],[b(2),c(2)]);
4      plot([c(1),d(1)],[c(2),d(2)]); plot([d(1),a(1)],[d(2),a(2)]);
5    else
6      // calcul des nouveaux sommets
7      e=[a(1),a(2)+2*(d(2)-a(2))/3];
8      f=[a(1),a(2)+(d(2)-a(2))/3];
9      g=[a(1)+(b(1)-a(1))/3,a(2)];
10     h=[a(1)+2*(b(1)-a(1))/3,a(2)];
11     i=[b(1),a(2)+(c(2)-b(2))/3];
12     j=[b(1),a(2)+2*(c(2)-b(2))/3];
13     k=[a(1)+2*(b(1)-a(1))/3,c(2)];
14     l=[a(1)+(b(1)-a(1))/3,c(2)];
15     m=[a(1)+(b(1)-a(1))/3,a(2)+2*(d(2)-a(2))/3];
16     n=[a(1)+(b(1)-a(1))/3,a(2)+(d(2)-a(2))/3];
17     o=[a(1)+2*(b(1)-a(1))/3,a(2)+(c(2)-b(2))/3];
18     p=[a(1)+2*(b(1)-a(1))/3,a(2)+2*(c(2)-b(2))/3];
19     // appels de tapis pour chacun des carres retenus
20     tapisSierpinski(N-1,a,g,n,f);
21     tapisSierpinski(N-1,f,n,m,e);
22     tapisSierpinski(N-1,e,m,l,d);
23     tapisSierpinski(N-1,l,k,p,m);
24     tapisSierpinski(N-1,p,j,c,k);
25     tapisSierpinski(N-1,j,p,o,i);
26     tapisSierpinski(N-1,b,h,o,i);
27     tapisSierpinski(N-1,n,o,h,g);
28   end
29 endfunction
30
31 A=[0,0]; B=[1,0]; C=[1,1]; D=[0,1]; // points du premier carre
32
33 N=input("Entrez n, le nombre de niveaux a dessiner : ");
34 tapis(N,A,B,C,D)
35 clf isoview(0,1,0,1); // affichage

```

L'exécution du code nous donne ainsi la figure 2.12.

Figure 2.12: Tapis de Sierpinski (itératif) pour n=100000 points

Tapis de Sierpinski pour 100000 points (méthode itérative)



2.5 Flocon de neige de Von Koch

2.5.1 Théorie

Présentation

Notre analyse continue avec l'étude d'une troisième fractale construite par Helge Von Koch, mathématicien suédois, fractale nommée le *flocon de Von Koch*.

Figure 2.13: Flocon de neige de Von Koch

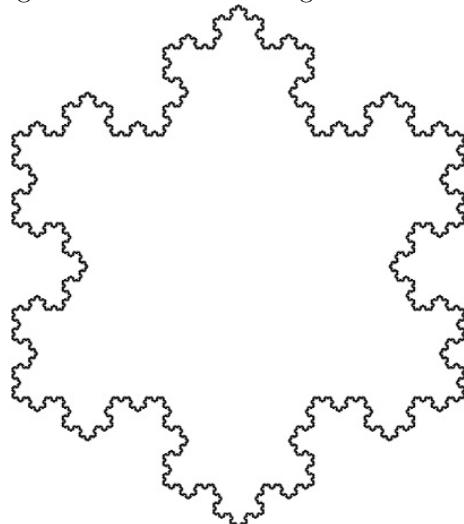


Figure 2.14: Von Koch 1870-1924

Définition algorithmique

Initialisation Un triangle équilatéral de côté 1

Opération de base : Chaque segment du triangle est partagé en 3 parties égales.

Le segment central S_C est remplacé par 2 segments égaux formant un triangle équilatéral ayant pour base S_C

Itération : Itérer l'opération sur chacun des segments obtenus

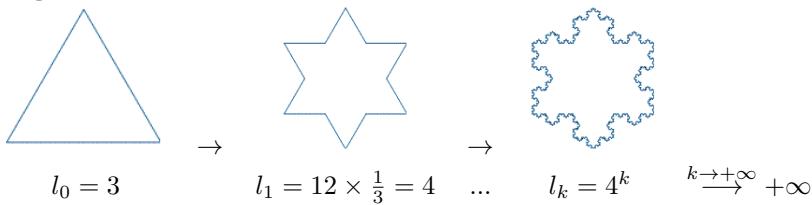
Remarques

A l'infini, on obtient donc $K^{(\infty)}$ le flocon de Van Koch. On a en outre les propriétés suivantes :

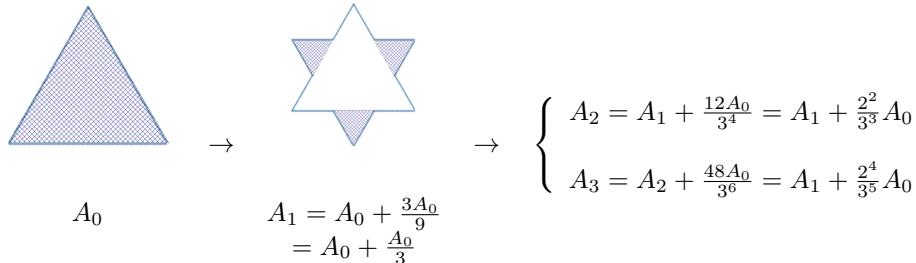
- i) $l(K^{(\infty)}) = +\infty$
- ii) $A(K^{(\infty)}) < \infty$

En effet :

longueur :



aire :

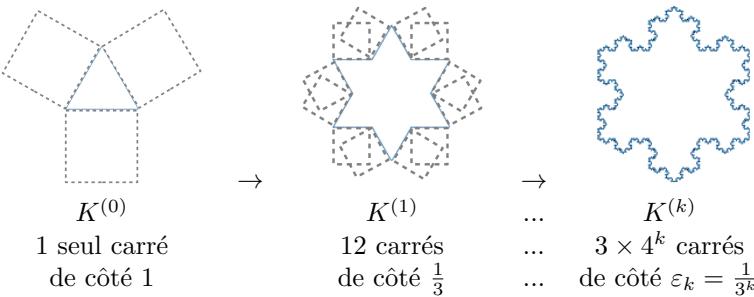


$$\text{donc } A_k = A_0 + A_0 \frac{1}{3} + A_0 \frac{2^2}{3^3} + A_0 \frac{2^4}{3^5} + \cdots + A_0 \frac{2^{2k}}{3^{2k+1}}$$

$$\text{à l'infini, on a donc : } A_\infty = A_0 + \sum_{k=0}^{\infty} \frac{2^{2k}}{3^{2k+1}} = A_0 + \frac{A_0}{3} \sum_{k=0}^{\infty} \left(\frac{2}{3}\right)^{2k} = ?$$

Dimension du Flocon de Von Koch

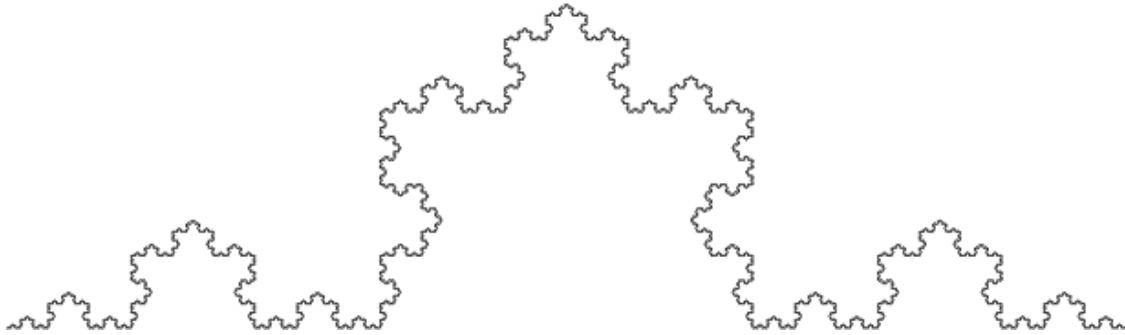
On applique donc la définition de la dimension de Hausdorff au flocon de Von Koch :



Donc $d = \lim_{k \rightarrow \infty} 3 \times \frac{\ln(4^k)}{\ln(3^k)} = 3 \times \frac{\ln(4)}{\ln(3)} = 3 \times \frac{2\ln(2)}{\ln(3)}$ est la dimension de Hausdorff.

On peut restreindre l'étude du flocon de Von Koch à l'étude de la courbe de Von Koch (figure 2.15). L'algorithme de construction est le même à l'exception que la figure de départ est un segment et non un triangle. On a alors $d = \frac{2\ln(2)}{\ln(3)}$.

Figure 2.15: Courbe de Von Koch



Dans nos applications dans *Scilab*, on se contentera de tracer la courbe de Von Koch.

2.5.2 Application dans *Scilab* (méthode récursive)

En s'inspirant toujours de la définition de la fractale, on implémente dans *Scilab* le code 2.7.

Code 2.7: Courbe de Von Koch (méthode récursive)

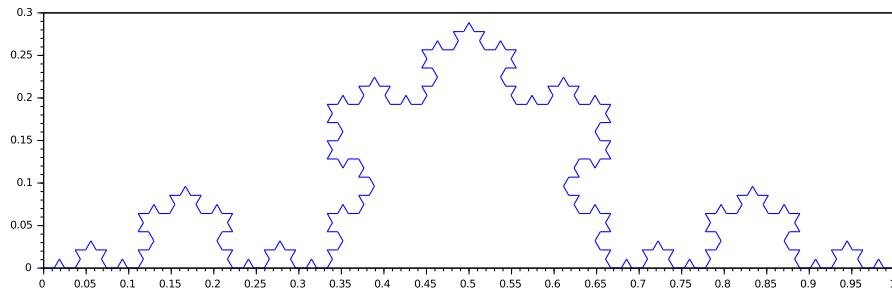
```

1 // utile pour le calcul de e (troisième point)
2 function K=rotation(theta)
3     K=[cos(theta), -sin(theta);
4         sin(theta), cos(theta)];
5 endfunction
6
7 function koch(n,a,b)
8     if n<1
9         plot([a(1),b(1)], [a(2),b(2)]);
10    else
11        // calcul des nouveaux points
12        c=a+(b-a)/3;
13        d=b-(b-a)/3;
14        e=c+rotation(%pi/3)*(d-c);
15        // appel de sierpinski pour les segments retenus
16        koch(n-1,a,c);
17        koch(n-1,c,e);
18        koch(n-1,e,d);
19        koch(n-1,d,b);
20    end
21 endfunction
22
23 clf;
24 a=[0,0]'; b=[1,0]'; // extrémités du segment initial
25
26 N=input("Entrez n, le nombre de niveaux à dessiner : ");
27 koch(N,a,b)
28 isoview(0,1,0,0.3);

```

L'exécution du code nous donne ainsi la figure 2.16.

Figure 2.16: Figure pour n=4
Courbe de Von Koch à 4 niveaux (méthode récursive)



2.5.3 Application dans *Scilab* (méthode itérative)

La complexité de l'algorithme récursif est ici en $O(4^n)$ (on réalise 4 appels récursifs pour chaque niveau). L'exécution devient donc très rapidement problématique. Pour des valeurs de n importantes, cette méthode n'est plus applicable. On implémente donc également une méthode itérative à l'aide d'un IFS. On a donc :

$$\left\{ \begin{array}{l} f_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ f_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{1}{6} & -\frac{\sqrt{3}}{6} \\ \frac{\sqrt{3}}{6} & \frac{1}{6} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} \\ f_3 \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} \frac{1}{6} & \frac{\sqrt{3}}{6} \\ -\frac{\sqrt{3}}{6} & \frac{1}{6} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{2} \\ \frac{\sqrt{3}}{6} \end{bmatrix} \\ f_4 \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{3} \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{2}{3} \\ 0 \end{bmatrix} \end{array} \right. \quad (2.6)$$

d'où $\left\{ \begin{array}{l} K_0 \text{ donné} \\ K_{n+1} = K(T_n) \text{ avec } K(T) = f_0(T) \cup f_1(T) \cup f_2(T) \cup f_3(T) \cup f_4(T) \end{array} \right.$

Finalement K_n converge au sens de la distance de Hausdorff vers K^∞ , le flocon de Von Koch. A nouveau, la dispersion des points doit être uniforme, on utilise donc des conditions d'équiprobabilité.

On implémente donc dans *Scilab* le code 2.8.

Code 2.8: Flocon de Von Koch (méthode itérative)

```

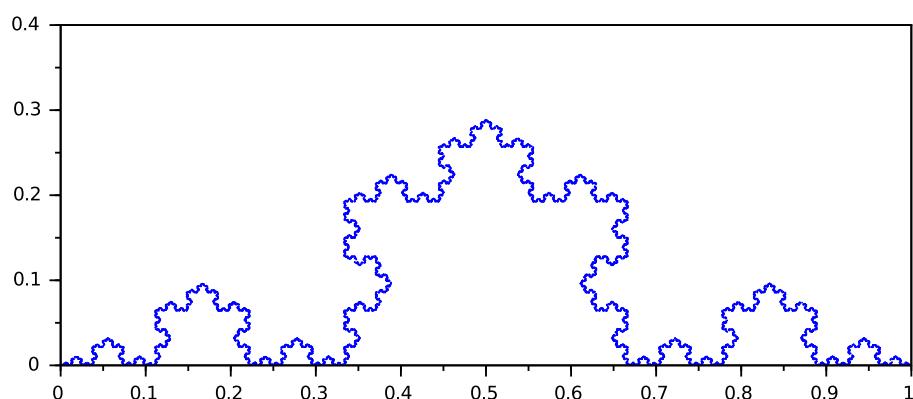
1 function Y=f1(X) //partie gauche basse (reduction 1/3)
2   Y=[1/3 ,0;0 ,1/3]*X
3 endfunction
4
5 function Y=f2(X) //partie gauche haute (reduction 1/3 et rotation)
6   Y=[1/6,-sqrt(3)/6;sqrt(3)/6 ,1/6]*X+[1/3;0]
7 endfunction
8
9 function Y=f3(X) //partie droite haute (reduction 1/3 et rotation)
10  Y=[1/6,sqrt(3)/6;-sqrt(3)/6 ,1/6]*X+[1/2;sqrt(3)/6]
11 endfunction
12
13 function Y=f4(X) //partie droite basse (reduction 1/3)
14  Y=[1/3 ,0;0 ,1/3]*X+[2/3;0]
15 endfunction
16
17 N=input("Entrez N, le nombre de points a dessiner :");
18 X=zeros(2,N); X(:,1)=[0 ,0]'; //initialisation de l'ensemble de N points
19
20 for i=2:N
21   t=floor(4*rand(1)+1); //selection aleatoire de la fonction
22   select t
23   case 1 then X(:,i)=f1(X(:,i-1));
24   case 2 then X(:,i)=f2(X(:,i-1));
25   case 3 then X(:,i)=f3(X(:,i-1));
26   case 4 then X(:,i)=f4(X(:,i-1));
27   end
28 end
29
30 clf isoview(0 ,1 ,0 ,0.4); plot(X(1,:),X(2,:),".",'markersize',1); //affichage

```

L'exécution du code nous donne ainsi la figure 2.17.

Figure 2.17: Flocon de Von Koch itératif pour 10000 points

Courbe de Von Koch pour 10000 points (méthode itérative)



2.6 Fougère de Barnsley

2.6.1 Théorie

Présentation

Pour compléter notre étude sur la théorie des IFS, on va s'intéresser à un objet qu'il est impossible de construire de manière récursive : la *fougère de Barnsley*, construite par Michael Barnsley, mathématicien britannique.

On a précédemment vu trois applications de la méthode des IFS avec à chaque fois une dispersion des points de manière uniforme. Il apparaît donc pertinent de s'intéresser à une fractale nécessitant une certaine dispersion des points : *la fougère de Barnsley*.



Figure 2.18: Barnsley 1946-...

Figure 2.19: Fougère de Barnsley



Définition

Le système de fonctions itérées est le suivant :

$$\begin{cases} f_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0,00 & 0,00 \\ 0,00 & 0,16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ f_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0,85 & 0,04 \\ -0,04 & 0,85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1,6 \end{bmatrix} \\ f_3 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0,20 & -0,26 \\ 0,23 & 0,22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1,6 \end{bmatrix} \\ f_4 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0,15 & 0,28 \\ 0,26 & 0,24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0,44 \end{bmatrix} \end{cases} \quad (2.7)$$

Ces fonctions sont responsables de la modélisation de chacune des parties de la fougère. Pour obtenir un dessin réaliste, il est donc nécessaire de privilégier certaines parties par rapport à d'autres (plus la partie gauche que le reste par exemple).

Il est conseillé de choisir les pondérations p_i suivantes pour chacune des fonctions f_i : $p = \begin{bmatrix} 0,01 \\ 0,85 \\ 0,07 \\ 0,07 \end{bmatrix}$

2.6.2 Application dans *Scilab*

On implémente donc dans *Scilab* le code 2.9.

Code 2.9: Fougère de Barnsley

```

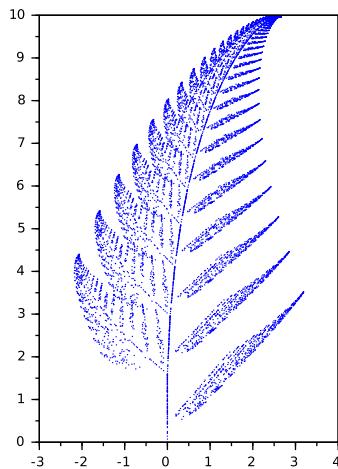
1  function Y=f1(X) //la tige
2      Y=[0 0;0 0.16]*X; endfunction
3
4  function Y=f2(X) //partie gauche
5      Y=[0.85 0.04;-0.04 0.85]*X+[0;1.6]; endfunction
6
7  function Y=f3(X) //partie droite
8      Y=[0.2 -0.26;0.23 0.22]*X+[0;1.6]; endfunction
9
10 function Y=f4(X) //sous-parties
11     Y=[0.15 0.28;0.26 0.24]*X+[0;0.44]; endfunction
12
13 p=[0.01, 0.85, 0.07, 0.07]; //probas conseillées
14 N=input("Entrez N, le nombre de points à dessiner : ");
15 X=zeros(2,N); X(:,1) = [0;0]; //initialisation de l'ensemble de N points
16
17 for i=2:N
18     t=rand(); //sélection aléatoire d'un nombre entre 0 et 1
19     if (t<p(1)) then X(:,i)=f1(X(:,i-1)); end
20     if (t>p(1)) & (t<p(1)+p(2)) then X(:,i)=f2(X(:,i-1)); end
21     if (t>p(1)+p(2)) & (t<p(1)+p(2)+p(3)) then X(:,i)=f3(X(:,i-1)); end
22     if (t>p(1)+p(2)+p(3)) then X(:,i)=f4(X(:,i-1)); end
23 end
24
25 clf; set(gca(),"isoview","on");
26 plot(X(1,:),X(2,:),".",'markersize',1); //affichage

```

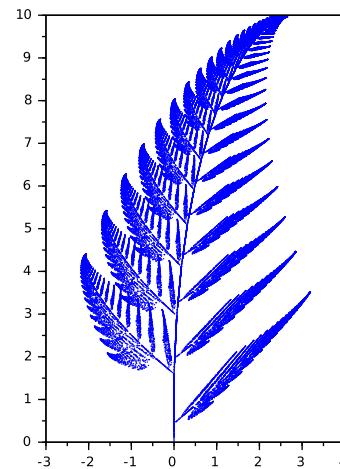
L'exécution du code nous donne ainsi la figure 2.20 pour différentes valeurs de N .

Figure 2.20: Différents affichages de la fougère en fonction de N

Fougere pour $N=10000$



Fougere pour $N=100000$



2.7 Ensemble de Julia

2.7.1 Théorie

Présentation

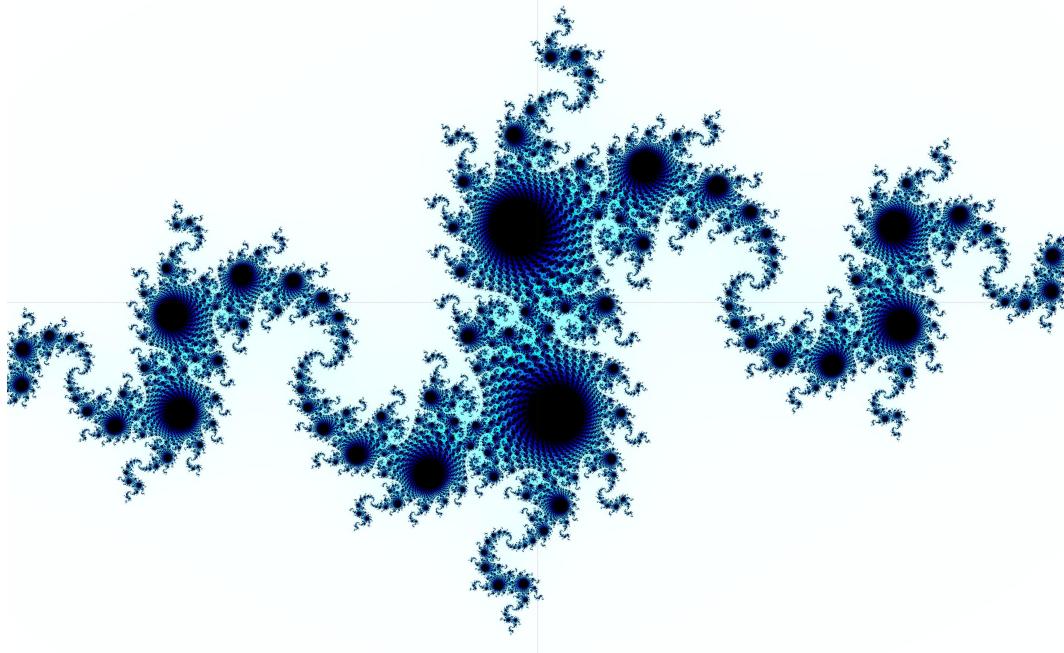
Nous allons maintenant nous intéresser à des fractales dont la construction est beaucoup moins immédiate que les précédentes (fractales dont on pouvait rapidement représenter quelques itérations, même avec un crayon et une feuille de papier).

La première de ces fractales *plus compliquées* s'appelle l'ensemble de Julia, construite par Gaston Julia, un mathématicien français. Il est à noter que la figure 2.24 n'est pas représentative de l'ensemble de Julia : ce dernier peut prendre des formes très différentes selon la modélisation choisie et surtout selon la valeur de c .



Figure 2.21: Julia
1893-1978

Figure 2.22: Ensemble de Julia (exemple parmi beaucoup d'autres)



Définition

On se donne un nombre complexe $c = a + ib$ avec $a, b \in \mathbb{R}$. On considère alors la suite :

$$\begin{cases} z_{n+1} = z_n^2 + c \\ z_0 \in \mathbb{C} \end{cases} \quad (2.8)$$

On appelle *bassin d'attraction* l'ensemble des points z_0 tels que la suite définie en (2.8) converge, et *bassin de répulsion* l'ensemble des points z_0 tels que la suite diverge. L'ensemble de Julia est alors la frontière entre ces deux bassins. L'ensemble de Julia est alors défini en fonction de c , qui en devient un paramètre capital. Le choix de c est classique et on peut trouver beaucoup de documentation en ligne sur les valeurs intéressantes à donner à cette variable. En outre, nous verrons un peu plus tard que ces valeurs ne sont pas le fruit du hasard.

2.7.2 Application dans *Scilab*

On implémente donc dans *Scilab* le code 2.10.

Code 2.10: Ensemble de Julia

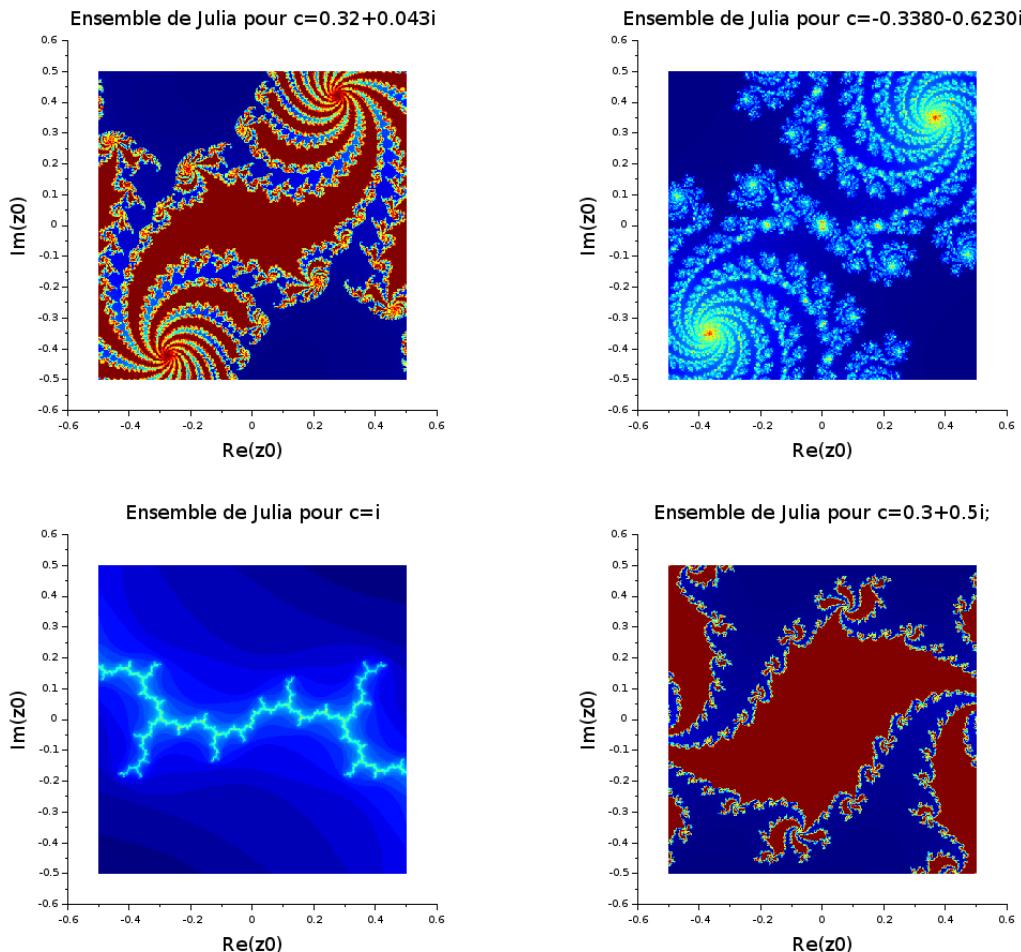
```

1 x=linspace( -0.5,0.5,400); //valeurs de x (partie reelle de )
2 y=linspace( -0.5,0.5,400); //valeurs de y (partie imaginaire de Z)
3 [X,Y]=meshgrid(x,y); Z=X+%i*Y; //on cree une grille de valeur pour Z
4
5 c=0.32+%i*0.043; //valeur de c que l'on peut changer
6 A=zeros(length(x),length(y)); //matrice de couleurs
7
8 for i=1:500
9     Rc=max(abs(c),2); Z=Z.^2+c;
10    conver=abs(Z)<Rc; //analyse de la convergence
11    A(conver)=A(conver)+1;
12 end
13
14 //affichage
15 clf; set(gca(),"isoview","on"); set(gcf(), 'color_map', jetcolormap(256));
16 grayplot(x,y,A);

```

L'exécution du code nous donne ainsi la figure 2.23 pour différentes valeurs de c .

Figure 2.23: Différents affichages de l'ensemble de Julia en fonction de c



2.8 Ensemble de Mandelbrot

2.8.1 Théorie

Présentation

Avec l'ensemble de Julia, nous avions vu que la sélection d'une valeur de c *intéressante* était primordiale à la construction de la fractale.

Ainsi la fractale que nous sommes sur le point d'étudier nous permet une représentation de ces valeurs de c en fonction de leur *intérêt*. Il s'agit donc de la fractale nommée *l'ensemble de Mandelbrot*, construite par Benoît Mandelbrot, mathématicien franco-américain.

Figure 2.24: Ensemble de Mandelbrot

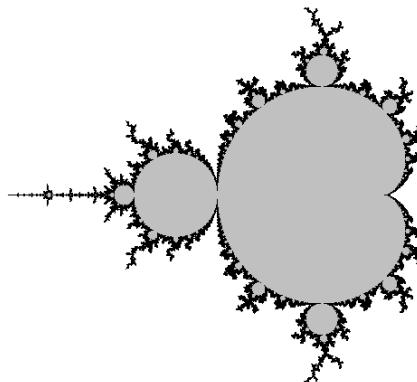


Figure 2.25: Mandelbrot 1924-2010

Définition

La définition de l'ensemble de Mandelbrot est donc largement similaire à celle de l'ensemble de Julia, à la seule différence près que la valeur de z_0 est donnée et que c est une variable.

2.8.2 Application dans Scilab

On implémente donc dans *Scilab* le code 2.11.

Code 2.11: Ensemble de Mandelbrot

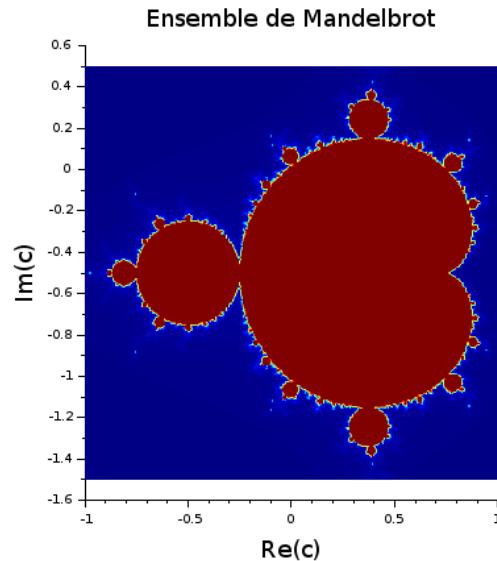
```

1 x=linspace(-1,1,400); //valeurs de x (partie réelle de C)
2 y=linspace(-1.5,0.5,400); //valeurs de y (partie imaginaire de C)
3 [X,Y]=meshgrid(x,y); C=Y+%i*X; //on crée une grille de valeur pour C
4
5 Z=0;
6 A=zeros(length(x),length(y)); //matrice de couleurs
7
8 for i=1:500
9     Rc=max(abs(c),2); Z=Z.^2+C;
10    conver=abs(Z)<Rc; //analyse de la convergence
11    A(conver)=A(conver)+1;
12 end
13
14 //affichage
15 clf; set(gca(),"isoview","on"); set(gcf(), "color_map", jetcolormap(256));
16 grayplot(x,y,A);

```

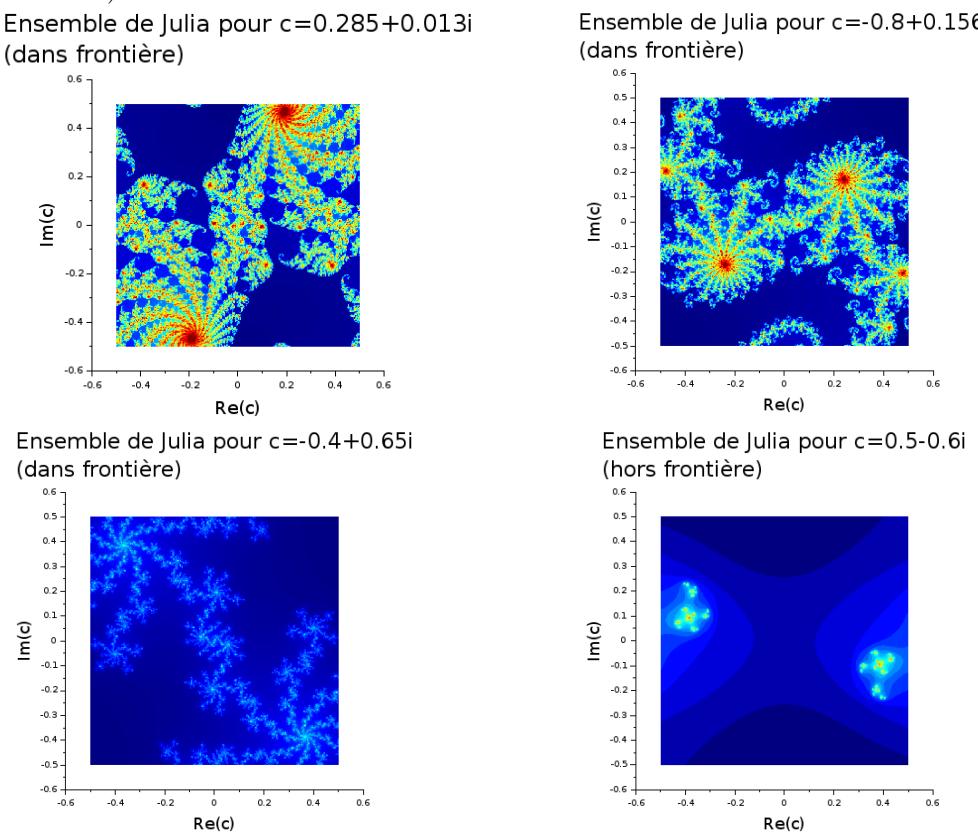
L'exécution du code nous donne ainsi la figure 2.26.

Figure 2.26: Ensemble de Mandelbrot



Ainsi, en construisant les ensembles de Julia associés aux c intéressants (dans la limite entre les zones rouge et bleue), on obtient une figure connexe.

En effet, on peut comparer les affichages de la figure 2.27.

Figure 2.27: Différents affichages de l'ensemble de Julia en fonction de c (dans ou hors la frontière de Mandelbrot)

CHAPITRE 3

ÉQUATIONS DIFFÉRENTIELLES

3.1 Introduction

Dans de très nombreux domaines comme la mécanique, la biologie, la chimie ou encore l'économie, beaucoup de problèmes font appel à des équations différentielles. On est ainsi souvent confronté à ce que l'on appelle des *problème de Cauchy* sous la forme suivante :

$$\begin{cases} y^{(m)} = h(t, y, y', \dots, y^{(m-1)}) \\ y(0) = a_0 \\ y'(0) = a_1 \\ \dots \\ y^{(m-1)}(0) = a_{m-1} \end{cases} \quad (3.1)$$

Pour résoudre (3.1), on se ramène alors à un système différentiel d'ordre 1 :

$$Y = \begin{pmatrix} y_1 & (\text{la solution}) \\ y_2 = y'_1 \\ \vdots \\ y_m = f(t, y_1, y_2, \dots, y_{m-1}) \end{pmatrix} \quad Y(0) = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{m-1} \end{pmatrix} \in \mathbf{R}^m$$

donc $\begin{cases} Y' = F(t, Y) \\ y(0) = (a_0, \dots, a_{m-1})^T \end{cases}$ avec $F(t, Y) = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{m-1} \\ f(t, y_1, y_2, \dots, y_{m-1}) \end{pmatrix}$

(3.2)

Cependant, considérons le problème suivant :

$$\begin{cases} y' = e^{-t^2} y \\ y(0) = 1 \end{cases} \quad (3.3)$$

on a (3.3) $\Rightarrow \frac{dy}{dt} = e^{-t^2} y \Rightarrow \frac{dy}{y} = e^{-t^2} dt \Rightarrow \int \frac{dy}{y} = \int e^{-t^2} dt \Rightarrow \ln(y) = \int e^{-t^2} dt$

Or on ne connaît pas de primitive de $t \rightarrow e^{-t^2}$, on ne peut donc pas résoudre le système (3.3) de manière analytique. Dans ce genre de situation, on a alors recours à un schéma numérique.

3.2 Idée, stabilité, consistance et ordre d'un schéma numérique

3.2.1 Idée

L'idée générale d'un schéma numérique est la suivante :
On crée une subdivision de l'intervalle d'étude $[t_0, t_0 + T]$.

On obtient donc $\sigma = \{t_0, t_1, \dots, t_N\}$ avec $t_N = t_0 + N$.
On note le pas de la subdivision $h = \max_{0 \leq i \leq N} |t_{i+1} - t_i|$.

On se placera toujours dans le cas d'une substitution *uniforme* : $h_i = |t_{i+1} - t_i| = h \quad \forall i$
L'idée est alors d'approcher $y(t_i)$ par z_i où :

$$\begin{cases} z_{i+1} = z_i + h\Phi(t_i, z_i, h) \\ z_0 = y_0 \end{cases} \quad (3.4)$$

Chacune de ces approximations sera alors appelée un *schéma*. Un schéma donné a différentes propriétés remarquables : une stabilité, une consistance et un ordre.

3.2.2 Stabilité d'un schéma numérique

Définition

soit (u_i) et (v_i) telle que :
 $\begin{cases} u_{i+1} = u_i + h\Phi(t_i, u_i, h) \\ u(0) \text{ donné} \end{cases}$ et $\begin{cases} v_{i+1} = v_i + h\Phi(t_i, v_i, h) + \xi_i \\ v(0) \text{ donné} \end{cases} \quad \forall i = 0, \dots, N-1$

Le schéma est stable si $\max_{0 \leq i \leq N-1} |u_i - v_i| \leq C \left(|u_0 - v_0| + \sum_{i=1}^N |\xi_i| \right)$
où C est une constante ne dépendant pas de (u_i) et (v_i) .

Théorème

Le schéma est stable $\Leftrightarrow \exists K > 0, \|\Phi(t, y, h) - \Phi(t, z, h)\| \leq K \|Y - Z\|$
 $\Leftrightarrow \Phi$ est K -lipschitzienne par rapport à la deuxième variable

3.2.3 Consistance d'un schéma numérique

Définition

Le schéma est consistant si $\xi(y) = \sum_i \|y - t_{i+1} - y(t_i) - \Phi(t_i, y(t_i), h)\| \xrightarrow[h \rightarrow 0]{} 0$
ou d'une manière équivalente : si $\Phi(t, y, 0) = f(t, y)$

3.2.4 Ordre d'un schéma numérique

Définition

Le schéma est d'ordre p si $\exists C > 0, \max_{1 \leq i \leq N} |y(t_i) - z_i| \leq Ch^p$

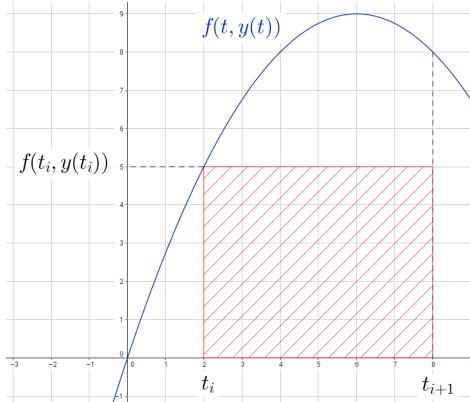
Autre définition

Le schéma est d'ordre p si l'erreur de consistance locale $T_i = \frac{y(t_{i+1}) - y(t_i)}{h} - \Phi(t_i, y(t_i), h)$
est telle que $|T_i| \leq Ch^p$

3.3 Schéma d'Euler

3.3.1 Théorie

Définition



Le théorème fondamental de l'analyse nous donne :
 $y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} y'(t) dt = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t)) dt$
La valeur de $\int_{t_i}^{t_{i+1}} f(t, y(t)) dt$ est inconnue, il faut donc l'approcher.

On l'approche par **l'aire du rectangle gauche** :
 $y(t_{i+1}) \sim y(t_i) + \text{Aire du rectangle gauche}$
 $\sim y(t_i) + h f(t_i, y(t_i))$

D'où le schéma d'Euler :

$$\begin{cases} z_{i+1} = z_i + h f(t_i, z_i) \\ z_0 \text{ donné}, z_0 = y_0 \end{cases} \quad (3.5)$$

Propriétés

Le schéma d'Euler est **stable**, **consistant** et **d'ordre 1** (si f est L-lipschitzienne par rapport à la 2^{ème} variable).

Démo (stabilité) :

On pose $\Phi(t, y, h) = f(t, y)$, ainsi $\|\Phi(t, y, h) - \Phi(t, z, h)\| = \|f(t, y) - f(t, z)\|$

On a f L-lipschitzienne par rapport à la 2^{ème} variable, alors $\|f(t, y) - f(t, z)\| \leq L\|y - z\|$

Démo (consistance) :

On pose $\Phi(t, y, h) = f(t, y) \forall h$ donc on a en particulier $\Phi(t, y, 0) = f(t, y)$

Démo (ordre) :

On a $\begin{cases} y_{i+1} = y_i + h f(t_i, y_i) \\ y(t_{i+1}) = y(t_i + h) \stackrel{\text{Taylor-Lagrange}}{=} y(t_i) + h \underbrace{y'(t_i)}_{f(t_i, y_i)} + \frac{h^2}{2} y''(\xi_i) \end{cases}$

d'où $\|y(t_{i+1}) - y_{i+1}\| = (y(t_i) - y_i) + h[f(t_i, y(t_i)) - f(t_i, y_i)] + \frac{h^2}{2} y''(\xi_i)$.

On a f L-lipschitzienne par rapport à la 2^{ème} variable : $\exists L > 0, \|f(t, y) - f(t, z)\| \leq L\|y - z\|$.

On suppose que $\exists M > 0$ tel que $\|y''(\xi)\| < M, \forall \xi \in [t_0, t_0 + T]$,

d'où, si on note $e_i = y(t_i) - z_i$, on a :

$$\begin{aligned} \|e_{i+1}\| &\leq \underbrace{(1 + hL)}_C \|e_i\| + \underbrace{\frac{h^2}{2} M}_D \\ \|e_{i+1}\| &\leq C \|e_i\| + D \\ &\leq C[C \|e_{i-1}\| + D] + D = C^2 \|e_{i-1}\| + D(1 + C) \\ &\leq C^{i+1} e_0 + D[1 + C + \dots + C^{i-1}] = C^i e_0 + D \frac{C^i - 1}{C - 1} \end{aligned}$$

Si $y(t_0) = y_0$ alors $e_0 = 0$. Donc :

$$\begin{aligned} \|e_{i+1}\| &\leq \frac{h^2}{2} M \frac{(1+hL)^i - 1}{(1+hL)-1} \\ &\leq \frac{h}{2L} M [(1 + hL)^i - 1] \end{aligned}$$

On utilise $(1 + x)^m \leq e^{xm}, \forall m, \forall x > 0$:

$$\begin{aligned} \|e_{i+1}\| &\leq h \frac{M}{2L} [e^{ihL} - 1] = h \frac{M}{2L} [e^{L(t_i - t_0)}] \\ &\leq [\frac{M}{2L} (e^T - 1)] h \\ &\leq Ch \end{aligned}$$

3.3.2 Application dans *Scilab*

Comme exemple d'application des différents schémas, nous allons donc considérer une équation différentielle particulière que nous sommes capable de résoudre analytiquement. Ainsi, nous pourrons comparer la valeur obtenue par approximation avec la solution exacte. On considère ainsi l'équation différentielle (3.6).

$$\begin{cases} y' = -ty + t & t \in [0, 4] \\ y(0) = 0 \end{cases} \quad (3.6)$$

La résolution analytique du système (3.6) est la suivante :

Solution homogène : L'équation homogène est donc $y'_h = -ty_h$ qui a pour solution $y_h = Ce^{-\frac{t^2}{2}}$

Solution particulière : Par la méthode de la variation de la constante, on a :

$$C'(t)e^{-\frac{t^2}{2}} = t \Rightarrow C'(t) = te^{\frac{t^2}{2}} \Rightarrow C = e^{\frac{t^2}{2}} \text{ donc } y_p = 1$$

Solution générale : La solution générale est donc $y = y_h + y_p = Ce^{-\frac{t^2}{2}} + 1$

Nous implémentons le schéma d'Euler afin d'approcher cette solution. On implémente donc dans *Scilab* le code 3.1.

Code 3.1: Schéma d'Euler

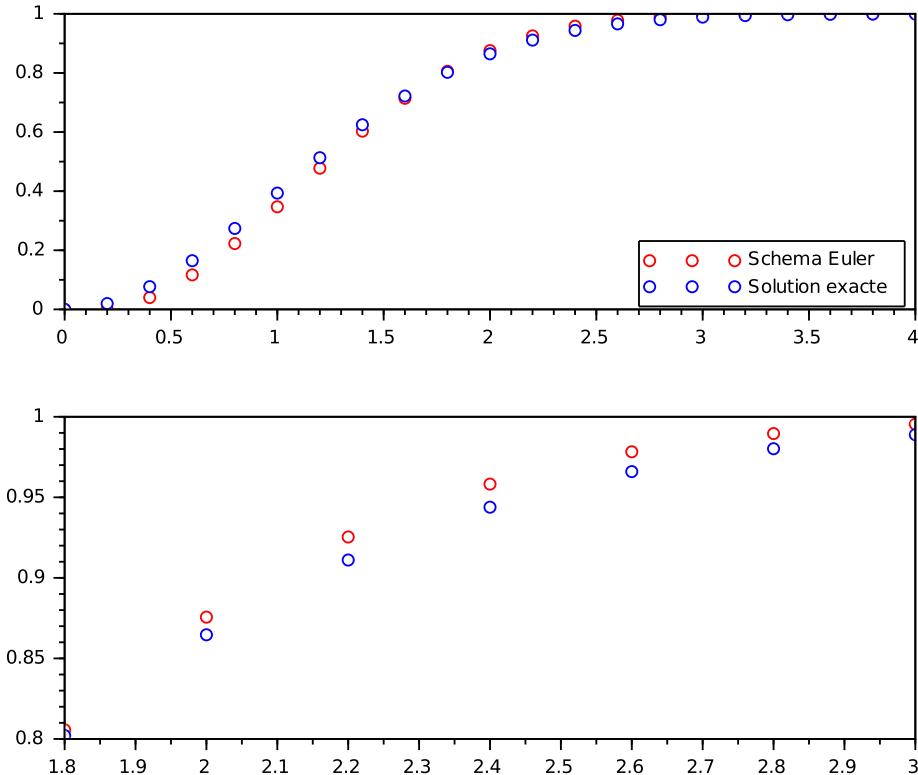
```

1 function dydt=f1(t,y)
2     dydt = -t*y+t;
3 endfunction
4
5 function y=euler(y0,t,f)
6     n=length(t);
7     h=t(2)-t(1);
8     y(1)=y0;
9     for i=1:n-1
10        y(i+1)=y(i)+h*f(t(i),y(i));
11    end
12 endfunction
13
14 function y = solution_exacte(t)
15     y = 1-exp(-t^2/2);
16 endfunction
17
18 T=4; N=20; //intervalle et nombre de points
19 h=T/N; //pas
20 t=[0:h:T]; //subdivision
21 y0=0;
22 y=euler(0,t,f1); //approximation par le schema
23
24 clf;
25 subplot(2,1,1);
26 plot(t,y,'ro');
27 plot(t,solution_exacte(t),'bo');
28 legend(["Schema Euler";"Solution exacte"],opt=4);
29 title("Application du schema d'Euler",'fontsize',5);
30 //on zoom sur certains points pour juger visuellement l'approximation
31 subplot(2,1,2);
32 plot(t(10:16),y(10:16),'ro');
33 plot(t(10:16),solution_exacte(t(10:16)), 'bo');
```

L'exécution du code 3.1 nous donne l'affichage (3.1) (comme commenté dans le code, on réalise deux affichages pour rendre compte au mieux de l'approximation : un affichage large sur l'intervalle $[0; 4]$ et un affichage *zoom* sur $[1, 8; 3]$).

Figure 3.1: Affichage (schéma d'Euler)

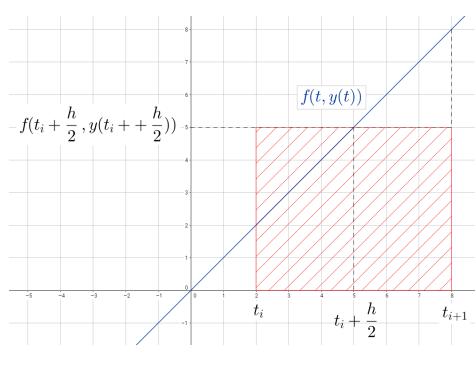
Application du schéma d'Euler



3.4 Schéma du point milieu

3.4.1 Théorie

Définition



On rappelle que l'on a :
 $y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} y' dt = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t)) dt$

On approche $\int_{t_i}^{t_{i+1}} f(t, y(t)) dt$ par l'aire du **rectangle du point milieu** :

$$\begin{aligned} y(t_{i+1}) &\sim y(t_i) + hf\left(t_i + \frac{h}{2}, y\left(t_i + \frac{h}{2}\right)\right) \\ &\sim y(t_i) + hf\left(t_i + \frac{h}{2}, y(t_i) + \frac{h}{2}f(t_i, y(t_i))\right) \end{aligned}$$

D'où le schéma dit du point milieu :

$$\begin{cases} z_{i+1} = z_i + hf\left(t_i + \frac{h}{2}, z_i + \frac{h}{2}f(t_i, z_i)\right) \\ z_0 \text{ donné}, z_0 = y_0 \end{cases} \quad (3.7)$$

Propriétés

Le schéma du point milieu est **stable**, **consistant** et **d'ordre 2**.
 (si f est L-lipschitzienne par rapport à la 2^{ème} variable)

Démo (stabilité) :

On pose $\Phi(t, y, h) = f(t + \frac{h}{2}, y + \frac{h}{2}f(t, y))$, ainsi :

$$\begin{aligned} \|\Phi(t, y, h) - \Phi(t, z, h)\| &= \|f(t + \frac{h}{2}, y + \frac{h}{2}f(t, y)) - f(t + \frac{h}{2}, z + \frac{h}{2}f(t, z))\| \\ &\leq L|y + \frac{h}{2}f(t, y) - z - \frac{h}{2}f(t, z)| \\ &\leq L \left[|y - z| + \frac{h}{2}|f(y, y) - f(t, z)| \right] \\ &\leq |y - z|(L + \frac{h}{2}L^2) \\ &\leq C|y - z| \end{aligned}$$

Ce qui montre que le schéma du point milieu est stable.

Démo (consistance) :

On a $\Phi(t, y, 0) = f(t, y)$

Ce qui montre que le schéma du point milieu est consistant.

Démo (ordre) :

$$T_i = \frac{y(t_{i+1}) - y(t_i)}{h} - \Phi(t, y(t_i), h)$$

On a d'une part :

$$\begin{aligned} y(t_{i+1}) &= y(t_i + h) = y(t_i) + hy'(t_i) + \frac{h^2}{2}y''(t_i) + \frac{h^3}{3!}y^{(3)}(\xi_i) \text{ pour } t_i < \xi_i < t_{i+1} \\ \frac{y(t_i + h) - y(t_i)}{h} &= y'(t_i) + \frac{h}{2}y''(t_i) + \frac{h^2}{3!}y^{(3)}(\xi_i) \\ &= f(t_i, y(t_i)) + \frac{h}{2} \frac{d}{dt}f(t, y(t))|_{t=t_i} + \frac{h^2}{6}y^{(3)}(\xi_i) \\ &= f(t_i, y(t_i)) + \frac{h}{2} \left[\frac{df}{dt}(t, y(t)) + \frac{df}{dy}y'(t) \right] |_{t=t_i} + \frac{h^2}{6}y^{(3)}(\xi_i) \\ &= f(t_i, y(t_i)) + \frac{h}{2} \left[\frac{df}{dt}(t_i, y(t_i)) + \frac{df}{dy}(t_i, y(t_i)) \times f(t_i, y(t_i)) \right] + \frac{h^2}{6}y^{(3)}(\xi_i) \end{aligned}$$

On a d'autre part :

$$\begin{aligned} \Phi(t, y(t_i), h) &= f(t_i + \frac{h}{2}, y(t_i) + \frac{h}{2}f(t_i, y(t_i))) \\ &= f(t_i, y(t_i)) + \frac{h}{2} \left[\frac{df}{dt}(t_i, y(t_i)) + \frac{df}{dy}(t_i, y(t_i)) \times f(t_i, y(t_i)) \right] - O(h^2) \end{aligned}$$

Donc (on soustrait les deux équations obtenues) :

$$T_i = \frac{h^2}{6}y^{(3)}(\xi_i) + O(h^2) \text{ d'où } T_i \leq Ch^{(2)}$$

Ce qui montre que le schéma du point milieu est d'ordre 2.

3.4.2 Application dans *Scilab*

On s'intéresse une nouvelle fois à l'équation différentielle (3.6), à savoir :

$$\begin{cases} y' = -ty + t & t \in [0, 4] \\ y(0) = 0 \end{cases}$$

Dont on rappelle la solution : $y = Ce^{-\frac{t^2}{2}} + 1$

On implémente donc dans *Scilab* le schéma du point milieu avec le code 3.2 qui nous donne l'affichage (3.2).

Code 3.2: Schéma du point milieu

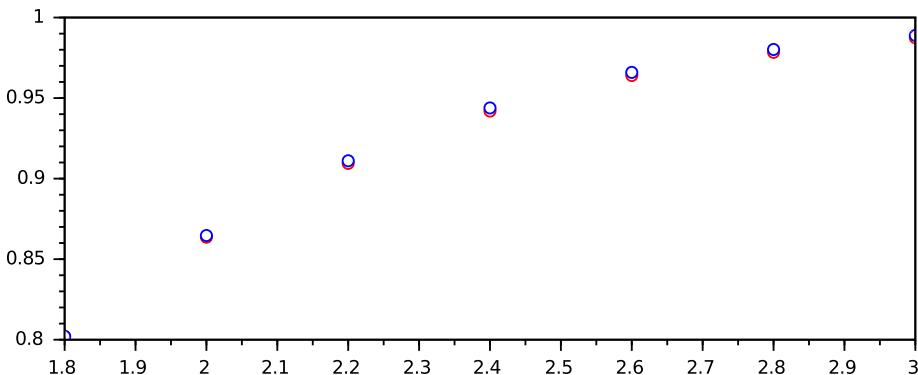
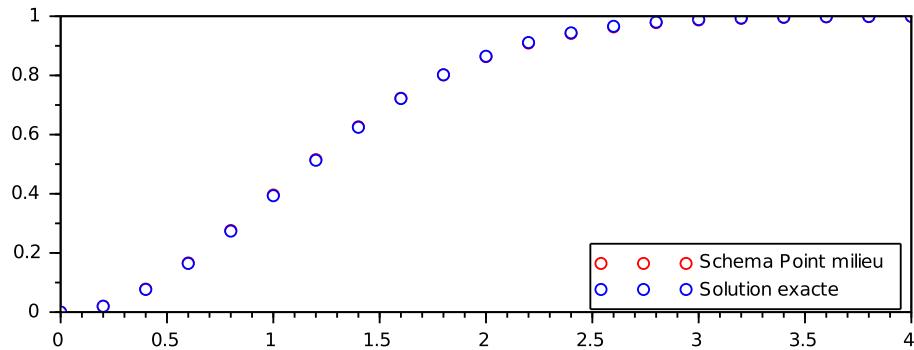
```

1 function dydt=f1(t,y)
2     dydt = -t*y+t; endfunction
3
4 function y=point_milieu(y0,t,f)
5     n=length(t); h=t(2)-t(1);
6     y=zeros(1,n); y(1)=y0;
7     for i=1:n-1
8         k1=f(t(i),y(i));
9         k2=f(t(i)+h/2,y(i)+h/2*k1);
10        y(i+1)=y(i)+h*k2; end endfunction
11
12 function y = solution_exacte(t)
13     y = 1-exp(-t^2/2); endfunction
14
15 T=4; N=20; //intervalle et nombre de points
16 h=T/N; t=[0:h:T]; //pas et subdivision
17 y=point_milieu(0,t,f1); //approximation par le schema
18
19 clf; subplot(2,1,1); plot(t,y,'ro'); plot(t,solution_exacte(t),'bo');
20 legend(["Schema Point milieu";"Solution exacte"],opt=4);
21 title("Application du schema du point milieu",'fontsize',5);
22 //on zoom sur certains points pour juger visuellement l'approximation
23 subplot(2,1,2);
24 plot(t(10:16),y(10:16),'ro'); plot(t(10:16),solution_exacte(t(10:16)), 'bo');

```

Figure 3.2: Affichage (schéma du point milieu)

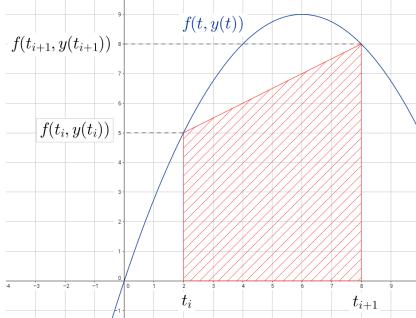
Application du schéma du point milieu



3.5 Schéma d'Euler-Cauchy

3.5.1 Théorie

Définition



On rappelle que l'on a :

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

On approche $\int_{t_i}^{t_{i+1}} f(t, y(t)) dt$ par **l'aire du trapèze** :

$$\begin{aligned} y(t_{i+1}) &\sim y(t_i) + \frac{h}{2} [f(t_i, y(t_i)) + f(t_{i+1}, y(t_{i+1}))] \\ &\sim y(t_i) + \frac{h}{2} [f(t_i, y(t_i)) + f(t_i + h, y(t_i) + hf(t_i, y(t_i)))] \end{aligned}$$

D'où le schéma d'Euler-Cauchy :

$$\left\{ \begin{array}{l} z_{i+1} = z_i + \frac{h}{2} [f(t_i, z_i) + f(t_i + h, z_i + hf(t_i, z_i))] \\ z_0 \text{ donné, } z_0 = y_0 \end{array} \right. \quad (3.8)$$

Propriétés

Le schéma d'Euler-Cauchy est **stable**, **consistant** et **d'ordre 2**.
(si f est L-lipschitzienne par rapport à la 2ème variable)

Démonstration (stabilité) :

On pose $\Phi(t, y, h) = \frac{f(t, y) + f(t+h, y+hf(t, y))}{2}$, ainsi :

$$\begin{aligned} \|\Phi(t, y, h) - \Phi(t, z, h)\| &= \frac{1}{2} \|f(t, y) - f(t, z) + f(t+h, y+hf(t, y)) - f(t+h, z+hf(t, z))\| \\ &\leq \frac{1}{2} [L|y-z| + L|y-z| + h(f(t, y) - f(t, z))] \\ &\leq \frac{1}{2}|y-z| [L + L + hL^2] \\ &\leq C|y-z| \end{aligned}$$

Ce qui montre que le schéma d'Euler-Cauchy est stable.

Démonstration (consistance) :

On a $\Phi(t, y, 0) = \frac{f(t, y) + f(t, y)}{2} = f(t, y)$

Ce qui montre que le schéma d'Euler-Cauchy est consistant.

Démonstration (ordre) :

$$\begin{aligned} T_i &= \frac{y(t_{i+1}) - y(t_i)}{h} - \Phi(t, y(t_i), h) \\ &= \frac{y(t_{i+1}) - y(t_i)}{h} - \frac{1}{2}(f(t_i, y(t_i)) + f(t_i + h, y(t_i) + hf(t_i, y(t_i)))) \\ &= f(t_i, y(t_i)) + \frac{h}{2} \left[\frac{df}{dt}(t_i, y(t_i)) + \frac{df}{dy}(t_i, y(t_i)) \right] + \frac{h^2}{2} y^{(3)}(\xi_i) \\ &\quad - \frac{1}{2}(f(t_i, y(t_i)) + f(t_i, y(t_i))) + h \frac{df}{dt}(t_i, y(t_i)) + h \frac{df}{dy}(t_i, y(t_i)) + O(h^2) \\ &= \frac{h^2}{2} y^{(3)}(\xi_i) + O(h^2) \end{aligned}$$

Donc $T_i \leq Ch^{(2)}$

Ce qui montre que le schéma d'Euler-Cauchy est d'ordre 2.

3.5.2 Application dans *Scilab*

On s'intéresse toujours à l'équation différentielle (3.6), à savoir :

$$\begin{cases} y' = -ty + t & t \in [0, 4] \\ y(0) = 0 \end{cases} \quad \text{solution : } y = Ce^{\frac{-t^2}{2}} + 1$$

On implémente donc dans *Scilab* le schéma d'Euler-Cauchy avec le code 3.3

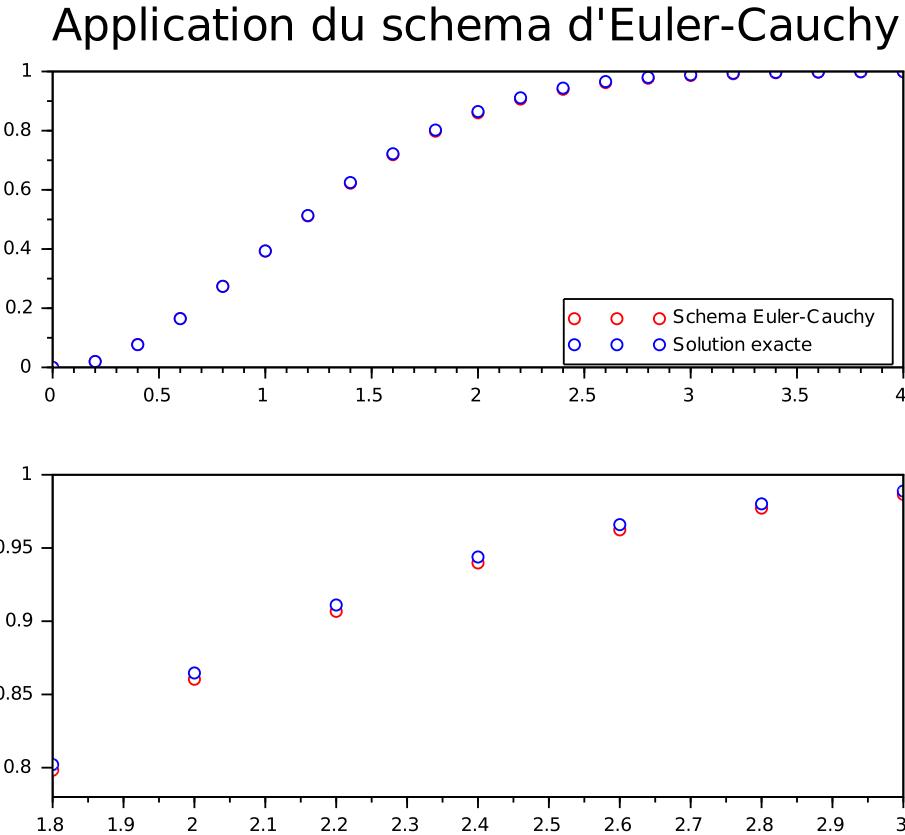
Code 3.3: Schéma d'Euler-Cauchy

```

1 function dydt=f1(t,y)
2   dydt = -t*y+t;
3 endfunction
4
5 function y=euler_cauchy(y0,t,f)
6   n=length(t);
7   y=zeros(1,n);
8   h=t(2)-t(1);
9   y(1)=y0;
10  for i=1:n-1
11    k1=f(t(i),y(i));
12    k2=f(t(i)+h,y(i)+h*k1);
13    y(i+1)=y(i)+h/2*(k1+k2);
14  end
15 endfunction
16
17 function y = solution_exacte(t)
18 y = 1-exp(-t^2/2);
19 endfunction
20
21 T=4; N=20; //intervalle et nombre de points
22 h=T/N; //pas
23 t=[0:h:T]; //subdivision
24 y0=0;
25 y=euler_cauchy(0,t,f1); //approximation par le schema
26
27 clf;
28 subplot(2,1,1);
29 plot(t,y,'ro');
30 plot(t,solution_exacte(t),'bo');
31 legend(["Schema Euler-Cauchy";"Solution exacte"],opt=4);
32 title("Application du schema d'Euler-Cauchy",'fontsize',5);
33 //on zoom sur certains points pour juger visuellement l'approximation
34 subplot(2,1,2);
35 plot(t(10:16),y(10:16),'ro');
36 plot(t(10:16),solution_exacte(t(10:16)),'bo');
```

L'exécution du code 3.3 nous donne la figure (figure 3.3).

Figure 3.3: Affichage (schéma d'Euler-Cauchy)



3.6 Schéma de Runge-Kutta

3.6.1 Théorie

On s'intéresse à un dernier schéma : le **schéma de Runge-Kutta** :

$$\begin{cases} k_1 = f(t_i, z_i) \\ k_2 = f\left(t_i + \frac{h}{2}, z_i + \frac{h}{2}k_1\right) \\ k_3 = f\left(t_i + \frac{h}{2}, z_i + \frac{h}{2}k_2\right) \\ k_4 = f(t_i + h, z_i + hk_3) \\ z_{i+1} = z_i \left(\frac{k_1 + 2k_2 + 2k_3 + k_4}{6} \right) \end{cases} \quad (3.9)$$

Propriétés

Le schéma de Runge-Kutta est **stable**, **consistant** et **d'ordre 4**.
(si f est L-lipschitzienne par rapport à la 2^{ème} variable)
(on admettra les démonstrations de ces propriétés)

3.6.2 Application dans Scilab

On s'intéresse une dernière fois à l'équation différentielle (3.6), à savoir :

$$\begin{cases} y' = -ty + t & t \in [0, 4] \\ y(0) = 0 \end{cases} \quad \text{solution : } y = Ce^{-\frac{t^2}{2}} + 1$$

On implémente donc dans *Scilab* le schéma de Runge-Kutta avec le code 3.4

Code 3.4: Schéma de Runge-Kutta

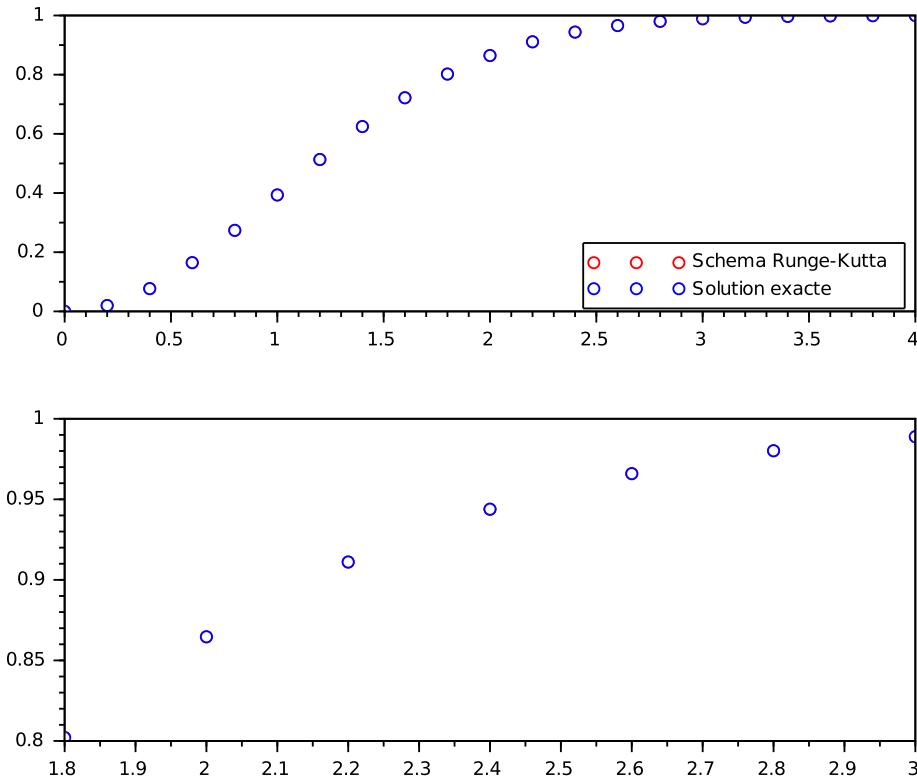
```

1 function dydt=f1(t,y)
2     dydt = -t*y+t;
3 endfunction
4
5 function y=runge_kutta(y0,t,f)
6     n=length(t);
7     y=zeros(1,n);
8     h=t(2)-t(1);
9     y(1)=y0;
10    for i=1:n-1
11        k1=f(t(i),y(i));
12        k2=f(t(i)+h/2,y(i)+h/2*k1);
13        k3=f(t(i)+h/2,y(i)+h/2*k2);
14        k4=f(t(i)+h,y(i)+h*k3);
15        y(i+1)=y(i)+h/6*(k1+2*k2+2*k3+k4);
16    end
17 endfunction
18
19 function y = solution_exacte(t)
20 y = 1-exp(-t^2/2);
21 endfunction
22
23 T=4; N=20; //intervalle et nombre de points
24 h=T/N; //pas
25 t=[0:h:T]; //subdivision
26 y0=0;
27 y=runge_kutta(0,t,f1); //approximation par le schema
28
29 clf;
30 subplot(2,1,1);
31 plot(t,y,'ro');
32 plot(t,solution_exacte(t),'bo');
33 legend(["Schema Runge-Kutta";"Solution exacte"],opt=4);
34 title("Application du schema de Runge Kutta",'fontsize',5);
35 //on zoom sur certains points pour juger visuellement l'approximation
36 subplot(2,1,2);
37 plot(t(10:16),y(10:16),'ro');
38 plot(t(10:16),solution_exacte(t(10:16)), 'bo');
```

L'exécution du code 3.4 nous donne l'affichage (figure 3.4).

Figure 3.4: Affichage (schéma de Runge Kutta)

Application du schéma de Runge Kutta



3.7 Application : mise en évidence de l'ordre de chaque schéma

Rappelons la définition de l'ordre d'un schéma :

Le schéma est d'ordre p si $\exists C > 0$, $\max_{1 \leq i \leq N} |y(t_i) - z_i| \leq Ch^p$

Notons $e_i = |y(t_i) - z_i|$, on a $\max_{1 \leq i \leq N} e_i \sim Ch^p \Leftrightarrow \ln(\max_{1 \leq i \leq N} e_i) \sim \ln(Ch^p) = \ln(C) + p \ln(h)$.

On peut donc tracer $\ln(\max_{1 \leq i \leq N} e_i)$ en fonction de $\ln(h)$, le coefficient de la droite obtenu, après régression linéaire, sera donc p , l'ordre du schéma.

Rappelons les différentes valeurs d'ordre que nous avons défini théoriquement :

Schéma d'Euler : d'ordre 1

Schéma du point milieu : d'ordre 2

Schéma d'Euler-Cauchy : d'ordre 2

Schéma de Runge-Kutta : d'ordre 4

On implémente donc dans *Scilab* le code 3.5.

Code 3.5: Ordre des schémas

```

1  function dydt=f1( t ,y )
2      dydt = -t*y+t ;
3  endfunction
4
5  function y=euler( y0 ,t ,f )
6      n=length( t );
7      y=zeros( 1 ,n );
8      h=t(2)-t(1);
9      y(1)=y0;
10     for i=1:n-1
11         y(i+1)=y(i)+h*f( t(i) ,y(i));
12     end
13 endfunction
14
15 function y=euler_cauchy( y0 ,t ,f )
16     n=length( t );
17     y=zeros( 1 ,n );
18     h=t(2)-t(1);
19     y(1)=y0;
20     for i=1:n-1
21         k1=f( t(i) ,y(i));
22         k2=f( t(i)+h,y(i)+h*k1 );
23         y(i+1)=y(i)+h/2*(k1+k2);
24     end
25 endfunction
26
27 function y=point_milieu( y0 ,t ,f )
28     n=length( t );
29     y=zeros( 1 ,n );
30     h=t(2)-t(1);
31     y(1)=y0;
32     for i=1:n-1
33         k1=f( t(i) ,y(i));
34         k2=f( t(i)+h/2,y(i)+h/2*k1 );
35         y(i+1)=y(i)+h*k2;
36     end
37 endfunction
38
39 function y=runge_kutta( y0 ,t ,f )
40     n=length( t );
41     y=zeros( 1 ,n );
42     h=t(2)-t(1);
43     y(1)=y0;
44     for i=1:n-1
45         k1=f( t(i) ,y(i));
46         k2=f( t(i)+h/2,y(i)+h/2*k1 );
47         k3=f( t(i)+h/2,y(i)+h/2*k2 );
48         k4=f( t(i)+h,y(i)+h*k3 );
49         y(i+1)=y(i)+h/6*(k1+2*k2+2*k3+k4 );
50     end
51 endfunction
52
53 function y = solution_exacte( t )
54 y = 1-exp(-t^2/2);
55 endfunction
56
57 function return=calcul_erreur(y,t)
58 y = abs(y-solution_exacte(t));
59 return = max(y);
60 endfunction

```

```

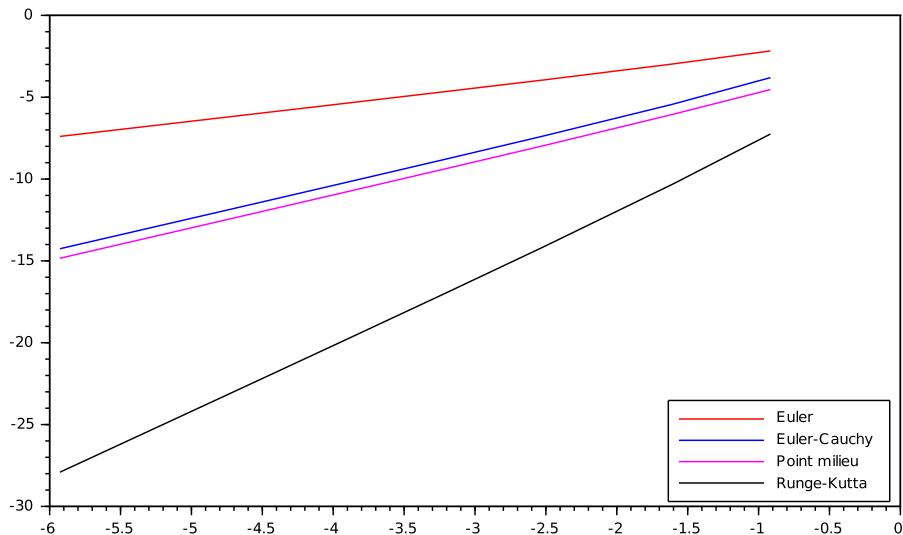
1 T=4;
2 N=[10,20,50,100,200,350,500,750,950,1500]; //variation du nombre de points
3
4 //initialisation
5 pas=zeros(10,1);
6 erreur_euler=zeros(10,1);
7 erreur_euler_cauchy=zeros(10,1);
8 erreur_point_milieu=zeros(10,1);
9 erreur_runge_kutta=zeros(10,1);
10
11 for j=1:10
12     h=T/N(j); //pas
13     pas(j)=h;
14     t=[0:h:T];
15     n=length(t);
16     y0=0;
17     //Euler
18     y=euler(y0,t,f1);
19     e=calcul_erreur(y,t);
20     erreur_euler(j)=e;
21     //Euler Cauchy
22     y=euler_cauchy(y0,t,f1);
23     e=calcul_erreur(y,t);
24     erreur_euler_cauchy(j)=e;
25     //point milieu
26     y=point_milieu(y0,t,f1);
27     e=calcul_erreur(y,t);
28     erreur_point_milieu(j)=e;
29     //Runge Kutta
30     y=runge_kutta(y0,t,f1);
31     e=calcul_erreur(y,t);
32     erreur_runge_kutta(j)=e;
33 end
34
35 //trace
36 clf;
37 plot(log(pas)',log(erreur_euler)','red');
38 plot(log(pas)',log(erreur_euler_cauchy)','blue');
39 plot(log(pas)',log(erreur_point_milieu)','magenta');
40 plot(log(pas)',log(erreur_runge_kutta)','black');
41 legend(["Euler";"Euler-Cauchy";"Point milieu";"Runge-Kutta"],opt=4);
42 title("Regression linéaire des différents schémas",'fontsize',5);
43
44 //calcul des coeff de regression linéaire
45 a1= reglin(log(pas)',log(erreur_euler)');
46 a2= reglin(log(pas)',log(erreur_euler_cauchy)');
47 a3= reglin(log(pas)',log(erreur_point_milieu)');
48 a4= reglin(log(pas)',log(erreur_runge_kutta)');

```

L'exécution du code 3.5 nous donne l'affichage (figure 3.5).

Figure 3.5: Affichage (ordres des différents schémas)

Regression linéaire des différents schémas



Et on a les coefficients de régression linéaire :

```
-->a1
= 1.0342713

-->a2
= 2.0673301

-->a3
= 2.0467572

-->a4
= 4.1013366
```

On retrouve bien une valeur approchée de chacun des ordres théoriques démontrés précédemment.

3.8 Fonction *ode*

Avant de commencer l'étude d'exemples concrets de système d'équation différentiel non soluble analytiquement, il apparaît judicieux de présenter la macro *ode* de *Scilab*. Comme pour la résolution de problème non linéaire (avec *fsolve*), *Scilab* possède déjà une méthode de résolution des systèmes d'équation différentielle : la macro *ode*.

Pour un système mis sous la forme :

$$\begin{cases} y'(t) = f(t, y) \\ y_0 \text{ donné} \end{cases} \quad (3.10)$$

ode a différents prototypes, les arguments qui nous intéressent ici sont :

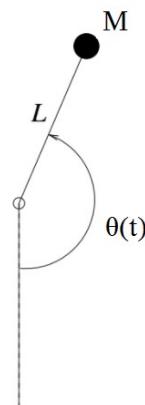
- le y_0 donné
- le t_0 donné
- le vecteur t
- la fonction f

Ainsi, la résolution d'un système d'équation différentielle avec *Scilab* peut se ramener à la mise en forme du problème sous la forme 3.10 puis à l'utilisation de la macro *ode*.

3.9 Application concrète : le pendule

Nous allons nous intéresser à une première application concrète : le problème du pendule. Le pendule que nous allons considérer est représenté sur la figure 3.6. On suppose que la tige reliant le poids de masse M à l'axe de rotation est de masse négligeable devant M . On s'intéresse à la déviation du pendule de la position verticale d'équilibre stable par l'angle $\theta(t)$ mesuré positivement comme défini sur la figure 3.6.

Figure 3.6: Schéma du pendule considéré



Après application des relations de la dynamique pour les solides en rotation autour d'un axe, on obtient le système d'équation différentielle suivant :

$$\begin{cases} \theta''(t) = -\frac{g}{L} \sin \theta(t) \\ \theta(0) = \theta_0 \\ \theta'(0) = 0 \end{cases} \quad (3.11)$$

avec $\theta(0)$ la déviation initiale du pendule, et en considérant que la vitesse angulaire initiale est nulle.

On ne connaît pas de solution analytique au système 3.11. Par conséquent, il est intéressant d'appliquer un des schémas que nous avons précédemment étudier (on appliquera le schéma d'Euler par souci de simplicité et de clarté du code).

Pour compléter l'étude, on peut se souvenir de ce qui est enseigné en terminale : le problème du pendule posé, on fait l'hypothèse suivante : *si $\theta(t)$ est faible, sa mesure en radian est très peu différente de celle de $\sin\theta(t)$* . Ainsi, en se contentant d'une solution approchée, on obtient à partir du système 3.11 le système 3.12.

$$\begin{cases} \phi''(t) = -\frac{g}{L}\phi(t) \\ \phi(0) = \theta_0 \\ \phi'(0) = 0 \end{cases} \quad (3.12)$$

Ainsi, la solution de cette équation différentielle est :

$$\phi(t) = \theta_0 \cos\left(\sqrt{\frac{g}{L}}t\right) \quad (3.13)$$

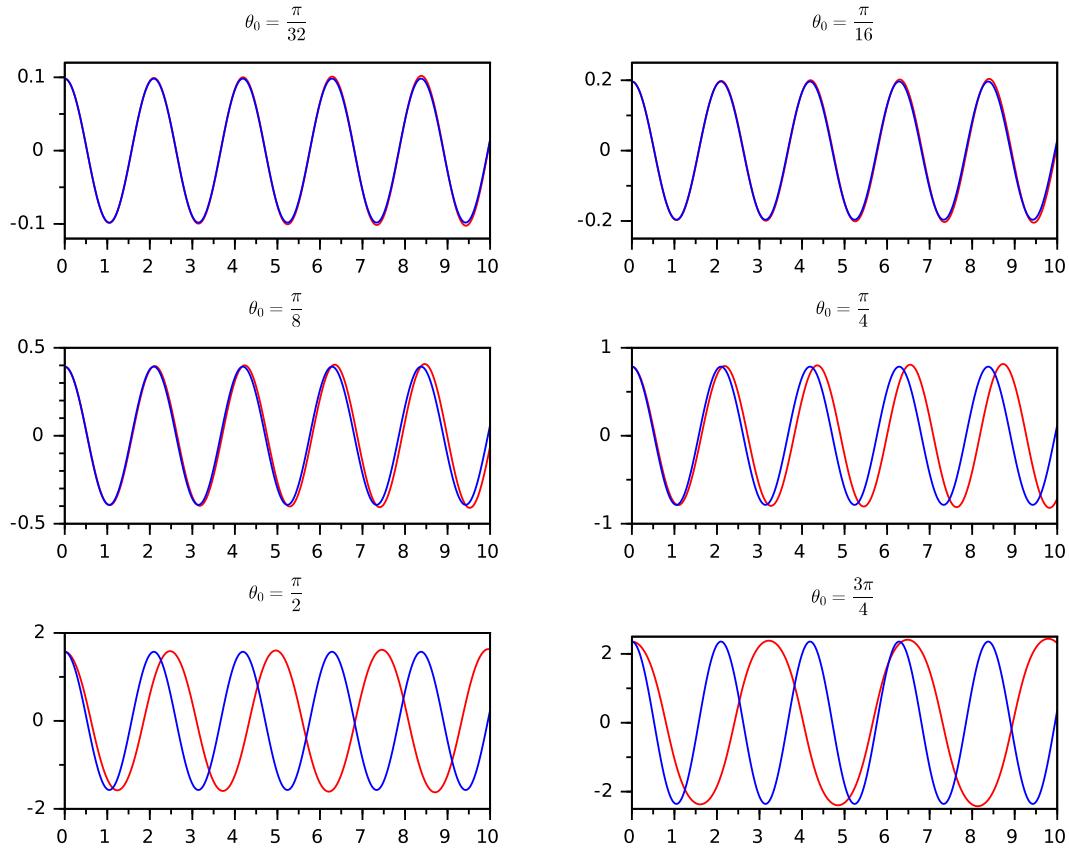
On implémente ainsi dans *Scilab* un programme qui permettra de comparer les deux solutions approchées (par le schéma d'Euler et par l'approximation $\theta(t) \simeq \sin\theta(t)$) avec le code 3.6, on obtient l'affichage (3.7).

Code 3.6: Résolution du problème du pendule

```

1 function dydt = f(t,y)
2     dydt = [y(2); -g/L*sin(y(1))]
3 endfunction
4
5 function y=euler(theta0,t)
6     n=length(t);
7     h=t(2)-t(1);
8     y(1,1)=theta0;
9     y(2,1)=0;
10    for i=1:n-1
11        y(1,i+1)=y(1,i)+h*y(2,i);
12        y(2,i+1)=y(2,i)+h*(-g/L*sin(y(1,i)));
13    end
14 endfunction
15
16 function y = solution_approx(theta0,t)
17     y = theta0*cos(sqrt(g/L)*t);
18 endfunction
19
20 g=9,81; L=1; //donnees
21 t=linspace(0,10,10000);
22 gammeTheta = [%pi/32,%pi/16,%pi/8,%pi/4,%pi/2,3*%pi/4];
23 nomTheta = ["\frac{\pi}{32}", "\frac{\pi}{16}", "\frac{\pi}{8}", "\frac{\pi}{4}", "\frac{\pi}{2}", "3*\frac{\pi}{4}"];
24 "\frac{\pi}{4}", "\frac{\pi}{2}", "\frac{3\pi}{4}"];
25
26 clf;
27 for i=1:length(gammeTheta)
28     theta0=gammeTheta(i);
29     y=euler(theta0,t);
30     subplot(3,2,i);
31     plot(t,y(1,:),'r');
32     plot(t,solution_approx(theta0,t),'b');
33     title(sprintf('$$\theta_0='+nomTheta(i)+ '$$'));
34 end

```

Figure 3.7: Superposition des solutions approchées pour différentes valeurs de θ_0 

Comme on pouvait s'y attendre, l'approximation $\theta(t) \simeq \sin\theta(0)$ est de moins en moins fidèle à la réalité à mesure que $\theta(0)$ augmente. Compte-tenu des graphiques obtenus figure (3.7), on peut raisonnablement considérer que cette approximation est bonne lorsque $\theta(0) \leq \frac{\pi}{4}$.

3.10 Application concrète : les systèmes proies / prédateurs

Nous allons nous intéresser à un deuxième exemple de systèmes d'équations différentielles classiques : les modèles proies / prédateurs. Considérons une population $x(t)$ de proies à l'instant t et une population $y(t)$ de prédateurs à l'instant t , populations évoluant dans le même milieu. On a alors le système d'équations différentielles (3.14).

$$\begin{cases} x'(t) = ax(t) - bx(t)y(t) \\ y'(t) = -cy(t) + dx(t)y(t) \end{cases} \quad \text{avec } a, b, c, d \in \mathbb{R}^+ \quad (3.14)$$

Avec a le taux de reproduction des proies (en l'absence de prédateurs), b le taux de mortalité des proies (en la présence de prédateurs), c le taux de mortalité des prédateurs (en l'absence de proies) et d le taux de reproduction des prédateurs (en la présence de proies). Ces coefficients peuvent être choisis arbitrairement afin de créer différents modèles (où les prédateurs sont avantageés, où les proies sont avantageées, etc). Nous choisissons des valeurs déjà connues comme étant de bons paramètres, à savoir :

$$a = \frac{2}{3}, \quad b = \frac{4}{3}, \quad c = d = 1 \quad (3.15)$$

On obtient donc le modèle proies / prédateurs (3.16).

$$\begin{cases} x'(t) = \frac{2}{3}x(t) - \frac{4}{3}x(t)y(t) \\ y'(t) = -y(t) + x(t)y(t) \end{cases} \quad (3.16)$$

Nous appliquons donc un des schémas étudiés précédemment (par souci de simplicité et de clarté du code, on choisit le schéma d'Euler) au système (3.16) afin de connaître l'évolution de la proportion de proies et de prédateurs en fonction du temps. On s'intéressera également à l'évolution de la proportion de proies en fonction de la proportion de prédateurs. Les conditions initiales sont choisies arbitrairement : $x(0) = y(0) = 1/2$ (notons que la modification de ces conditions initiales ne changent pas l'allure des courbes obtenues). On implémente ainsi dans *Scilab* le code 3.7.

Code 3.7: Résolution du problème proies / prédateurs

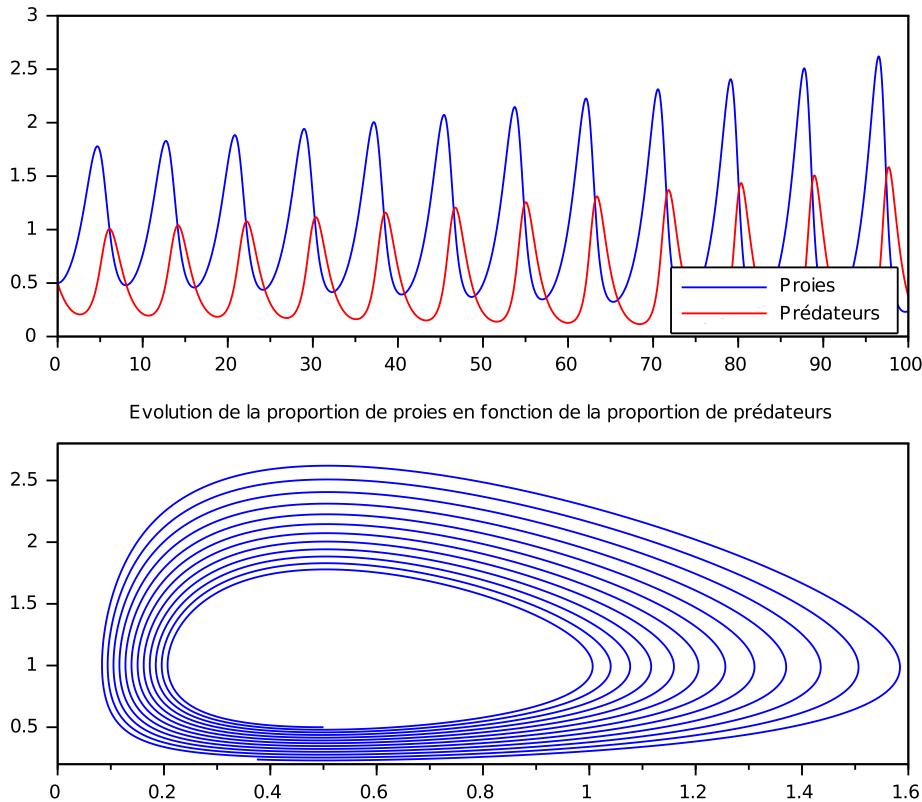
```

1 function dxdt=f(x,y)
2     dxdt = 2/3*x -4/3*x*y;
3 endfunction
4
5 function dydt=g(x,y)
6     dydt=y + x*y;
7 endfunction
8
9 function [x,y]=euler(x0,y0,t,f1,f2)
10    n=length(t);
11    h=t(2)-t(1);
12    x(1)=x0;
13    y(1)=y0;
14    for i=1:n-1
15        x(i+1)=x(i)+h*f1(x(i),y(i)); //proie
16        y(i+1)=y(i)+h*f2(x(i),y(i)); //predateur
17    end
18 endfunction
19
20 T=100; //intervalle
21 N=5000; //nombre de points
22 h=T/N; //pas
23 t=[0:h:T]; //subdivision
24
25 x0=1/2; //condition initiale proie
26 y0=1/2; //condition initiale predateur
27 [x,y]=euler(x0,y0,t,f,g); //approximation par le schema
28
29 clf;
30 subplot(2,1,1);
31 plot(t,x,'b');
32 plot(t,y,'r');
33 legend(["Proies";"Predateurs"],opt=4);
34 title("Evolution des proportions de proies et de predateurs
            en fonction du temps");
35 subplot(2,1,2);
36 plot(y,x,'b');
37 title("Evolution de la proportion de proies en fonction
            de la proportion de predateurs");
38

```

On obtient alors l'affichage (3.8). L'évolution de la proportion de proies et de la proportion de prédateurs indiquent une certaine périodicité dans l'évolution, mise en évidence par l'évolution de la proportion de proies en fonction de la proportion de prédateurs : il s'agit d'un cycle. Tout ceci indique l'existence d'un équilibre dans le milieu. Sans cet équilibre, on peut imaginer que les prédateurs, en présence de proies, se reproduiraient exponentiellement, en dépit des proies qui finiraient par s'éteindre, ce qui entraînerait l'extinction des prédateurs, privés de nourriture. Il n'en est rien, la nature est tout de même bien faite !

Figure 3.8: Évolution des proportions proies-prédateurs
Evolution des proportions de proies et de prédateurs en fonction du temps



3.11 Application concrète : la mécanique céleste

On s'intéresse à un dernier exemple de systèmes d'équations différentielles : la mécanique céleste. L'idée est ici de simuler le mouvements de plusieurs corps soumis à leurs attractions gravitationnelles mutuelles. On considère donc deux corps sphériques de masses respectives m_1 et m_2 et de centres :

$$u_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \quad u_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$

On sait que la force de gravitation exercée par un corps (1) sur un autre corps (2) est égale à :

$$F_{21} = G \frac{m_1 m_2}{\|u_2 - u_1\|^3} (u_2 - u_1)$$

Où G est la constante de gravitation universelle, $G = 6,67 \times 10^{-11}$.

En écrivant les équations de la dynamique, on peut simuler le mouvement des deux corps en fonction d'une configuration initiale de position et de vitesse, on obtient alors le système d'équations différentielles (3.17).

$$\begin{cases} u_1'' = +Gm_2 \frac{u_2 - u_1}{\|u_2 - u_1\|^3} \\ u_2'' = -Gm_1 \frac{u_2 - u_1}{\|u_2 - u_1\|^3} \end{cases} \quad (3.17)$$

On a les conditions initiales de position :

$$u_1(0) = (0, 0), \quad u_2(0) = (d_{TL}, 0) \quad (3.18)$$

Et de vitesse :

$$u_1'(0) = (0, 0), \quad u_2'(0) = (0, \frac{2\pi}{T} d_{TL}) \quad (3.19)$$

Où d_{TL} est la distance Terre-Lune, $d_{TL} = 3,84402 \times 10^8$ mètres et T est la période de rotation, $T = 27,55$ jours.

Pour pouvoir travailler directement avec la macro *ode* de *Scilab*, on met les équations (3.17) sous forme du premier ordre en temps. On considère $v \in \mathbb{R}^8$ tel que

$$v = \begin{pmatrix} u_1 \\ u'_1 \\ u_2 \\ u'_2 \end{pmatrix} \quad (3.20)$$

On peut donc écrire le système (3.17) sous la forme $v' = f(v)$. On implémente ainsi dans *Scilab* le code 3.8.

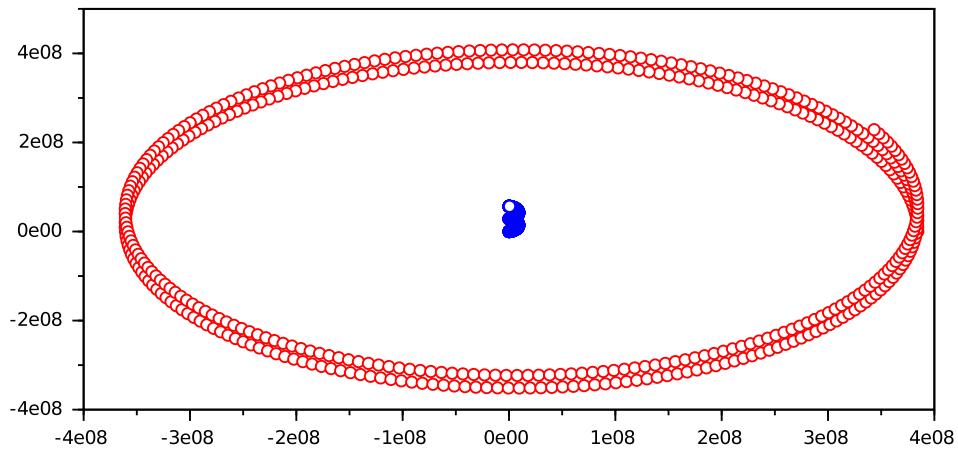
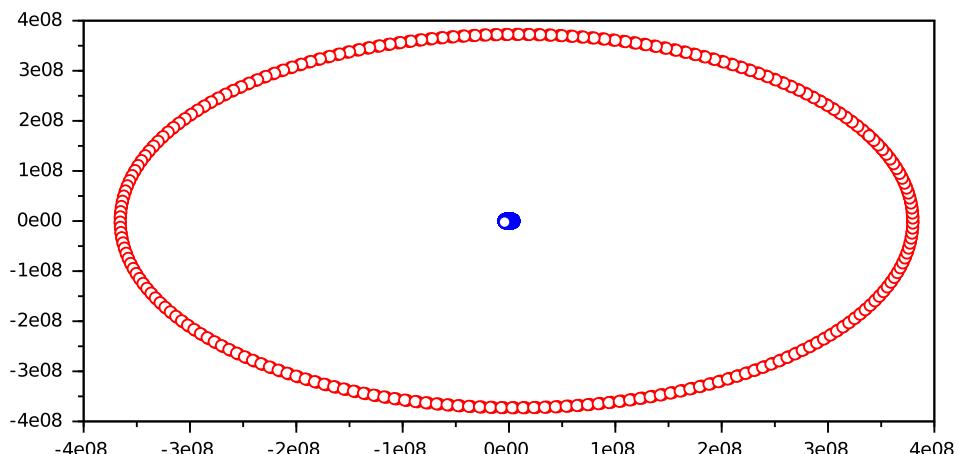
Code 3.8: Résolution du modèle de mécanique céleste

```

1 function dvdt=celest(t,v)
2   u1=[v(1),v(2)]';
3   u1_prime=[v(3),v(4)]';
4   u2=[v(5),v(6)]';
5   u2_prime=[v(7),v(8)]';
6   dvdt=[u1_prime,G*m2*(u2-u1)/(norm(u2-u1))^3,
7         u2_prime,-G*m1*(u2-u1)/(norm(u2-u1))^3];
8 endfunction
9
10 // constantes : donnees de l'exercice
11 m1=5.975e24;
12 m2=7.35e22;
13 G=6.67e-11;
14 d=3.84402e8;
15 T=27.55*24*60*60;
16
17 t=[0:10000:2*T];
18 u1_zero=[0,0]';
19 u1_prime_zero=[0,0]';
20 u2_zero=[d,0]';
21 u2_prime_zero=[0,2*d*pi/T]';
22 v0=[u1_zero;u1_prime_zero;u2_zero;u2_prime_zero];
23 v=ode(v0,0,t,celest);
24
25 clf;
26 //trace des trajectoires (affichage 1)
27 subplot(2,1,1);
28 plot(v(1,:),v(2,:),'bo',v(5,:),v(6,:),'ro')
29 title("Trajectoires des deux corps",'fontsize',4);
30
31 //code animation dynamique trajectoires
32 //comet(v(1,:),v(2,:), "colors",color("blue"));
33 //comet(v(5,:),v(6,:), "colors",color("red"));
34
35 // coordonnees du centre de gravite
36 Gx=((m1*v(1,:)+m2*v(5,:))/(m1+m2));
37 Gy=((m1*v(2,:)+m2*v(6,:))/(m1+m2));
38
39 //trace des trajectoires (affichage 2)
40 subplot(2,1,2);
41 plot(v(1,:)-Gx,v(2,:)-Gy,'bo',v(5,:)-Gx,v(6,:)-Gy,'ro')
42 title("Trajectoires des deux corps (centres sur leur centre de gravite)",
      'fontsize',4);
43
```

On obtient alors l'affichage (3.9).

Figure 3.9: Mécanique céleste : trajectoires des corps

Trajectoires des deux corps**Trajectoires des deux corps (centrées sur leur centre de gravité)**

CHAPITRE 4

VALEURS PROPRES

4.1 Introduction

4.2 Base théorique

4.2.1 Rappels d'algèbre linéaire

On a le théorème (4.1) :

Théorème :

$$\begin{aligned} A \in \mathcal{M}_{n,n}(\mathbb{R}) \text{ est une matrice symétrique} \\ \Leftrightarrow \exists P \text{ orthogonale et } D \text{ diagonale } \in \mathcal{M}_{n,n} \text{ telles que } A = PDP^T \\ \text{avec } D = \begin{pmatrix} \lambda_1 & & 0 \\ 0 & \ddots & 0 \\ 0 & & \lambda_n \end{pmatrix} \text{ et } \lambda_1, \dots, \lambda_n \text{ les valeurs propres de } A \end{aligned} \quad (4.1)$$

Ce théorème, très pratique, ne peut toutefois s'appliquer que sur des matrices carrées et symétriques. Dès lors, que faire lorsqu'on est confronté à des matrices qui ne respectent pas ces conditions ?

4.2.2 La décomposition SVD (*Singular Value Decomposition*)

Considérons $A \in \mathcal{M}_{m,n}(\mathbb{R})$ avec $m \geq n$ (la matrice possède plus de lignes que de colonnes). On a alors le théorème (4.2) :

Théorème :

$$\begin{aligned} \text{Pour toute matrice } A \in \mathcal{M}_{m,n}(\mathbb{R}), \\ \exists U \in \mathcal{M}_{m,m}(\mathbb{R}) \text{ et } V \text{ orthogonales } \in \mathcal{M}_{N,n}(\mathbb{R}) \\ \text{et } \Sigma \text{ une matrice de la forme } \Sigma = \begin{pmatrix} \sigma_1 & & 0 \\ 0 & \ddots & 0 \\ 0 & & \sigma_n \\ & & 0 \end{pmatrix} \in \mathcal{M}_{m,n}(\mathbb{R}) \\ \text{telles que } A = U\Sigma V^T \end{aligned} \quad (4.2)$$

Idée de la preuve :

AA^T ($\in \mathcal{M}_{m,m}(\mathbb{R})$) et A^TA ($\in \mathcal{M}_{n,n}(\mathbb{R})$) sont des matrices symétriques.

Donc, d'après le théorème (4.1) elles admettent des décompositions :

$$A^TA = PD_1P^T \text{ et } AA^T = QD_2Q^T$$

$$\text{Supposons le théorème : } A^TA = V\Sigma^T \underbrace{U^TU}_I \Sigma V^T = V\Sigma^T \Sigma V^T$$

Par identification, σ_i^2 sont les valeurs propres de A^TA et V est la matrice de passage orthogonale associée.

De même, U est la matrice de passage orthogonale associée à AA^T .

$$\sigma_i^2 \text{ valeur propre de } A^TA \implies \sigma_i = \pm \sqrt{\text{valeur propre de } A^TA}$$

Si on veut que Σ soit unique, il suffit donc d'imposer $\sigma_i \geq 0$.

$$\text{Ainsi } A = U\Sigma V^T \text{ avec } \Sigma = \begin{pmatrix} \sigma_1 & & 0 \\ 0 & \ddots & 0 \\ 0 & & \sigma_n \\ & & 0 \end{pmatrix} \text{ et } \sigma_i \geq 0 \ \forall i = 1, \dots, n$$

4.2.3 Rang et noyau de $A \in \mathcal{M}_{m,n}(\mathbb{R})$

$$A = U\Sigma V^T \Rightarrow AV = U\Sigma. \text{ Notons } V = [V_1 \cdots V_n] \text{ et } U = [U_1 \cdots U_m]$$

$$\text{On a donc } A[V_1 \cdots V_n] = [U_1 \cdots U_m] \begin{pmatrix} \sigma_1 & & 0 \\ 0 & \ddots & 0 \\ 0 & & \sigma_n \\ & & 0 \end{pmatrix} \Rightarrow AV_i = \sigma_i U_i, \forall i = 1, \dots, n$$

Soit r le nombre de valeurs propres singulières non nulles : $\sigma_{r+1} = \dots = \sigma_n = 0$

$$\text{On a donc } \begin{cases} AV_i = \sigma_i U_i & \text{si } 1 \leq i \leq r \\ AV_i = 0 & \text{si } r+1 \leq i \leq n \end{cases}$$

Donc, par définition de $Ker(A)$, $Im(A)$ et la formule du rang :

$$Ker(A) = vect\{V_{r+1}, \dots, V_n\}, Im(A) = vect\{U_1, \dots, U_r\}, rang(A) = r$$

4.3 Application 1 : La compression d'images

4.3.1 Présentation du problème

Une image, constituée de m lignes et n colonnes de pixels peut-être représentée par une matrice $A = (a_{ij}) \in \mathcal{M}_{m,n}(\mathbb{R})$ avec la valeur de a_{ij} représentant le niveau de gris du pixel (i, j) .

Ainsi nous nous sommes intéressée à un cas d'école de la compression d'image : celle de la photo de Lena, une *playmate* prise dans un numéro du magazine *Playboy* (4.1).

On implémente dans *Scilab* le code 4.1 afin de récupérer depuis un fichier .csv la matrice de Lena en niveaux de gris, on obtient l'affichage (4.2).

Figure 4.1: Image originale Lena



Code 4.1: Lena : Récupération et affichage de l'image originale

```

1 n=512;
2 funcprot(0);
3 l=read('~/home/marlow/scilab/TD7/lena.csv',512,512);
4 lena=l';
5 x=[1:512];
6 y=[512:-1:1];
7 xset('colormap',graycolormap(256));
8 grayplot(x,y,lena);
9 set(gca(),'isoview','on');
```

Figure 4.2: Image originale Lena en niveaux de gris (obtenue grâce à la matrice)



La matrice *lena* que nous venons de créer et de remplir possède donc 512 colonnes et 512 lignes, elle occupe une place en mémoire absolument colossale. C'est pourquoi nous nous intéressons à sa compression.

4.3.2 Théorie de résolution (théorème et corollaire)

Pour cela, on a le théorème (4.3) et son corollaire (4.4).

Théorème

$$A \in \mathcal{M}_{m,n}(\mathbb{R}) \text{ alors } A = \sum_{i=1}^r \sigma_i U_i V_i^T \quad (4.3)$$

A est une somme de matrice de rang 1 (car $U_i V_i^T$ est une matrice de rang 1)

Corollaire

$$A_k \sum_{i=1}^k \sigma_i U_i V_i^T \quad (4.4)$$

On a $\|A - A_k\|_F^2 = \sigma_k^2 + 1$

Où $\|A\|_F^2$ est la norme de Frobenius de la matrice A : $A = (a_{ij})$, $\|A\|_F^2 = \sum a_{ij}^2$

Dans le cas des images, nous n'avons pas besoin de toutes les valeurs de σ_i pour construire l'image. En effet, pour un seuil $\epsilon > 0$ donné, on a $|\sigma_m| < \epsilon$ pour $m \geq k+1$, avec $k << 512$. On remplace donc A par A_k pour différentes valeurs de k (on prendra ici 10, 20 et 30). Ainsi on diminue considérablement la place de l'image en mémoire : on conserve les k valeurs de σ_i , et les vecteurs U_i et V_i associés (on conserve donc $3k$ valeurs, contre 512^2 avant compression).

4.3.3 Résolution du problème

On implémente ainsi dans *Scilab* le code 4.2 qui nous donne l'affichage (4.3) et l'affichage (4.4).

Code 4.2: Lena : Compression de l'image

```

1 // recuperation de l'image originale
2 n=512; funcprot(0);
3 l=read('~/home/marlow/scilab/TD7/lena.csv',512,512); lena=l';
4 x=[1:512]; y=[512:-1:1]; xset('colormap',graycolormap(256));
5
6 // resolution du SVD
7 [U,S,V]=svd(lena);
8 sig=diag(S);
9 clf;
10 plot(sig,'o');
11 xlabel("k",'fontsize',4); ylabel("$\sigma_k$",'fontsize',4);
12 eps=0.005; // seuil
13 for i=1:n if sig(i)<eps*sig(1) then break; end end
14 clf;
15
16 // creation de differentes valeurs de k
17 gammek = [10,20,30]; nomk = ["10","20","30"];
18
19 // application pour creer differentes compressions
20 for i=1:length(gammek)
21     k=gammek(i); Ak=sig(1)*U(:,1)*V(:,1)';
22     for j=2:k
23         Ak=Ak+sig(j)*U(:,j)*V(:,j)';
24     end
25     subplot(2,3,i); // affichage de l'image
26     grayplot(x,y,Ak); set(gca(),'isoview','on');
27     title(sprintf('k=%d',nomk(i)));
28     subplot(2,3,i+3); // affichage de l'erreur
29     grayplot(x,y,Ak-lena); set(gca(),'isoview','on');
30     title('Erreur');
31 end

```

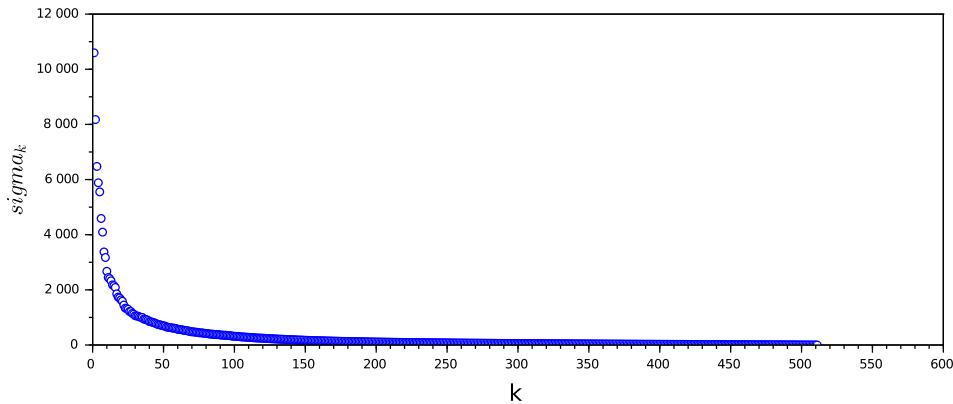
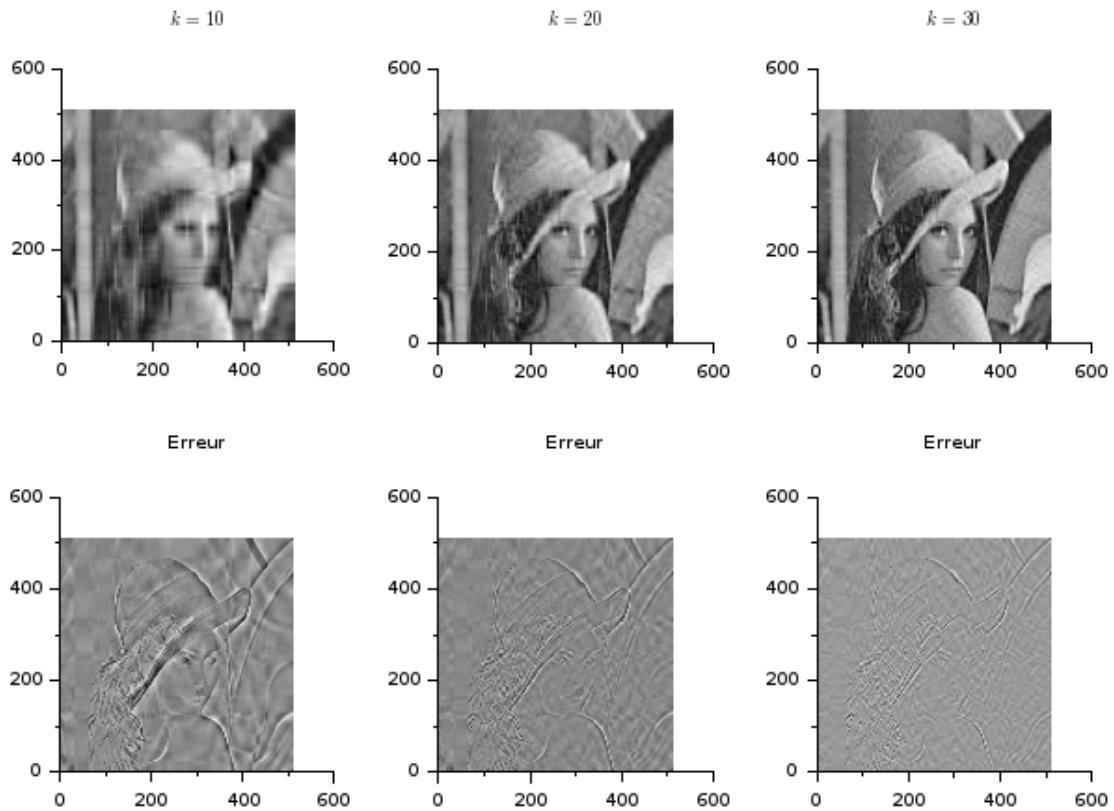
Figure 4.3: Valeurs propres de Σ 

Figure 4.4: Différentes compressions de l'image et erreur associée



Pour s'interroger sur la puissance de la décomposition *SVD*, on peut se poser une question : est il possible, une fois que l'image a été *bruitée*, déformée, floutée, etc, de récupérer une image plus nette, proche de l'image originale ?

C'est tout à fait possible en utilisant une nouvelle fois la décomposition *SVD* : on implémente dans *Scilab* le code 4.3. Le code nous donne l'affichage (4.5) : la décomposition *SVD* permet donc de récupérer une image plus nette à partir d'une image bruitée (même si l'image finale n'est pas aussi nette que l'image originale).

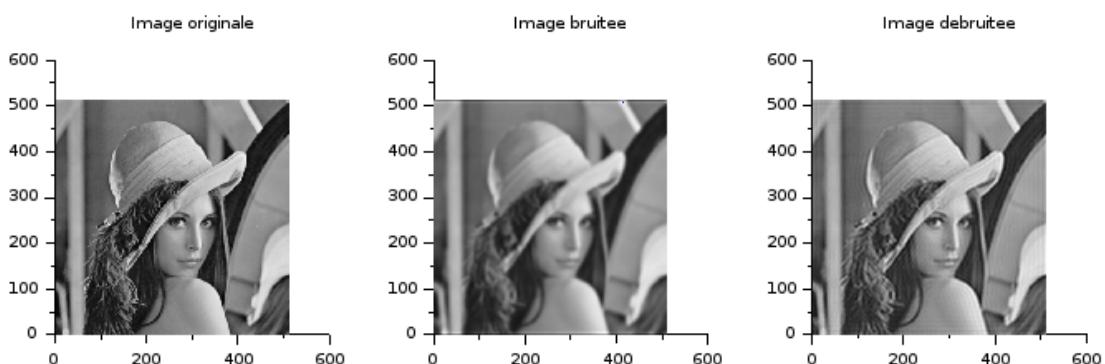
Code 4.3: Lena : Bruitage puis débruitage de l'image

```

1 //recuperation de l'image originale
2 n=512; funcprot(0);
3 l=read('~/home/marlow/scilab/TD7/lena.csv',512,512); lena=l';
4 x=[1:512]; y=[512:-1:1]; xset('colormap',graycolormap(256));
5
6 //resolution du SVD
7 [U,S,V]=svd(lena);
8 sig=diag(S);
9 eps=0.005; //seuil
10 for i=1:n if sig(i)<eps*sig(1) then break; end end
11
12 //signal bruite
13 k=i; T=zeros(n,n);
14 for i=1:n
15   for j=1:n
16     T(i,j)=exp(-1/10*(i-j)^2);
17   end
18   C(i)=sum(T(i,:));
19   T(i,:)=T(i,:)/C(i);
20 end
21 eta=rand(n); eta=eta\|norm(eta); eta=eta*norm(lena)*0.001;
22 v=T*lena*T; w=v+eta; Ab=T*w*T;
23
24 //signal debrouitee
25 [U,ST,V]=svd(T);
26 sigT=diag(ST);
27 Td=zeros(T');
28 for i=1:k
29   Td=Td+sigT(i)*U(:,i)*V(:,i)';
30 end
31 Ar=pinv(Td)*w*pinv(Td);
32
33 //affichage
34 clf;
35 subplot(1,3,1); set(gca(), 'isoview', 'on');
36 grayplot(x,y,lena); title('Image originale');
37 subplot(1,3,2); set(gca(), 'isoview', 'on');
38 grayplot(x,y,Ab); title('Image bruitee');
39 subplot(1,3,3); set(gca(), 'isoview', 'on');
40 grayplot(x,y,Ar); title('Image debrouitee');

```

Figure 4.5: Résultat du débruitage



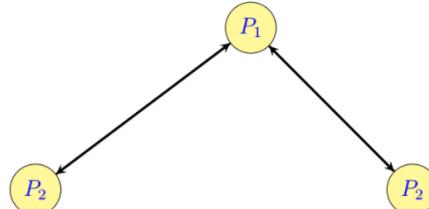
4.4 Application 2 : le *PageRank*

4.4.1 Présentation du problème

Pour détailler et expliquer la méthode du *PageRank*, il apparaît pertinent de s'intéresser à un exemple très simple de configurations de pages internet (figure (4.6)). Cette configuration ne comprend que 3 pages notées P_1 , P_2 et P_3 telles que :

Figure 4.6: PageRank : configuration 1

- la page P_1 a un lien vers P_2 et P_3
- la page P_2 a un lien vers P_1
- la page P_3 a un lien vers P_1



On se demande donc : laquelle des trois pages a le plus d'importance ? Autrement dit, laquelle est la plus pertinente ?

4.4.2 Théorie de résolution (modélisation et méthode de la puissance)

On peut dans un premier temps poser les suites suivantes :

- p_n la probabilité d'être sur P_1 au bout de n clics
- q_n la probabilité d'être sur P_2 au bout de n clics
- r_n la probabilité d'être sur P_3 au bout de n clics

Notons $G = (g_{ij})$ la matrice définie $\begin{cases} g_{ii} = 0 \\ g_{ij} = 1 \text{ si } j \text{ pointe sur } i \end{cases}$

Notons $H = (h_{ij})$ la matrice définie $\begin{cases} h_{ij} = \frac{g_{ij}}{c_j} \text{ avec } c_j = \sum_{i=1}^3 g_{ij} \text{ si } c_j \neq 0 \\ h_{ij} = \frac{1}{3} \text{ sinon} \end{cases}$

Afin d'élaborer un premier modèle dit naïf, on considère que l'internaute clique sur le lien d'une page et s'y dirige selon les probabilités :

$$\begin{pmatrix} p_{n+1} \\ q_{n+1} \\ r_{n+1} \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 & 1 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{pmatrix}}_H \begin{pmatrix} p_n \\ q_n \\ r_n \end{pmatrix} \quad (4.5)$$

Posons $A = \begin{pmatrix} 0 & 1 & 1 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{pmatrix}$, supposons que $n \rightarrow +\infty$, si $\begin{pmatrix} p_n \\ q_n \\ r_n \end{pmatrix} \rightarrow C$ alors on a $C = AC$

Donc C est le vecteur propre de A associée à la valeur propre 1.

Or dans notre exemple, on a $\begin{pmatrix} p_n \\ q_n \\ r_n \end{pmatrix} = A^n \begin{pmatrix} p_0 \\ q_0 \\ r_0 \end{pmatrix} = A^n \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}$ diverge lorsque $n \rightarrow +\infty$

En effet, A a trois valeurs propres distinctes : $\lambda_1 = 1$, $\lambda_2 = -1$ et $\lambda_3 < 1$, donc $|\lambda_1| = |\lambda_2| = 1$. A possède donc deux valeurs propres dominantes, donc il est normal que la suite diverge. Le modèle naïf (4.5) ne permet donc pas de déterminer le PageRank de notre configuration très simple.

Il est donc indispensable de modifier le modèle. Pour cela, nous allons introduire une part de hasard : on suppose que l'internaute peut à tout moment réactualiser sa recherche et accéder à n'importe quelle autre page aléatoirement (de manière équitable).

Soit A_n cet événement, notons $\alpha = P(A_n)$, on a alors, selon la formule des probabilités totales :

- $P(p_{n+1}) = P(p_{n+1}|A_n) \times P(A_n) + P(p_{n+1}|\overline{A_n}) \times P(\overline{A_n}) = \alpha \times \frac{1}{3} + (1-\alpha)(q_n + r_n)$
- $P(q_{n+1}) = P(q_{n+1}|A_n) \times P(A_n) + P(q_{n+1}|\overline{A_n}) \times P(\overline{A_n}) = \alpha \times \frac{1}{3} + (1-\alpha)p_n \times \frac{1}{2}$
- $P(r_{n+1}) = P(r_{n+1}|A_n) \times P(A_n) + P(r_{n+1}|\overline{A_n}) \times P(\overline{A_n}) = \alpha \times \frac{1}{3} + (1-\alpha)p_n \times \frac{1}{2}$

$$\text{d'où } \begin{pmatrix} p_{n+1} \\ q_{n+1} \\ r_{n+1} \end{pmatrix} = \left[\alpha \times \frac{1}{3} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} + (1-\alpha) \begin{pmatrix} 0 & 1 & 1 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{pmatrix} \right] \begin{pmatrix} p_n \\ q_n \\ r_n \end{pmatrix}$$

Par ailleurs, on a $p_n + q_n + r_n = 1$
 Donc $\begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} p_n \\ q_n \\ r_n \end{pmatrix} = ee^T \begin{pmatrix} p_n \\ q_n \\ r_n \end{pmatrix}$ avec $e = (1, 1, 1)^T$

Finalement, avec $p = 1 - \alpha$, on a le modèle :

$$\begin{pmatrix} p_{n+1} \\ q_{n+1} \\ r_{n+1} \end{pmatrix} = \underbrace{\left[p \begin{pmatrix} 0 & 1 & 1 \\ \frac{1}{2} & 0 & 0 \\ \frac{1}{2} & 0 & 0 \end{pmatrix} + (1-p) \times \frac{1}{3} ee^T \right]}_A \begin{pmatrix} p_n \\ q_n \\ r_n \end{pmatrix} \quad (4.6)$$

La matrice A admet donc comme unique valeur propre dominante et comme valeur propre simple $\lambda = 1$. On peut ainsi lui appliquer la méthode de la puissance afin de déterminer x tel que $x = Ax$, x sera alors le PageRank de la configuration.

Méthode de la puissance :

Soit $A \in \mathcal{M}_{n,n}(\mathbb{R})$ telle que A admet $\lambda_1, \lambda_2, \dots, \lambda_n$ valeurs propres

On suppose $|\lambda_1| > |\lambda_2| \geq \dots \geq |\lambda_n|$, on dit que λ_1 est la valeur propre dominante de A .

Le but de la méthode est d'approcher λ_1 et v_1 le vecteur propre associé.

On suppose v_1, \dots, v_n une base de vecteurs propres de A .

On part de $x^{(0)} \in \mathbb{R}^n$ tel que $x^{(0)} = \sum_{i=1}^n \alpha_i v_i$ avec $\alpha_i \neq 0$

On a naïvement $x^{(1)} = Ax^{(0)} = A\left(\sum_{i=1}^n \alpha_i v_i\right) = \sum_{i=1}^n \alpha_i Av_i = \sum_{i=1}^n \alpha_i \lambda_i v_i$

Et donc : $x^{(k)} = Ax^{(k-1)} = A^k x^{(0)} = \sum_{i=1}^n \alpha_i \lambda_i^k v_i = \lambda_1^k \alpha_1 v_1 + \sum_{i=2}^n \alpha_i \lambda_i^k v_i = \lambda_1^k \alpha_1 \left[v_1 + \sum_{i=2}^n \frac{\alpha_i}{\alpha_1} \left(\frac{\lambda_i}{\lambda_1} \right)^k v_i \right]$

Si $k \gg 1$, on a $x^{(k)} \sim \lambda_1^k \alpha_1 v_1$, donc $x^{(k)}$ est dans la direction de v_1

Ainsi, on a la suite définie par récurrence :

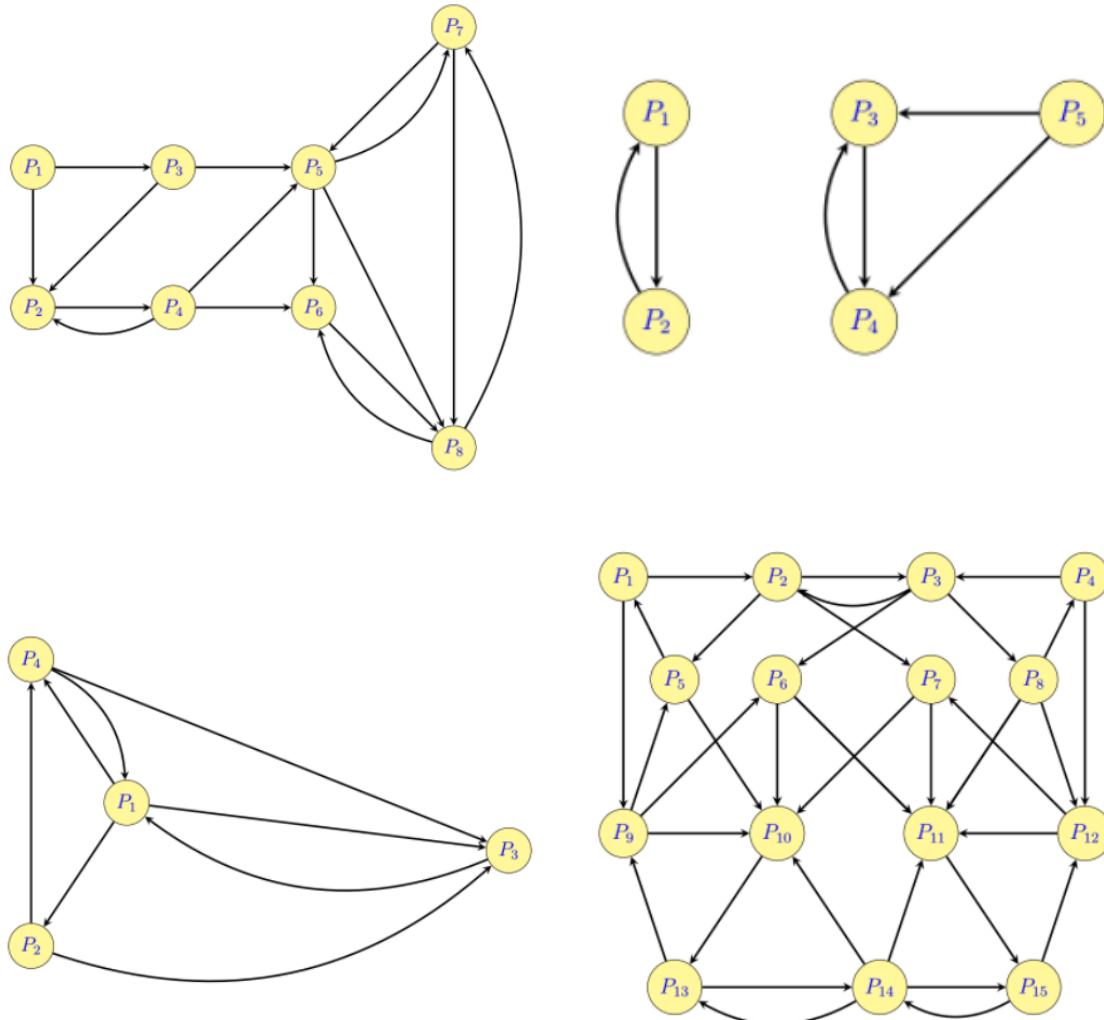
$$\begin{cases} \forall n \in \mathbb{N}, w^{(n+1)} = Aw^{(n)} \\ \text{Avec } \frac{w^{(n)}}{\|w^{(n)}\|} \xrightarrow{\|v_1\|} \frac{v_1}{\|v_1\|} \\ \text{Et } \frac{(w^{(n)})^T Aw^{(n)}}{\|w^{(n)}\|^2} \xrightarrow{\|v_1\|^2} \lambda_1 \end{cases}$$

Dans la pratique, on normalise les vecteurs $w^{(n)}$ afin d'éviter les valeurs trop importantes. L'idée est donc la suivante : on définit (de manière arbitraire) un premier vecteur w et, itérativement, on applique A et on normalise. Ainsi, au bout de n itérations, on a approché le vecteur propre v_1 associé à la valeur propre dominante λ_1 .

Dans le cadre du PageRank, on cherche donc à obtenir le vecteur x tel que $x = Ax$, avec A vérifiant les conditions d'application de la méthode de la puissance. L'idée est donc d'appliquer A à un vecteur x (on partira de conditions d'équiprobabilité) et de normaliser le résultat n fois (on prendra $n = 100$).

La configuration 1 décrite dans la figure (4.6) constitue un exemple très simple. Ainsi, pour étoffer notre étude, on appliquera donc également la méthode aux configurations plus complexes suivantes. On définira leur modèle respectif de manière analogue à celui de la configuration 1 (modèle (4.6)).

Figure 4.7: PageRank : configurations 2 à 5



4.4.3 Résolution du problème

On implémente ainsi dans *Scilab* le code 4.4.

Code 4.4: Détermination du PageRank d'un ensemble de page internet

```

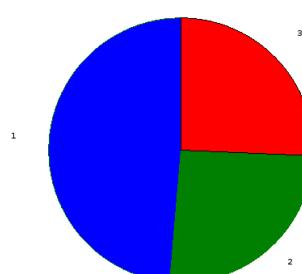
1 function A=constrA(G,n)
2     e=ones(n,1);
3     A=p*G+(1-p)/n*e*e';
4 endfunction
5
6 function H=constrH(G,n)
7     H=zeros(G);
8     for i=1:n
9         for j=1:n
10            s=sum(G(:,j));
11            if s == 0 then
12                G(:,j)=ones(n,1)/n; s=1;
13            end
14            H(i,j)=G(i,j)/s;
15            end
16        end
17 endfunction
18
19 alpha=0.15; p=1-alpha;
20 G=sparse([[1,2;1,3;2,1;3,1;], ones(4,1)); //config 1
21 //config 2 : G=sparse([[2,1;3,1;4,2;2,3;5,3;2,4;5,4;6,4;6,5;7,5;8,5;8,6;5,7;
22 //           //8,7;6,8;7,8;], ones(16,1));
23 //config 3 : G=[sparse([[1,2;2,1;3,4;3,5;4,3;4,5], ones(6,1));0 0 0 0];
24 //config 4 : G=sparse([[1,3;1,4;2,1;3,1;3,2;3,4;4,1;4,2;], ones(8,1))
25 //config 5 : G=sparse([[1,5;2,1;2,3;3,2;3,4;4,8;5,2;5,9;6,3;6,9;7,2;7,12;8,3;
26 //           //9,1;9,13;10,5;10,6;10,7;10,14;11,6;11,7;11,8;11,12;
27 //           //11,14;12,4;12,8;12,15;13,10;13,14;14,13;14,15;15,11;
28 //           //15,14;], ones(33,1));
29
30 n=length(G(1,:)); //nombre de pages
31 H=constrH(G,n);
32 A=constrA(H,n);
33 x=ones(n,1)/n; //equiprobabilite au depart
34 for i=1:100
35     x=A*x; x=x/sum(x);
36 end
37
38 disp(x); pie(x); //permet de tracer un diagramme circulaire

```

On obtient ainsi les vecteurs associés aux différentes configurations :

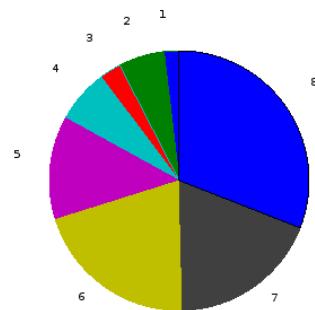
Configuration 1 :

0.4864865
0.2567568
0.2567568

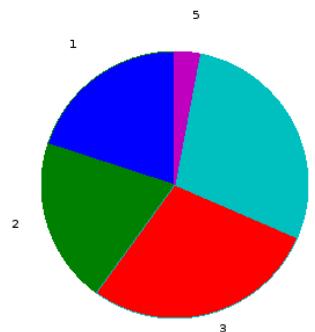


Configuration 2 :

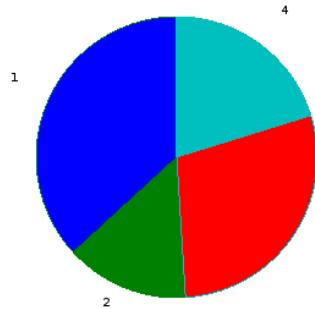
0.01875
0.0571505
0.0267188
0.0673279
0.1284873
0.2056777
0.1866015
0.3092864

**Configuration 3 :**

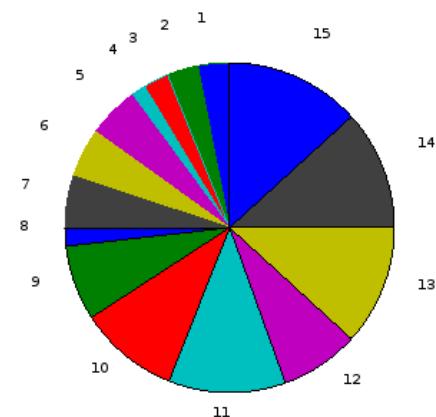
0.2
0.2
0.285
0.285
0.03

**Configuration 4 :**

0.3681507
0.1418094
0.2879616
0.2020783

**Configuration 5 :**

0.0311925
0.0303093
0.0248910
0.0148315
0.0498646
0.0483295
0.0515047
0.0170525
0.0735929
0.0984108
0.1149669
0.0774520
0.1184380
0.1166532
0.1325107



4.4.4 Extension de la méthode à d'autres problèmes

Il est intéressant de constater que la méthode de résolution mise en place pour résoudre le problème du *PageRank* ne se résume pas aux pages internet : on peut également s'intéresser à un système de connections interurbaines.

Considérons n villes reliées entre elles par un réseau ferroviaire. On désire connaître le taux d'accessibilité des villes entre elles pour mesurer le taux de facilité d'accès entre les villes. Il s'agit d'un problème complètement analogue à celui du *PageRank*. En effet, le problème se résout en obtenant le vecteur propre associé à la valeur λ de plus grand module de la matrice $A = (a_{ij})_{i,j=1,\dots,n}$ telle que :

$$a_{ij} = \begin{cases} 1 & \text{si la i-eme ville est reliée à la j-eme ville} \\ 0 & \text{sinon} \end{cases}$$

On s'inspire donc de la méthode et du code mis en place pour résoudre le problème du *PageRank* afin de déterminer l'importance des villes dans le réseau. On implémente ainsi le code (4.5).

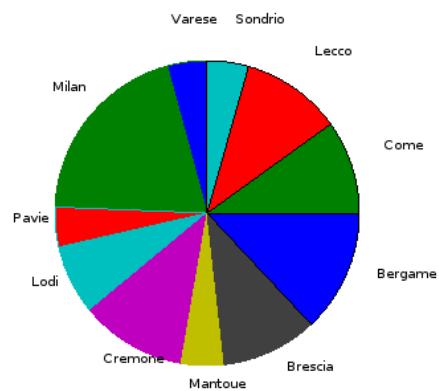
Code 4.5: Détermination de l'importance d'un ville dans un réseau ferroviaire

```

1  function A=constrA(G,n)
2      e=ones(n,1);
3      A=p*G+(1-p)/n*e*e';
4  endfunction
5
6  function H=constrH(G,n)
7      H=zeros(G);
8      for i=1:n
9          for j=1:n
10             s=sum(G(:,j));
11             if s == 0 then
12                 G(:,j)=ones(n,1)/n;
13                 s=1;
14             end
15             H(i,j)=G(i,j)/s;
16         end
17     end
18 endfunction
19
20 alpha=0.15;
21 p=1-alpha;
22 n=11; //nombre de villes
23 x=ones(n,1)/n; //equiprobabilité au départ
24 nom_villes=[ "Varese" , "Milan" , "Pavie" , "Lodi" , "Cremona" , "Mantoue" , "Brescia" ,
25           "Bergame" , "Come" , "Lecco" , "Sondrio" ];
26 G=sparse([1,2;2,1;2,3;2,4;2,9;2,8;2,7;3,2;4,2;4,5;5,4;5,6;5,7;6,5;7,5;7,2;
27           7,8;8,7;8,2;8,9;8,10;9,2;9,8;9,10;10,9,10,8;10,11,11,10], 
28           ones(28,1));
29 H=constrH(G,n);
30 A=constrA(H,n);
31
32 for i=1:100
33     x=A*x;
34     //on normalise x
35     x=x/sum(x);
36 end
37
38 disp(x);
39 pie(x,nom_villes);
```

On obtient les résultats ci-dessous.

(Varèse)	0.0421801
(Milan)	0.2014850
(Pavie)	0.0421801
(Lodi)	0.0740944
(Crémone)	0.1126387
(Mantoue)	0.0455507
(Brescia)	0.1016265
(Bergame)	0.1295628
(Côme)	0.0999746
(Lecco)	0.1068086
(Sondrio)	0.0438988



CHAPITRE 5

PROBLÈMES NON LINÉAIRES II

5.1 Introduction

5.2 Approche statistique

On a un jeu de données (x_i, y_i) . Considérons l'hypothèse suivante : les x_i sont donnés et les y_i sont des variables statistiquement indépendantes. On suppose qu'il existe un modèle parfait que l'on approche :

$$y_i = f(x_i, \theta) + \varepsilon_i$$

Où ε est une variable aléatoire d'espérance $E(\varepsilon_i) = 0$, de variance $Var(\varepsilon_i) = \sigma^2$ et de densité g . On cherche à trouver la densité de probabilité Φ de y , avec Φ_i de y_i définie par :

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

Si ε suit une loi normale, donc $g(\varepsilon) = (\sigma\sqrt{2\pi})^{-1} \exp(-\frac{1}{2\sigma^2}\varepsilon^2)$, on a :

$$\Phi_i(y_i, \theta) = (\sigma\sqrt{2\pi})^{-1} \exp\left(-\frac{1}{2\sigma^2}(y_i - f(x_i, \theta))^2\right)$$

Finalement, comme y_i sont des variables indépendantes, on a :

$$\Phi(y, \theta) = \prod_{i=1}^n \Phi_i(y_i, \theta)$$

On interprète alors ces formules :

pour $D \subset \mathbb{R}^n$, $\text{Prob}[y \in D | \theta] = \int_D \Phi(y, \theta) dy_1, \dots, dy_n$

Pour y fixé, alors $\Phi(y, \theta) = L(\theta, y)$ est appelée la fonction de vraisemblance de paramètre θ que l'on cherche évidemment à maximiser.

$$\begin{aligned} L(\theta, y) &= \prod_{i=1}^n \Phi_i(y_i, \theta) \\ &= \prod_{i=1}^n (\sigma\sqrt{2\pi})^{-1} \exp\left(-\frac{1}{2\sigma^2}(y_i - f(x_i, \theta))^2\right) \\ &= (\sigma\sqrt{2\pi})^{-n} \exp\left(-\frac{1}{2\sigma^2} \underbrace{\sum_{i=1}^n (y_i - f(x_i, \theta))^2}_{(*)}\right) \end{aligned}$$

Pour maximiser $L(\theta, y)$, on cherche donc à minimiser $(*)$, donc $\arg \max_{\theta \in \mathbb{R}^p} L(\theta, y) = \arg \min_{\theta \in \mathbb{R}^p} S(\theta)$, ce qui revient à dire que, résoudre le problème des moindres carrés revient à calculer le modèle dont la vraisemblance statistique est la plus grande (en supposant que l'erreur suit une loi normale).

5.3 Problèmes des moindres carrés linéaires

5.3.1 Théorie

On suppose donc que l'on possède un des données (x_i, y_i) et qu'il en existe un modèle tel que :

$$y = f(x, \theta) \text{ est linéaire} \Leftrightarrow y = \sum_{k=1}^p \theta_k \Phi_k(x) \text{ (pour } x \text{ fixé, } y \text{ est une combinaison linéaire des } \theta_k)$$

On cherche donc à minimiser :

$$S(\theta) = \sum_{i=1}^n (\theta_1 + \theta_2 x_i + \dots + \theta_p x_i^p - y_i)^2 = \|r(\theta)\|^2$$

On pose :

$$A = \begin{bmatrix} \theta_1 + \theta_2 x_1 + \dots + \theta_p x_1^p \\ \vdots \\ \theta_1 + \theta_2 x_n + \dots + \theta_p x_n^p \end{bmatrix} \quad (\text{A est appelée la matrice de Vandermonde})$$

Finalement, on a :

$$S(\theta) = \|r(\theta)\|^2 = \|A\theta - y\|$$

Et on cherche $\hat{\theta}$ tel que $\hat{\theta} = \arg \min_{\theta \in \mathbb{R}^p} S(\theta)$

On connaît une condition nécessaire : $\nabla S(\hat{\theta}) = 0$ où $\nabla S(\hat{\theta})$ est le gradient de S en $\hat{\theta}$.

Rappel : si $f : \mathbb{R}^n \rightarrow \mathbb{R}$ est une fonction différentiable en a , alors :

$$f(a+h) = f(a) + \nabla f(a)^T h + \|h\|\varepsilon(h) \text{ où } \lim_{h \rightarrow 0} \varepsilon(h) \rightarrow 0$$

Ainsi on calcule $\nabla S(\hat{\theta})$ en développant $S(\theta)$, ainsi :

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

Par identification, on a donc : $\nabla S(\theta) = 2A^T(A\theta - y)$.

Par ailleurs, on a le théorème :

Théorème :

Une solution au problème des moindres carrés est donnée par $\hat{\theta}$ qui vérifie

$$\nabla S(\hat{\theta}) = 0 \Leftrightarrow A^T A \hat{\theta} = A^T y \quad (5.1)$$

De plus, si $\text{rang}(A) =$ son nombre de colonnes, alors cette solution est unique.

Démonstration :

Équivalence

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

Unicité

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

Finalement, résoudre des problèmes de moindres carrés linéaires n'est histoire que d'algèbre linéaire. Par ailleurs, lorsque la solution est unique, dans *Scilab*, on peut directement résoudre θ en utilisant la commande *theta = A \ y*.

5.3.2 Application : Régression polynomiale avec validation

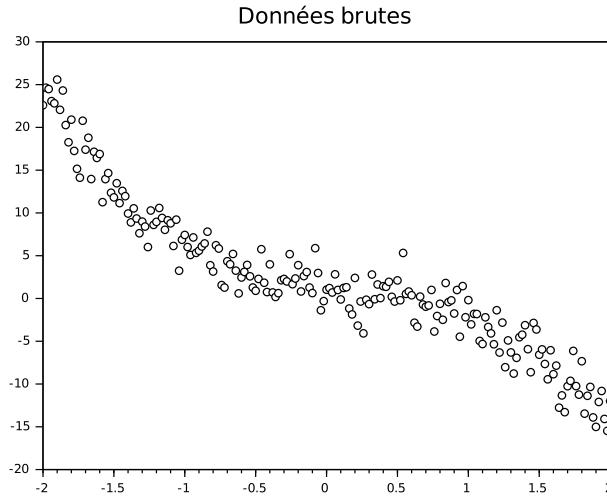
On applique donc la méthode étudiée pour un premier problème de simple régression linéaire. Le code (5.1) nous permet dans un premier temps de récupérer des données (un ensemble de couples (t_i, y_i)) et de les représenter sur un graphique (affichage (5.1)).

Code 5.1: Récupération des données

```

1 data=[...]; // ici les données
2 t=data(:,1);
3 y=data(:,2);
4 clf;
5 plot(t,y,'ko');
6 title("Donnees brutes",'fontsize',4);
```

Figure 5.1: Ensemble de données



On cherche donc à approcher le modèle inconnu par des polynômes de différents degrés. On implémente alors le code (5.2) qui nous donne les affichages (5.2) et (5.3).

Code 5.2: Approximation par différents polynômes

```

1 data=[...]; // ici les donnees
2 t=data(:,1);
3 y=data(:,2);

4
5 function return=constrA(t,y,d)
6     n=length(t);
7     A=ones(n,d+1);
8     for i=1:d
9         A(:,i+1)=t.^i;
10    end
11    return=A;
12 endfunction

13
14 function [theta , reg , erreur]=reglin(t,y,d)
15     A=constrA(t,y,d);
16     theta=A\y;
17     erreur=norm(A*theta-y)^2;
18     reg=A*theta;
19 endfunction

20
21 nom_degre=["degre 0","degre 1","degre 2","degre 3","degre 4","degre 5"];
22 erreur=zeros(6,1);
23 n=length(t);
24 for degré=0:5
25     subplot(3,2,degré+1);
26     plot(t,y,'ko');
27     [theta , reg , erreur(degré+1)]=reglin(t,y,degré);
28     plot(t,reg,"b");
29     title(nom_degre(degré+1));
30 end
31 scf(1);
32 plot((0:5),erreur');
33 title("Variation de l'erreur en fonction du degré du polynôme",'fontsize',4);

```

Figure 5.2: Affichage des différentes approximations

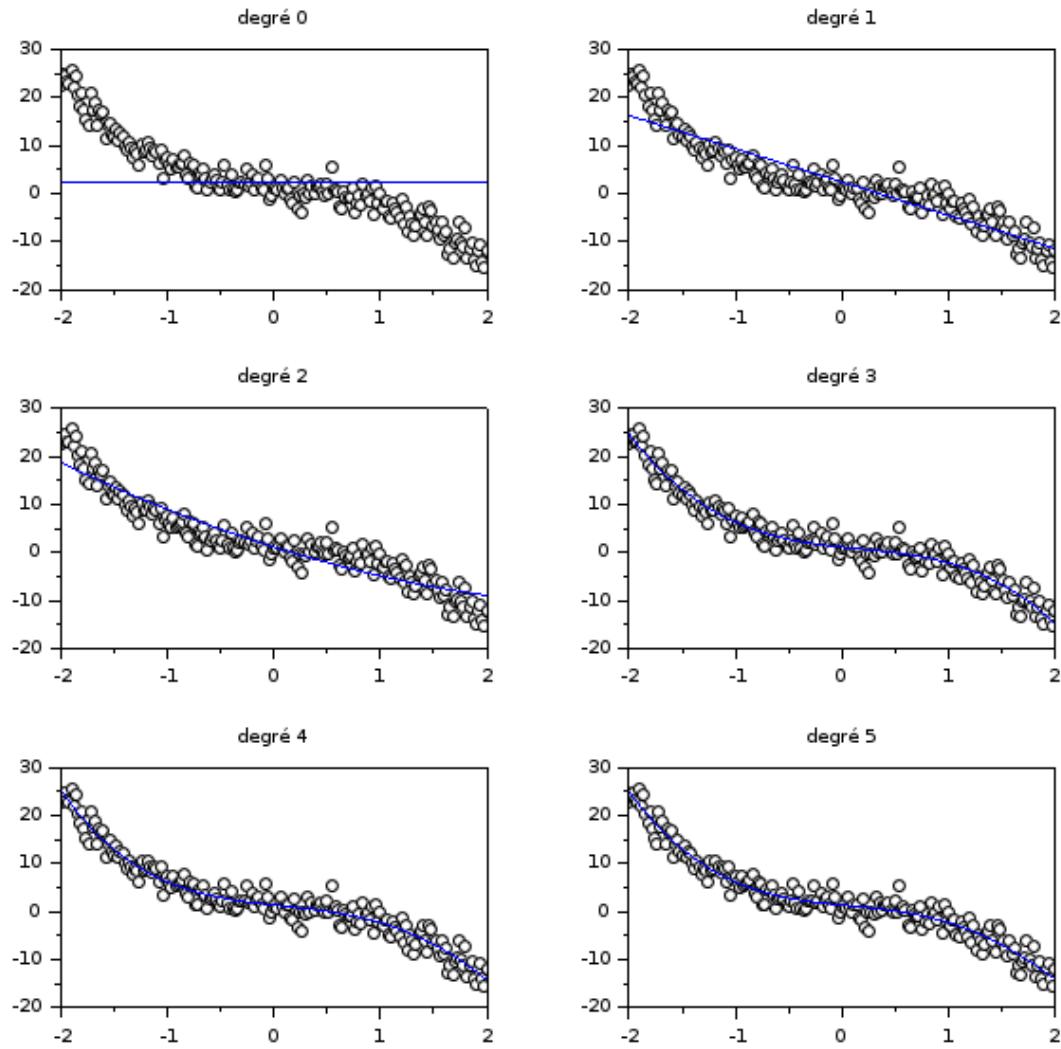
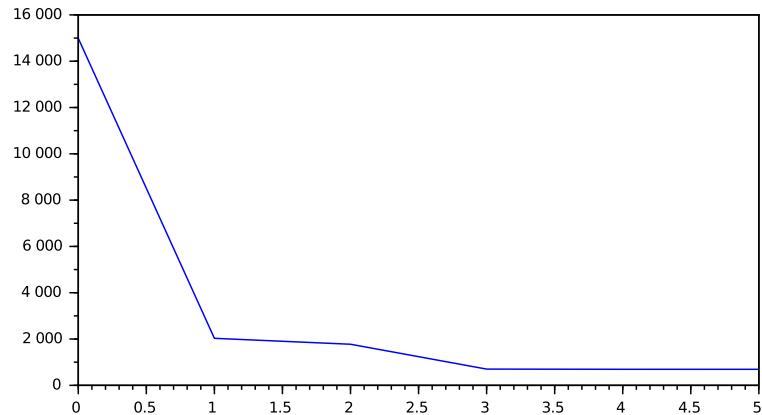


Figure 5.3: Évolution de l'erreur en fonction du degré du polynôme

Variation de l'erreur en fonction du degré du polynôme



Grâce aux tracés des différents polynômes (5.2) et en particulier au tracé de l'erreur en fonction du degré (5.3), on constate que l'erreur ne diminue plus de manière significative à partir du troisième degré. Dès lors, comment choisir le modèle qui représente le mieux les données ?

Pour répondre à cette problématique, on applique la méthode dite de validation. On considère un ensemble de données \mathcal{T} dit d'apprentissage et un ensemble de données \mathcal{V} dit de validation (il s'agit du complémentaire de \mathcal{T}). On applique alors la méthode de résolution du problème de moindres carrés sur \mathcal{T} pour obtenir un modèle. On applique ensuite ce modèle sur \mathcal{V} . On évalue l'erreur sur chacun des ensembles :

$$S_{\mathcal{T}}(\theta) = \sum_{i \in \mathcal{T}} (P(t_i) - y_i)^2 \quad S_{\mathcal{V}}(\theta) = \sum_{i \in \mathcal{V}} (P(t_i) - y_i)^2$$

L'erreur sur l'ensemble d'apprentissage $S_{\mathcal{T}}(\theta)$ ne nous donne pas plus d'information que l'erreur précédemment calculée sur l'ensemble des données. En revanche, l'erreur de validation $S_{\mathcal{V}}(\theta)$ nous permet de choisir un modèle car la différence entre chaque degré est bien plus significative (comme les données n'ont pas servi à l'élaboration du modèle, elles permettent de le valider ou non).

Dans notre cas, on choisit $\mathcal{V} = \{t \in [0, 2] \mid \text{abs}(t) \leq 1\}$. On implémente donc dans *Scilab* le code (5.3) qui nous donne les affichages (5.4) et (5.5).

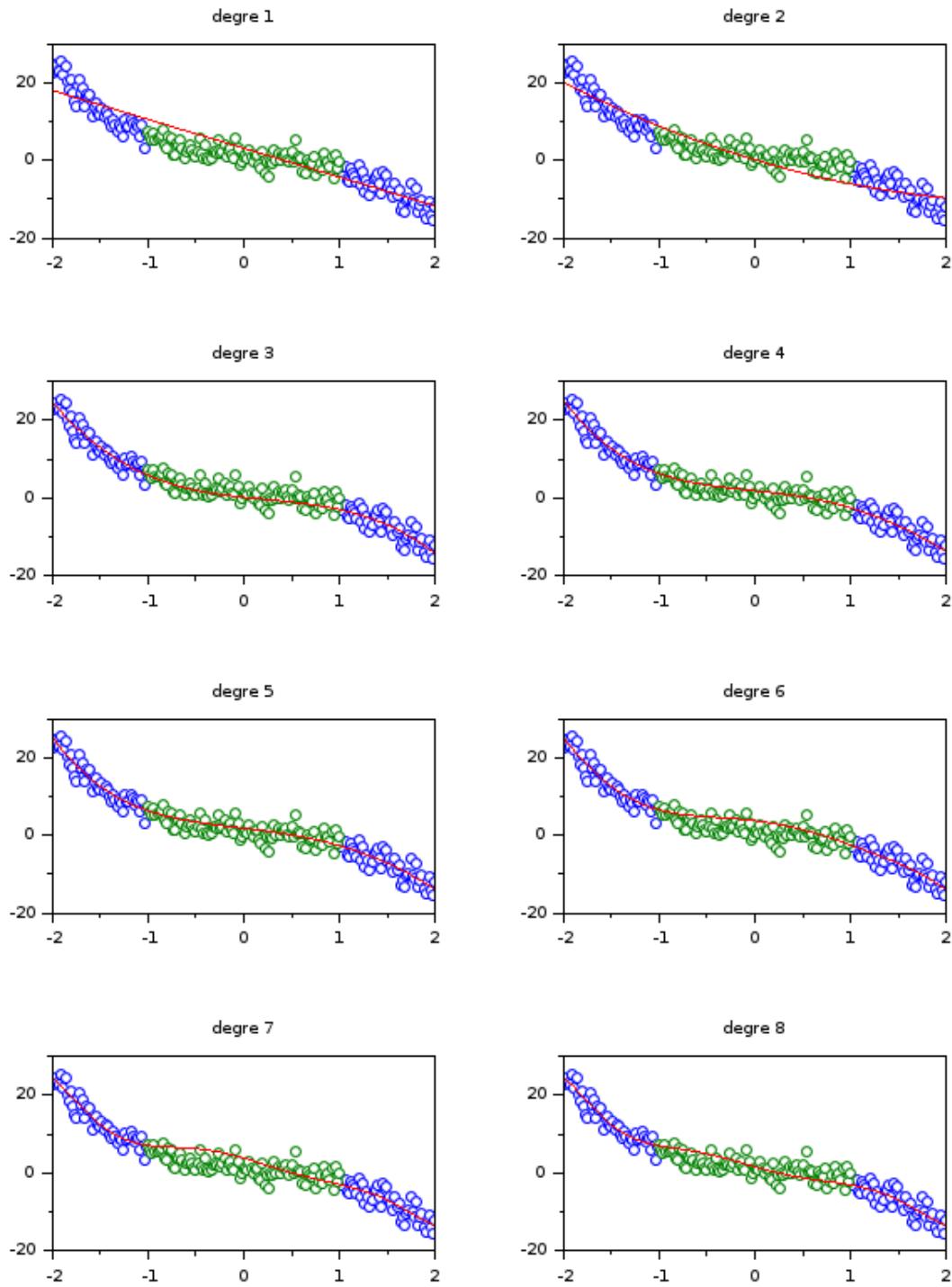
Code 5.3: Approximation par différents polynômes et méthode de validation

```

1 data=[...]; // ici les donnees
2 t=data(:,1); y=data(:,2);
3
4 function return=constrA(t,y,d)
5     n=length(t); A=ones(n,d+1); for i=1:d A(:,i+1)=t.^i; end return=A;
6 endfunction
7
8 function [theta ,reg ,erreur]=reglin(t,y,p)
9     A=constrA(t,y,p); theta=A\y; erreur=norm(A*theta-y)^2; reg=A*theta;
10 endfunction
11
12 function y=evalpoly(theta ,t)
13     y=zeros(t)+theta(1);
14     for i=2:length(theta)
15         y=y+theta(i)*t .^(i-1); end
16 endfunction
17
18 clf;
19 n=length(t);
20 V=find(abs(t)<=1); erreur_v=zeros(8,1); // validation
21 T=setdiff(1:n,V); erreur_t=zeros(8,1); // apprentissage
22
23 nom_degre=["degre 1","degre 2","degre 3","degre 4","degre 5","degre 6",
24             "degre 7","degre 8"];
25 for degré=1:8
26     [theta ,reg ,erreur_t(degré)]=reglin(t(T),y(T),degré);
27     yv=evalpoly(theta ,t(V));
28     erreur_v(degré)=(norm(y(V)-yv))^2;
29     subplot(4,2,degré);
30     plot(t(T),y(T), 'o', t(V),y(V), 'o', t , evalpoly(theta ,t ), 'r');
31     title(nom_degre(degré));
32 end
33
34 scf(1); plot(1:8 ,log(erreur_t) ,'-o',1:8 ,log(erreur_v) ,'-o');
35 title("Variation de l'erreur sur T et sur V en fonction du degré",
36       'fontsize',3);
37 legend(["Erreur d'apprentissage";"Erreur de validation"] ,opt=2);

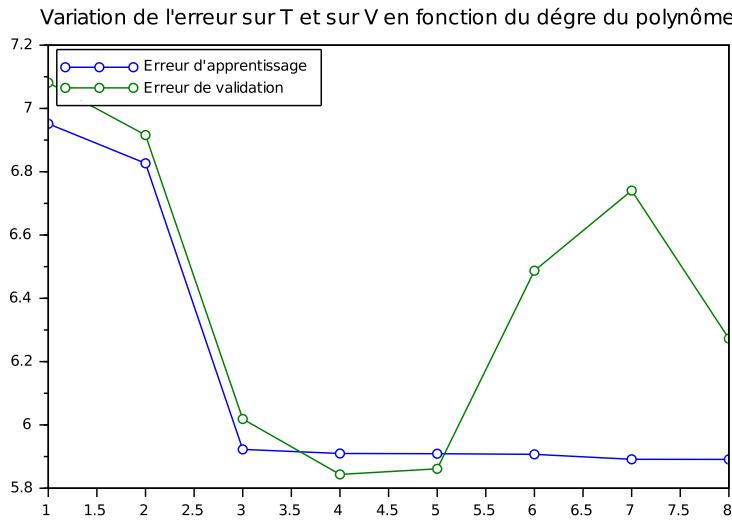
```

Figure 5.4: Affichage des différentes approximations sur T et V



Les points bleus et verts représentent réciproquement l'ensemble d'apprentissage et l'ensemble de validation. Le modèle (représenté par la courbe rouge) est élaboré sur l'ensemble d'apprentissage et est évalué sur l'ensemble des données (donc en particulier sur l'ensemble de validation).

Figure 5.5: Évolution de l'erreur sur T et sur V en fonction du degré du polynôme



Comme on pouvait s'y attendre, l'erreur d'apprentissage ne nous offre pas plus d'information concernant la précision du modèle en fonction du degré du polynôme. En revanche, l'erreur de validation nous montre quel modèle est le plus représentatif : il s'agit donc du degré 4.

5.4 Problèmes des moindres carrés non linéaires