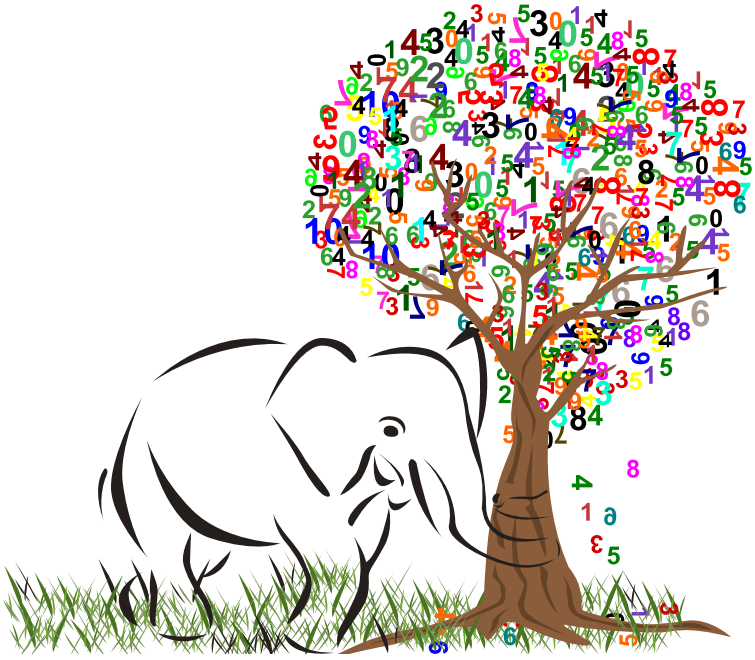
Université M. Khider de Biskra - Algérie
Département des sciences de la matière

Méthodes numériques via Matlab

Recueil de travaux pratiques corrigés

Recueil de travaux pratiques de l'analyse numérique, rédigé par :

Samir KENOUCHE
✉ kennouchesamir@gmail.com
✉ samir.kenouche@univ-biskra.dz



Important

Les programmes Matlab présentés dans ce recueil, sont exclusivement réservés aux étudiants de Département des sciences de la matière. Ce document peut être téléchargé et reproduit uniquement pour un usage individuel. La vente ou la reproduction comme support de cours payants est strictement interdite.

Préambule

Dans ce recueil de travaux pratiques, j'ai fait le choix de présenter systématiquement et succinctement les fondements théoriques de chaque méthode numérique avant d'entamer l'écriture des algorithmes. Ceci est primordial afin d'appréhender les différents concepts de l'analyse numérique mais également pour éveiller "l'instinct" de programmation chez l'étudiant(e). Tous les algorithmes sont écrits sous Matlab. Ce dernier, est pourvu d'une interface interactive et conviviale, et permet avec une grande flexibilité d'effectuer des calculs numériques et des visualisations graphiques de très haut niveau. Les notions abordées dans ce recueil sont : l'intégration numériques (méthode du point milieu, du trapèze et celle de Simpson), recherche de racines d'une fonction réelle de variable réelle (méthode de point fixe, Aitken-Shanks, dichotomie, Newton, sécante), l'interpolation polynomiale (méthode de Lagrange, celle de Hermite et Tchebychev) et la résolution numérique d'équations différentielles. Par ailleurs, l'objectif principal de ce recueil de travaux pratiques, est la mise à la disposition des étudiants(es) d'un outil pratique dédié au calcul numérique sur ordinateur. Il a aussi pour vocation de donner les connaissances de base nécessaires à la compréhension des algorithmes inhérents à cette discipline. Bien évidemment, la liste des méthodes numériques présentées ici est loin d'être exhaustive, sont présentées uniquement les méthodes les plus couramment utilisées en graduation. J'invite les lecteurs à signaler d'éventuelles erreurs et imperfections en envoyant un mail à l'adresse.

✉ kennouchesamir@gmail.com
☎ xx xx xx xx

Matlab est un produit de *The Mathswork Inc.*, www.mathswork.com

```
                    < M A T L A B (R) >
          (c) Copyright 1984-2008 The MathWorks, Inc.
                       All Rights Reserved
                    Version 7.6.0.324 (R2008a)
```

Notons au passage, que la société *MathWorks* commercialise deux versions de MATLAB annuellement. Les lettres a et b désignent respectivement les versions sorties en Mars et en Septembre.

Liste des Travaux pratiques

Liste des Figures

1. Intégration numérique

Méthodes du point milieu, du trapèze et de Simpson

Introduction

Très souvent le calcul explicite de l'intégrale, d'une fonction f continue sur $[a, b]$ dans $\mathbb{R}$, définie par $I(f) = \int_a^b f(x)\,dx$ peut se révéler très laborieux, ou tout simplement impossible à atteindre. Par conséquent, on fait appel à des méthodes numériques, afin de calculer une approximation de $I(f)$. Dans ces méthodes numériques, la fonction f, est remplacée par une somme finie. Dans ce TP, nous allons étudier et implémenter, sous Matlab, quelques méthodes usuelles (`point mileu, trapèze` et `Simpson`) dédiées à l'intégration numérique.

1.1 Méthode du point milieu

Soit f une fonction continue sur l'intervalle $[a, b]$, par définition son intégrale se calcule suivant :

$$I(f) = \int_a^b f(x)\,dx \tag{1}$$

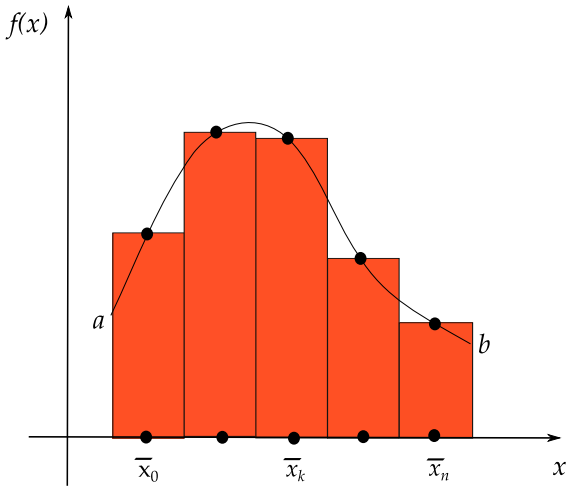
Notons que l'expression analytique de $f(x)$ peut être connue comme elle peut être inconnue.

L'idée de base de cette méthode, consiste à subdiviser l'intervalle $[a, b]$ en n sous-intervalles $[x_k, x_{k+1}]$. Dans le cas où les sous-intervalles sont équidistants, on écrira $\Delta x = (b-a)/n$. Ainsi, le schéma numérique de cette méthode s'écrira comme :

$$I(f) = \sum_{k=1}^{n} \int_{I_k} f(x)\,dx \tag{2}$$

$$I(f) = \Delta x \times \sum_{k=1}^{n} f(\bar{x}_k) \qquad \text{avec} \qquad \bar{x} = \frac{x_{k+1} - x_k}{2} \tag{3}$$

Comme illustré par l'Éq. (3), pour chaque $\Delta x = [x_k, x_{k+1}]$ on prend le point central de l'ordonnée $f(\bar{x})$, valeur médiane entre $f(x_k)$ et $f(x_{k+1})$. $I(f)$, comme illustré sur la figure ci-dessous, représente l'aire comprise entre la courbe $y = f(x)$ et l'axe des



abscisses entre les droites $x = a$ et $x = b$. Par ailleurs, notons qu'il existe plusieurs façons de mettre en œuvre la méthode des rectangles. Ainsi, on a également la possibilité de prendre la borne inférieure ou bien la borne supérieure sur chaque sous-intervalle $[x_k, x_{k+1}]$.

1.2 Méthode du trapèze

Cette méthode est basée sur l'interpolation, de chaque sous-intervalle $[x_k, x_{k+1}]$, par un polynôme de degré un. En d'autres mots, sur chaque $[x_k, x_{k+1}]$, la fonction f continue et dérivable sur $[a, b]$, est substituée par la droite joignant les points $(x_k, f(x_k))$ et $(x_{k+1}, f(x_{k+1}))$. Le schéma numérique de la méthode du trapèze est donné par :

$$I(f) \approx \frac{\Delta x}{2}\left[f(a) + f(b)\right] + \Delta x \times \sum_{k=1}^{n-1} f(x_k) \tag{4}$$

La méthode du trapèze $(O(h^2))$ fournit une bien meilleure précision que la méthode du point milieu $(O(h))$.

1.3 Méthode de Simpson

Cette méthode est basée sur l'interpolation, de chaque sous-intervalle $[x_k, x_{k+1}]$, par un polynôme de degré deux. Ainsi, la fonction f est substituée par ce polynôme du second degré qui définit donc un arc de parabole passant par les points d'ordonnées $f(x_k)$, $f(x_{k+1})$ et $f(x_{k+2})$. Le schéma numérique de cette méthode est donné par :

$$I(f) \approx \frac{\Delta x}{6}\left(f(a) + f(b) + 2 \times \sum_{k=1}^{n-1} f(x_k) + 4 \times \sum_{k=1}^{n-1} f\left(\frac{x_{k+1} - x_k}{2}\right)\right) \tag{5}$$

La méthode de Simpson $(O(h^4))$ fait mieux que celle du trapèze. Ceci provient du fait qu'elle pondère plus le point central.

Énoncé du TP ❶ ☞ ®

- Calculez les approximations de l'intégrale :

$$I(f) = \int_0^{2\,\pi} x\, exp(-x)\, cos(2\,x)\, dx \simeq -0.122122604618968 \qquad (6)$$

  en utilisant les méthodes du point milieu, du trapèze et de Simpson. Conclure.
- Tracer l'aire de l'intégrale, pour `n = 150` sous-intervalles.
- Étudier l'influence du nombre de sous-intervalles (`n`) sur l'erreur d'intégration.
- Appliquez les mêmes étapes pour l'intégrale :

$$I(f) = \int_0^1 exp\left(-\frac{x^2}{2}\right) dx \simeq +0.856086378341836 \qquad (7)$$

Note

L'algorithme du point milieu doit être écrit de deux façons différentes. D'une part, en utilisant la boucle for et d'autre part, au moyen des fonctions préprogrammées sum et linspace

Script Matlab

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 15/11/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DU POINT MILIEU, du
% TRAPEZE ET DE SIMPSON

%%%%%%%%%%%%%%%%%%%%%%% POINT MILIEU : 1ERE APPROCHE %%%%%%%%%%%%%%%%%%
a = 0 ; b = 2*pi ; n = 100 ; dx = (b - a)/n ;
x = a:dx:b ; fun =  x.*exp(-x).*cos(2.*x) ; int = 0;

for ik = 1:length(x)-1

    xbar = (x(ik) + x(ik+1))/2;

    func = eval('fun', xbar);

    int =  int + dx*func(ik);
end

disp(strcat('L''INTEGRALE, PAR LA METHODE DU POINT MILIEU VAUT :',...
num2str(int)))
```

```
22
23 %%%%%%%%%%%%%%%%%%% POINT MILIEU : 2EME APPROCHE %%%%%%%%%%%%%%%%%%
24 a = 0 ; b = 2*pi ; n = 100 ; dx = (b - a)/n ; x = a:dx:b ;
25 fun =  inline('x.*exp(-x).*cos(2.*x)') ; x = linspace(a+dx,b-dx,n);
26 fun = feval(fun ,x) ; int = dx.*sum(fun)
27
28 %%%%%%%%%%%%%%%%%%%%%%%%%%%% TRAPEZE %%%%%%%%%%%%%%%%%%%%%%%%%%%%
29 a = 0 ; b = 2*pi ; n = 1000 ; dx = (b - a)/n; x = a :dx: b ;
30 f =  x.*exp(-x).*cos(2.*x) ; int = 0 ; init = (f(1)+ f(end))*(dx/2) ;
31
32 for ik = 1:length(x)-1
33
34     int = int + dx*f(ik);
35 end
36 int = init + int
37 disp(strcat('L''INTEGRALE, PAR LA METHODE DU TRAPEZE VAUT :',...
38 num2str(int)))
39
40 %%%%%%%%%%%%%%%%%%%%%%%%%% Simpson %%%%%%%%%%%%%%%%%%%%%%%%%%
41 clear all; clc; close all;
42
43 a = 0 ; b = 2*pi ; n = 1000 ; dx = (b - a)/(n) ; x = a :dx: b ;
44 fun =  x.*exp(-x).*cos(2.*x) ; som1 = 0 ; som2 = 0 ;
45
46 for ik = 1:n-1
47
48     som1 = som1 + fun(ik);
49
50     xbar = (x(ik+1) - x(ik))/2;
51
52     fun2 = eval('fun',xbar);
53     som2 = som2 + fun2(ik);
54 end
55
56 int = (dx/6)*(fun(1) + fun(end) + 2*som1 + 4*som2)
57 disp(strcat('L''INTEGRALE, PAR LA METHODE DE SIMPSON VAUT :',...
58 num2str(int)))
59 %%%%%%%%%%%%%%%%%%%%%%%% AIRE DE L'INTEGRALE %%%%%%%%%%%%%%%%%%%%%%%%
60
61 figure('color',[1 1 1]) ; plot(x, fun,'LineWidth', 1) ; hold on
62
63 for ik = 1 : numel(x)
64
65  plot([x(ik), x(ik)],[0, fun(ik)],'r','LineWidth', 1)
```

```
66
67 end
68
69 ih = gca ;
70 str(1) = {'Integral area of:'};
71 str(2) = {['$$\int_{0}^{2\pi}x \, \exp(-x) \, \cos(2 \, x) \, dx = $$
      ',num2str(int)]} ; set(gcf,'CurrentAxes', ih) ;
72 text('Interpreter','latex', 'String',str,'Position',[3 -0.2],'
      FontSize',12) ;  xlabel('x') ; ylabel('f(x)') ;
```

Les résultats de l'intégrale, calculés par les trois méthodes, sont affichés sous Matlab comme suit :

```
L'INTEGRALE, PAR LA METHODE DU POINT MILIEU VAUT : -0.1228
L'INTEGRALE, PAR LA METHODE DU TRAPEZE VAUT : -0.1222
L'INTEGRALE, PAR LA METHODE DE SIMPSON VAUT : -0.1221
```

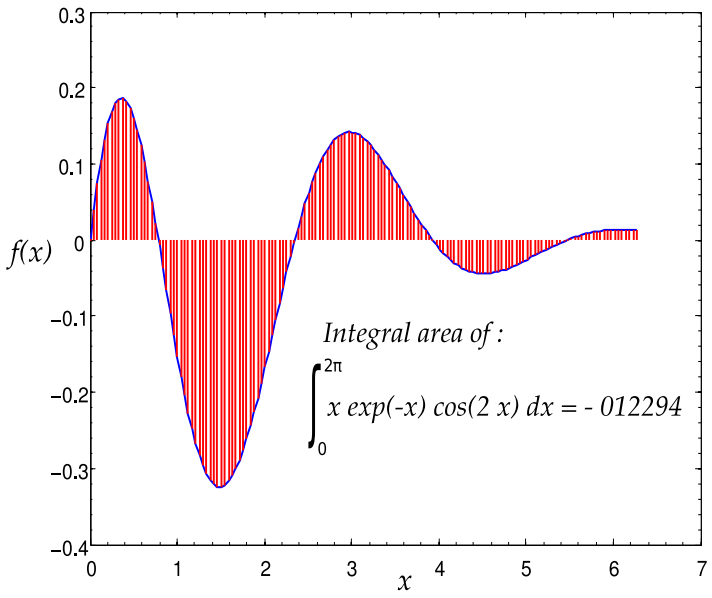


FIGURE 1: Aire de l'intégrale

Influence du nombre de sous-intervalles sur l'erreur d'intégration, voici le script Matlab :

```
1 clear all ; clc ; close all ; format long
2 %%%%%%%%%%%%%%%%%%%%%%%% ERREUR D'INTEGRATION %%%%%%%%%%%%%%%%%%%%%%%%
3 a = 0 ; b = 2*pi ; n = 150 ; ih = 0 ; intexact = - 0.12212260461896 ;
4
5 for n = 60 : 20 : 600
6
```

```
    dx = (b - a)/n ; x = a :dx: b ; fun = x.*exp(-x).*cos(2.*x) ;
    som1 = 0 ; som2 = 0 ; ih = ih + 1 ;

    for ik = 1:n-1

    som1 = som1 + fun(ik) ; xbar = (x(ik+1) - x(ik))/2;
    fun2 = eval('fun',xbar) ; som2 = som2 + fun2(ik);

    end

    int(ih) = (dx/6)*(fun(1) + fun(end) + 2*som1 + 4*som2) ;
    err(ih) = abs((int(ih) - intexact)/intexact) ; % ERREUR RELATIVE
plot(n, err(ih), '+','MarkerSize',8, 'LineWidth', 1/2) ; hold on ;
xlabel('NOMBRE DE SOUS-INTERVALLES') ; ylabel('ERREUR RELATIVE D''
   INTEGRATION (x 100 \%)') ;
end
```

Le graphe généré par ce script Matlab est :

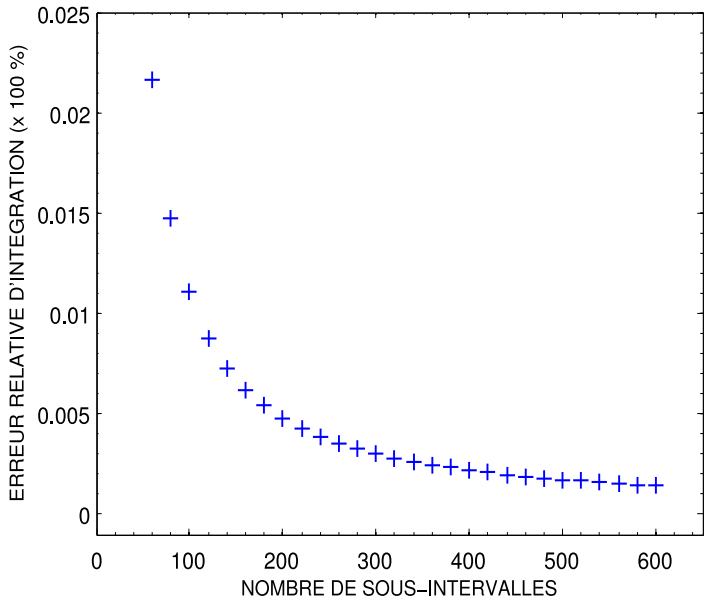


FIGURE 2: Influence du nombre de sous-intervalle sur l'erreur d'intégration

À partir de ce graphique, il apparait clairement que plus le nombre de sous-intervalles est élevé plus l'erreur d'intégration est faible. Ce résultat s'explique par le fait que plus n (nombre de sous-intervalles) est grand plus on s'approche de la forme continue de la fonction à intégrer. Autrement dit, plus n est élevé plus l'égalité $\int_n = \lim_{n \to +\infty} \sum_n$ est vraie.

1.4 Au moyen de routines Matlab

Introduction

Il existe des routines Matlab prédéfinies qui permettent de résoudre numériquement des intégrales simple, double et triple. Ces commandes sont les suivantes : `quad`, `quadl`, `quadgk` et `quadv`. Les commandes relatives au calcul d'intégrales double et triple seront abordées au cours de la séance. Nous donnons dans le script Matlab ci-dessous, un exemple d'utilisation de ces commandes pour le calcul d'intégrales simples.

Énoncé du TP ❷ ☞ ®

- Au moyen des commandes `quad`, `quadl` et `quadgk`, évaluer l'intégrale :

$$I(f) = \int_0^{2\,\pi} x\, exp(-x)\, cos(2\,x)\, dx \simeq -0.122122604618968 \tag{8}$$

- Se servant de la commande `quadv`, évaluer l'intégrale :

$$\int_0^{2\,\pi} x\, exp(-x)\, cos(k\,x)\, dx \tag{9}$$

  pour différentes valeurs du paramètre `k = 2 : 24`.
- Tracer le courbe représentant la valeur de l'intégrale en fonction du paramètre `k`.

✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎ Script Matlab ✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎

```
clear all ; clc ;
%%%%%%%%%%%%%%%%%% AU MOYEN DES ROUTINES MATLAB %%%%%%%%%%%%%%%%%%
% @copyright 16/11/2015 Samir KENOUCHE

fun = @(x) x.*exp(-x).*cos(2.*x) ;

lowerBound = 0 ;  upperBound = 2*pi; tol = 1e-07 ;

[int1, funEval1] = quad(fun, lowerBound, upperBound, tol) % METHODE
% ADAPTATIVE DE Simpson. funEval1 : Nbr D'EVALUATIONs DE fun

[int2, funEval2] = quadl(fun, lowerBound, upperBound, tol) % METHODE
%ADAPTATIVE DE Lobatto

[int3, errorbnd] = quadgk(fun, lowerBound, upperBound,'RelTol',...
    1e-8,'AbsTol',1e-12) % METHODE ADAPTATIVE DE Gauss-Kronrod
% errorbnd : TRADUIT L'ERREUR ABSOLUE D'INTEGRATION |Q - I|, AVEC
% Q ET I SONT RESPECTIVEMENT LES VALEURS APPROCHHE ET EXACTE.

n = 1000 ; dx = (upperBound - lowerBound)/n ;
```

```
x = lowerBound : dx : upperBound ;

int4 = trapz(x, fun(x)) ; % METHODE DU TRAPEZE

for k = 2:14  % POUR LES INTEGRALES PARAMERTRIQUE

[int5, funEval4] = quadv(@(x) x.*exp(-x).*cos(k.*x), lowerBound,...
 upperBound);

figure(1) ; hold on
plot(k,int5,'+','MarkerSize',10,'LineWidth',1)
xlim([1 15]) ; xlabel('VALEUR DU PARAMETRE k','FontSize',12) ;
ylabel('VALEUR DE L''INTEGRALE','FontSize',12)

end
```

Ci-dessous, les résultats renvoyés par le script Matlab.

```
int1 = -0.1221 ; funEval1 = 125
int2 = -0.1221 ; funEval2 = 48
int3 = -0.1221 ; errorbnd = 1.4734e-15
```

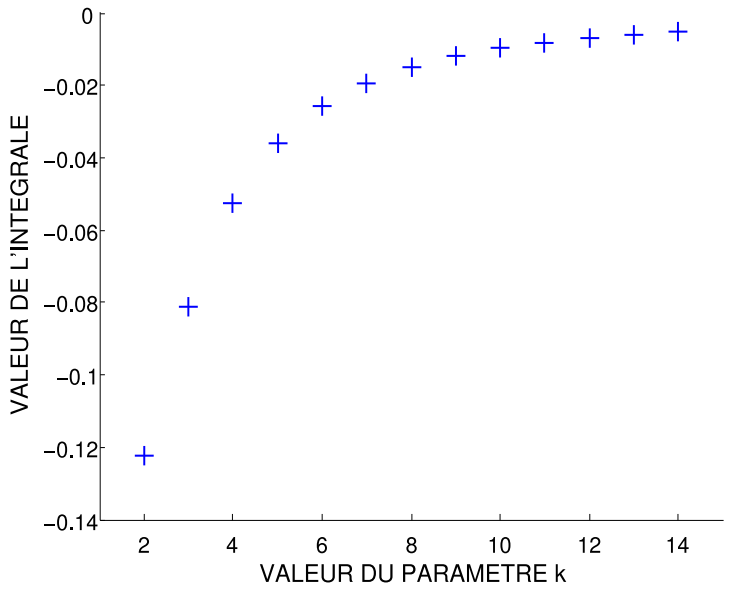


FIGURE 3: Influence du paramètre k sur la valeur de l'intégrale

À partir de ce graphe, on observe que plus le paramètre k augmente, plus la valeur de l'intégrale est élevée. Il s'agit donc d'une corrélation positive.

2. Résolution d'équations non-linéaires

Méthodes du point fixe et de dichotomie

Introduction

Il existe toute une panoplie de méthodes numériques (dichotomie, point fixe, `Newton`, `Lagrange`, ... etc) permettant de trouver numériquement les zéros de fonction $f(x) = 0$ d'une variable réelle. La majorité de ces méthodes sont itératives. En d'autres mots, elles calculent des approximations successives $x_1, x_2, x_3, ...$ de la véritable racine x^* de l'équation $f(x) = 0$, à partir d'une valeur initiale x_0 plus au moins bien choisie. Ce qui les distingue, entre autre, c'est leurs vitesses de convergence et leurs robustesses. Dans certaines applications, cette vitesse de convergence devient un facteur déterminant notamment quand il s'agit de calculer les racines d'une constellation de fonctions.

2.1 Méthode du point fixe

Le principe de la méthode du point fixe consiste à transformer, la fonction $f(x) = 0$, $f : [a \quad b] \longrightarrow R$, en une fonction $\varphi(x) = x$. La fonction $\varphi : [a \quad b] \longrightarrow R$, est construite de façon à ce que $\varphi(\alpha) = \alpha$ quand $f(\alpha) = 0$. Trouver la racine de $f(x)$, se résume donc à déterminer un $\alpha \in [a \quad b]$ tel que :

$$\alpha = \varphi(\alpha) \tag{10}$$

Dans le cas où un tel point existe, il sera qualifié de `point fixe` de φ et cette dernière est dite `fonction d'itération`. Le schéma numérique de cette méthode est donné par :

$$x^{(k+1)} = \varphi(x^{(k)}) \quad pour \quad k \geq 0 \tag{11}$$

On rappelle que le vecteur erreur e_n est calculé à partir de : $e_n = |x_n - x_{app}|$. Avec, x_{app} est la solution approchée, de la valeur exacte, déterminée avec une tolérance fixée préalablement. n, étant le nombre d'itérations. Par ailleurs, l'estimation de l'erreur servira, entre autre, à comparer la vitesse de convergence pour des méthodes numériques différentes. Sur le plan pratique, l'erreur est représentée graphiquement en traçant e_{n+1} en fonction de e_n avec une échelle logarithmique. Ainsi, `l'ordre`, noté p, d'une méthode numérique s'obtient à partir de :

$$|e_{n+1}| \approx A\,|e_n|^p \Longrightarrow \log|e_{n+1}| \approx p\,\log|e_n| + \log A \tag{12}$$

Ainsi l'ordre, p, est quantifié via la pente de l'équation ci-dessus. On en déduira que :

1. Si $p = 1 \Longrightarrow x_n$ converge `linéairement` vers la solution approchée. Dans ce cas on gagne la même quantité de précision à chaque itération.
2. Si $p = 2 \Longrightarrow x_n$ converge `quadratiquement` vers la solution approchée. Dans ce cas on gagne le double de précision à chaque itération.
3. Si $p = 3 \Longrightarrow x_n$ converge `cubiquement` vers la solution approchée. Dans ce cas on gagne le triple de précision à chaque itération.

D'un point de vue pratique, et pour un n suffisamment élevé, la vitesse de convergence d'une méthode itérative est évaluée au moyen de la relation :

$$K_p(x,n) = \frac{x_{n+2} - x_{n+1}}{(x_{n+1} - x_n)^p} \tag{13}$$

Tenant compte de cette équation, il vient que plus $K_p(x,n)$ tend vers zéro plus la vitesse de convergence de la méthode est élevée.

Énoncé du TP ❸ ☞ ®

Dans ce TP, il est demandé de trouver la racine de la fonction $f_1(x) = x - cos(x)$, en utilisant la méthode du `point fixe`

1. Tracer la fonction $f_1(x)$ sur l'intervalle $[-1/2 \quad 3]$.
2. Écrire un programme Matlab permettant l'implémentation du schéma numérique de cette méthode.
3. Afficher, sur le même graphe, la fonction $f_1(x)$ et la solution approchée.
4. Afficher le graphe représentant le nombre d'approximations successives $x^{(k+1)}$ en fonction du nombre d'itérations.
5. Tracer l'évolution de l'erreur en fonction du nombre d'itérations.
6. Tracer l'erreur $\ln|e_{n+1}|$ en fonction de $\ln|e_n|$ et déterminer l'ordre de la méthode numérique.
7. Calculer la vitesse de convergence de la méthode numérique.

Appliquez le même algorithme pour résoudre l'équation : $f_2(x) = x + exp(x) + 1$ avec $x \in [-2 \quad 3/2]$.

On donne : tolérance $= 10^{-6}$, *les valeurs initiales sont* $x_0 = 0.8$ *pour* $f_1(x)$ *et* $x_0 = -1/5$ *pour* $f_2(x)$

✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎ Script Matlab ✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 15/11/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DU POINT FIXE
x0 = 0.8; it = 0; tol = 1e-05; Nbrit = 30;

while it < Nbrit

    it = it +1;
    x = cos(x0) ;
    x0 = x ;

    xn(it) = x;

      if abs(x - cos(x)) < tol

        sol = x;
        break
    end
end

a = 0; b = 3; n = 1000; dx = (b - a)/n; x1 = a:dx:b;
y = inline('cos(x)');

figure('color', [1 1 1])
plot(x1, x1,'k') ; hold on ; plot(x1, y(x1),'LineWidth',2) ; hold on
    plot(sol,y(sol),'ro','MarkerSize',12,'LineWidth',2)
     hold on ; plot(sol,y(sol) ,'rx','MarkerSize',12,'LineWidth',2)

hold on
    plot(sol,0,'ko','MarkerSize',12,'LineWidth',2) ; hold on
 plot(sol,0 ,'kx','MarkerSize',12,'LineWidth',2) ; hold on
 line(x1, zeros(1, length(x1)),'LineStyle','-.','Color','k','
   LineWidth',1)
xlabel('x') ; ylabel('f(x)')

hold on ; line(sol.*ones(1, length(x1)), y(x1),'LineStyle','-.','
   Color','k','LineWidth',1) ; axis([0 3 -1 1.5])

%%%%%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
39 text('Interpreter','latex',...
40  'String','$ x = cos(x) $','Position',[2 -1/3],'FontSize',15)
41 text('Interpreter','latex','String','$ y = x $',...
42  'Position',[1.2 1],'FontSize',15) ; text(sol,2*sol,['\alpha = ',
      num2str(sol)],...
43  'Position',[0.8 -1/4],'BackgroundColor',[1 1 1]);
44
45 %%%%%%%%%%%%%%%%%%%%%%%% CALCUL D'ERRRUR %%%%%%%%%%%%%%%%%%%%%%%%
46 err = abs(xn - sol); error_1 = err(1:end-1) ; error_2 = err(2: end);
47
48 figure('color', [1 1 1]) ; loglog(error_1(1:end-4),error_2(1:end-4),'
      +')
49 ylabel('Ln |e_{n+1}|') ;  xlabel('Ln |e_{n}|')
50
51 cutoff = 7;  % ATTENTION AU CHOIX DE LA NIEME ITERATION
52
53 pente = (log(error_2(end-cutoff)) - log(error_2(end-(cutoff+1))))/(
      log(error_1(end-cutoff)) - log(error_1(end-(cutoff+1)))); %
      VITESSE DE CONVERGENCE log|en+1| = log|en|
54 pente = round(pente);
55
56 %%%%%%%%%%%%%%%%%% VITESSE DE CONVERGENCE (CV) %%%%%%%%%%%%%%%%%%
57 n = 20;  % ATTENTION AU CHOIX DE LA NIEME ITERATION
58 vitesse_CV = (xn(n+2) - xn(n+1))/(xn(n+1) - xn(n));
59
60 msg1 = gtext(strcat('CETTE METHODE EST D''ORDRE : ', num2str(pente)))
      ;% CLIQUER SUR LA FIGURE POUR AFFICHER msg1
61
62 msg2 = gtext(strcat('LA VITESSE DE CONVERGENCE VAUT : ', num2str(
      vitesse_CV)));
63 % CLIQUER SUR LA FIGURE POUR AFFICHER msg2
```

Les résultats des différents calculs sont portés sur les figures suivantes

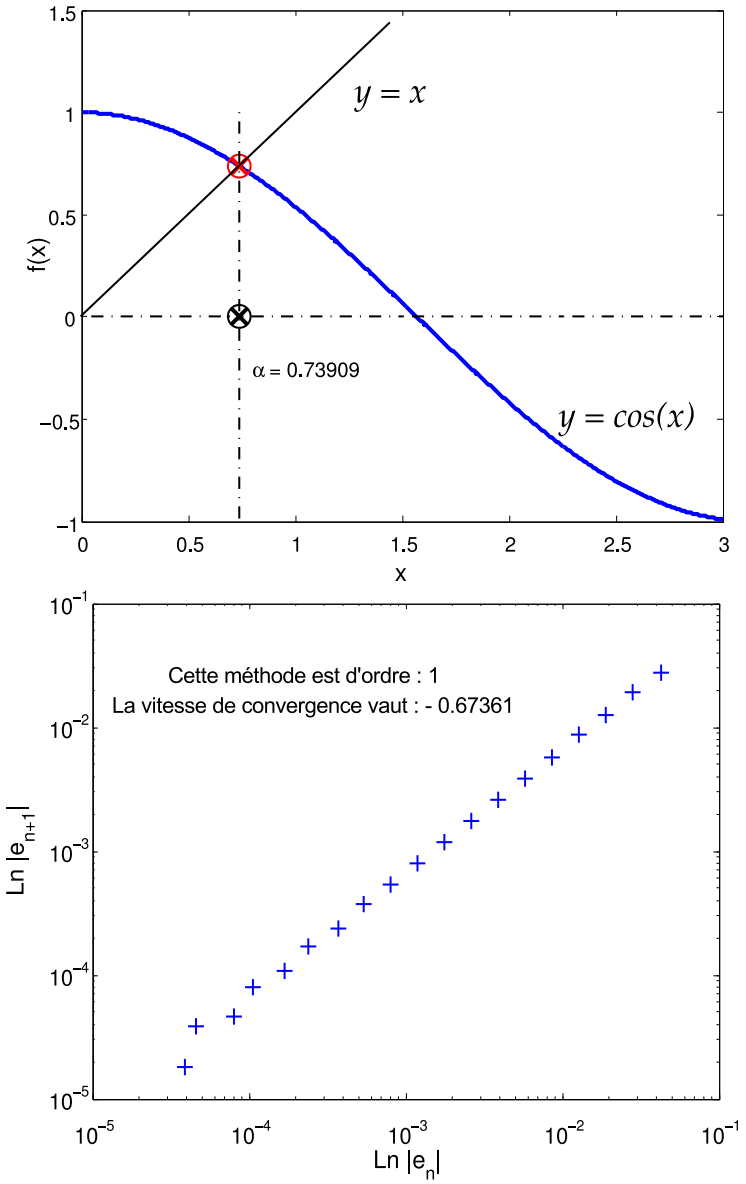


FIGURE 4: Racine de f et ordre de convergence de la méthode du `point fixe`

L'inconvénient de la méthode du point fixe, est qu'elle converge trop lentement. Afin d'améliorer sa convergence on peut la transformer en une nouvelle suite par le biais de l'algorithme d'accélération de convergence. Cet algorithme est connu sous le nom de `Aitken-Shanks`.

$$x^{(k+1)} = x^{(k)} - \frac{(\varphi(x^{(k)}) - x^{(k)})^2}{\varphi(\varphi(x^{(k)})) - 2\,\varphi(x^{(k)}) + x^{(k)}} \tag{14}$$

L'algorithme de `Aitken-Shanks` peut être appliqué à toute méthode de point fixe. Très souvent la convergence de cette méthode est très rapide.

Script Matlab

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 28/11/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE Aitken-Shanks
xinit = 0.8 ; it = 0 ; tol = 1e-05 ; Nbrit = 30 ;

while it < Nbrit
    it = it +1;
    phi = cos(xinit);
    phiphi = cos(phi);

    x = xinit - ((phi - xinit).^2)/(phiphi - 2*phi + xinit);
    xinit = x ;
    xn(it) = x;

      if abs(x - cos(x)) < tol

        sol = x;
        break
    end

end
```

À titre comparatif et tenant compte des deux codes Matlab présentés ci-dessus, la méthode du point fixe converge vers la racine approchée `sol = 7.390850858357935e-01`, au bout de 24 itérations. Alors que celle de `Aitken-Shanks` converge vers la même solution au bout de 2 itérations seulement. Ainsi, le gain en terme du temps d'exécution apparait clairement. La convergence est 12 fois plus rapide avec l'algorithme de `Aitken-Shanks`.

2.2 Méthode de dichotomie

Le principe de la méthode de `dichotomie`, encore appelée méthode de `bissection`, est basé sur le théorème de la valeur intermédiaire. La méthode est décrite comme suit : soit, $f : [a \quad b] \longrightarrow R$, une fonction continue sur l'intervalle $[a \quad b]$. Si $f(a) \times f(b) < 0 \longrightarrow$ il existe donc au moins une racine de $f(x)$ appartenant à l'intervalle $[a \quad b]$. On prend $c = \dfrac{a+b}{2}$ la moitié de l'intervalle $[a \quad b]$ tel que :

1. Si $f(c) = 0 \longrightarrow c$ est la racine de $f(x)$.
2. Sinon, nous testons le signe de $f(a) \times f(c)$ (et de $f(c) \times f(b)$).
3. Si $f(a) \times f(c) < 0 \longrightarrow$ la racine se trouve dans l'intervalle $[a \quad c]$ qui est la moitié de $[a \quad b]$.
4. Si $f(c) \times f(b) < 0 \longrightarrow$ la racine se trouve dans l'intervalle $[c \quad b]$ qui est la moitié de $[a \quad b]$.

Ce processus de division, par deux, de l'intervalle (à chaque itération on divise l'intervalle par deux) de la fonction est réitéré jusqu'à la convergence pour la tolérance considérée. Ainsi, pour la nième itération, on divise : $[a_n \quad b_n]$ en $[a_n \quad c_n]$ et $[c_n \quad b_n]$, avec à chaque fois $c_n = \dfrac{a_n + b_n}{2}$.

Énoncé du TP ❹ ☞ Ⓡ

Dans ce TP, il est demandé de trouver la racine de $f(x) = x + exp(x) + \dfrac{10}{1+x^2} - 5$, en utilisant la méthode de `dichotomie`

1. Écrire un programme Matlab permettant l'implémentation du schéma numérique de cette méthode.
2. Afficher, sur le même graphe, la fonction $f(x)$, la solution approchée et les approximations successives.
3. Calculer l'ordre et la vitesse de convergence de la méthode numérique.

On donne : tolérance $= 10^{-8}$, $a_0 = -1.30$ *et* $b_0 = 3/2$

✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎ Script Matlab ✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 14/11/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DE DICHOTOMIE
a = -1.3 ;   b = 3/2 ; itmax = 100 ; tol = 1e-6 ;
it = 0 ; center = (b + a)/2 ;  x = [a, center, b] ;
fun = @(x) x + exp(x) + 10./(1 + x.^2) - 5 ;
```

```
8
9  while it < itmax
10
11     if fun(a)*fun(center) < 0
12
13      b = x(2) ; a = x(1) ; center = (a + b)/2 ; x = [a, center, b] ;
14
15     elseif fun(center)*fun(b) < 0
16
17      b = x(3) ; a = x(2) ; center = (a + b)/2 ; x = [a, center, b] ;
18
19     end
20
21     if abs(fun(center)) < tol
22
23      sol = center ;
24        break
25     end
26
27     if it == itmax & abs(fun(center)) > tol
28
29      disp('PAS DE CONVERGENCE POUR LA TOLERANCE CONSIDEREE')
30
31     end
32
33      it = it + 1;
34      xit(it) = center ;
35      itNumber = it ;
36 end
37
38 disp(strcat('LA RACINE APPROCHEE VAUT :', num2str(sol)))
39 %%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE GRAPHIQUE %%%%%%%%%%%%%%%%%%%%%%%
40  ainit = -1.3 ; binit = 3/2 ; n = 500 ; dx = (binit - ainit)/n ;
41  xn = ainit :dx: binit ;
42
43 figure('color', [1 1 1])  ; plot(xn,fun(xn),'LineWidth',1) ; hold on
44 plot(sol,fun(sol) ,'ro','MarkerSize',12,'LineWidth',2)
45 plot(sol,fun(sol) ,'rx','MarkerSize',12,'LineWidth',2)
46 plot(xit,fun(xit),'kx','MarkerSize',10,'LineWidth',1.5)
47 plot(xit,fun(xit),'ko','MarkerSize',10,'LineWidth',1.5)
48
49 line(xn, zeros(1, length(xn)),'LineStyle','-.','Color','k',...
50 'LineWidth',1) ; xlabel('x') ; ylabel('f(x)')
51 title('RACINE DE f(x) = 0 PAR LA METHODE DE DICHOTOMIE')
```

Les sorties renvoyées par ce script MATLAB sont :

```
LA RACINE APPROCHEE VAUT :-0.92045
itNumber = 21
```

Sur ce graphique, les étiquettes noires représentent les approximations successives `xit` et l'étiquette rouge c'est la racine calculée `sol` par cette méthode numérique.

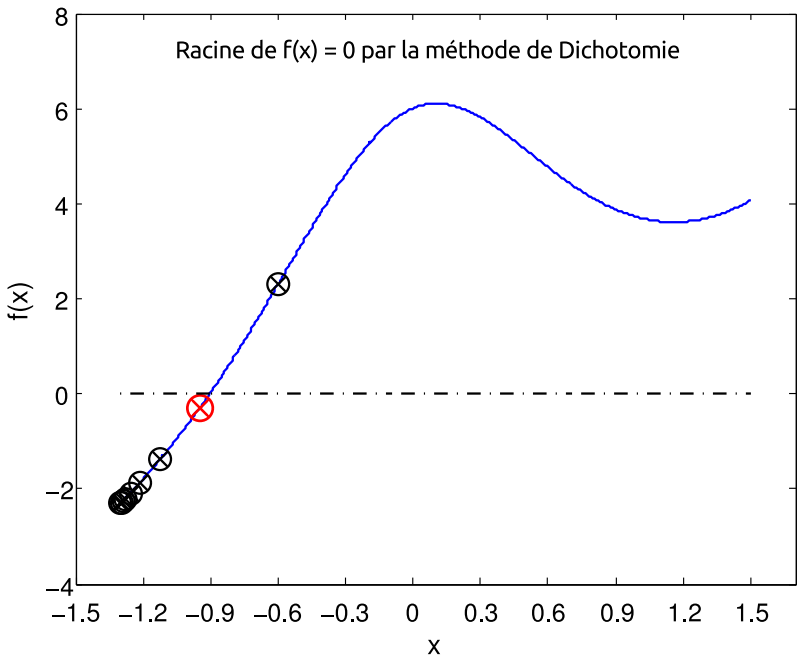


FIGURE 5: Racine de f par la méthode de `dichotomie`

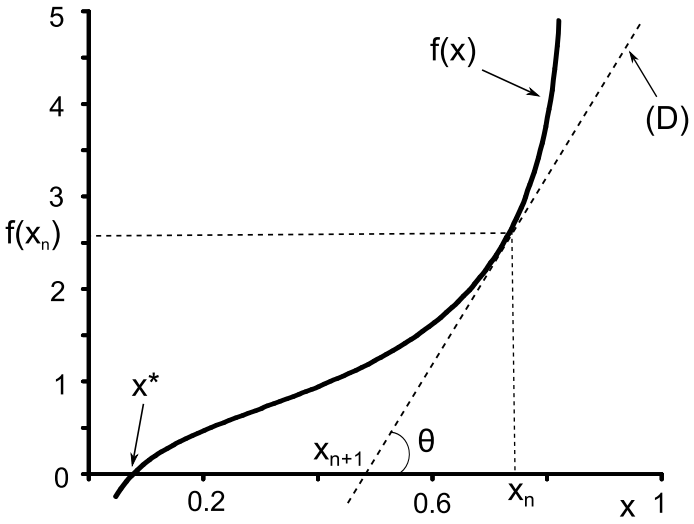
La méthode de dichotomie est simple à mettre en œuvre. Néanmoins, elle souffre d'un inconvénient du fait que son seul critère d'arrêt consiste à contrôler la longueur de l'intervalle I_n à chaque itération. Ceci risque de rejeter la racine recherchée, car elle ne tient pas suffisamment en compte du comportement effectif de la fonction f en question.

3. Résolution d'équations non-linéaires

Méthodes de Newton et de la sécante

3.1 Méthode de Newton

Comme il a été montré précédemment, la méthode de dichotomie exploite uniquement le signe de la fonction f aux extrémités des sous-intervalles. Lorsque cette fonction est différentiable, on peut établir une méthode plus efficiente en exploitant les valeurs de la fonction f et de ses dérivées. Le but de ce TP est la programmation, sous Matlab, de la méthode itérative de `Newton`. Afin d'appréhender cette dernière, soit la figure ci-dessous :



Géométriquement, la solution approchée x_{n+1} n'est autre que le point d'intersection de l'axe des abscisses et la tangente, au point $(x_n,\ f(x_n))$, d'équation D : $y = f'(x_n) \times (x_n - x_{n+1}) + f(x_n)$. Notons que x^* est la véritable racine de l'équation $f(x) = 0$, dont on cherche à approcher. À partir de la figure ci-dessus, on a :

$$\tan\theta = \frac{f(x_n)}{x_n - x_{n+1}} \tag{15}$$

Or on sait que :

$$f'(x_n) = \lim_{(x_n - x_{n+1}) \to 0} \frac{f(x_n) - 0}{x_n - x_{n+1}} = \tan\theta \tag{16}$$

À partir des Eqs. (15) et (16) on obtient ainsi le schéma numérique de la méthode de `Newton`, soit :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{17}$$

Tenant compte de toutes les méthodes vues jusqu'à présent, on constate que la méthode de Newton nécessite à chaque itération l'évaluation des deux fonctions f et de sa dérivée. Néanmoins, cet effort est compensé par une vitesse de convergence accrue, puisque cette méthode est d'ordre deux. Cet accroissement de la vitesse de convergence est conditionné par le choix de la valeur initiale qui doit être la proche possible du zéro recherché.

Énoncé du TP ❺ ☞ ®

Nous allons résoudre l'équation : $f(x) = x + exp(x) + 1$. Nous choisissons $x_0 = -1/2$ comme valeur initiale. Écrire un code matlab, portant sur l'implémentation de la méthode de `Newton`, en suivant les étapes suivantes :

1. Faire un test si $f'(x) = 0 \Longrightarrow$ arrêt du programme.
2. Le critère d'arrêt est : $|x_{n+1} - x_n| < \varepsilon$, x_n étant la solution approchée et ε, la tolérance considérée.
3. Afficher la solution approchée x_n.
4. Afficher le nombre d'itérations conduisant à la solution approchée.
5. Afficher sur le même graphe, la fonction $f(x)$, la solution approchée x_n et la droite tangente au point $(x_n, f(x_n))$.

Appliquez le même algorithme pour résoudre l'équation : $f(x) = 8x^3 - 12x^2 + 1$

✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎ Script Matlab ✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 09/11/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DE NEWTON

Nmax = 100; x = -1/2; it = 0; tol = 1e-04;
verif = tol + 1/2;

```

```
while (it < Nmax & verif >= tol)

 fx   = inline('x +exp(x) + 1');
 dfx = inline('1 + exp(x)');
 fx = feval(fx, x);
 dfx = feval(dfx,x);

if dfx ~= 0
 xn = x - (fx/dfx);
 verif = abs(fx/dfx);
 x = xn;
elseif dfx == 0
 disp('PAS DE SOLUTION, LA DERIVEE EST NULLE')
end

if (it == Nmax & verif > tol)
  disp('LA METHODE DE NEWTON NE CONVERGE PAS POUR LA TOLERANCE
   CONSIDEREE')
end

  it = it +1;
end

disp(strcat('CONVERGENCE, LA SOLUTION APPROCHEE EST xn = ', num2str(
   xn)))
disp(strcat('LE NOMBRE D''ITERATION EST = ', num2str(it)))
%%%%%%%%%%%%%%%%%%%%%%%%% AFFICHAGE GRAPHIQUE %%%%%%%%%%%%%%%%%%%%%%
xi = linspace(-12/2,6/2,1000);
fx   = inline('xi +exp(xi) + 1');
fxi = feval(fx, xi);
dfx = inline('1 + exp(x)'); droite = dfx(xn)*(xi - xn) + fx(xn);

figure('color',[1 1 1]) ; plot(xi,fxi,'LineWidth',2) ; hold on ;
plot(xi,droite,'r','LineWidth',1) ; hold on
plot(x,fx(x),'kx','MarkerSize',12,'LineWidth',2) ; hold on
plot(x,fx(x),'ko','MarkerSize',12,'LineWidth',2)

xlabel('x','fontweight','b','fontsize',12,'LineWidth',1)
ylabel('f(x)','fontweight','b','fontsize',12,'LineWidth',1)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

text('Interpreter','latex','String','$f(x) = x + exp(x) + 1 $',...
 'Position',[-10/2 20/2],'FontSize',15)
```

```
51
52 text(xn,2*xn,['x_n = ',num2str(xn)],'Position',[-3 2],...
53  'BackgroundColor',[1 1 1]);
```

Les résultats des différents calculs sont portés sur la figure suivante

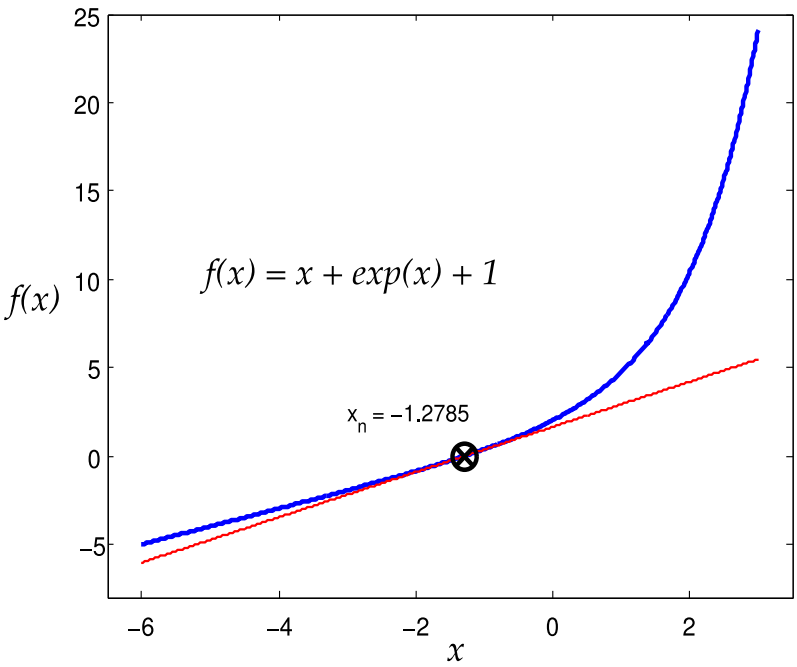


FIGURE 6: Racine de f par la méthode de `Newton`

Notons que la méthode de `Newton` converge de façon quadratique uniquement dans le cas où la racine recherchée de f est simple. Dans le cas contraire, elle converge de façon linéaire. Par ailleurs, cette méthode, peut être utilisée afin de résoudre des systèmes d'équations non linéaires.

3.2 Méthode de la sécante

La méthode de `Newton` est rapide et très utilisée, quand sa dérivée existe. Malheureusement dans certaines situations nous n'avons pas accès à cette dérivée. C'est pourquoi on s'est proposé d'étudier une autre méthode, dite de la `sécante`. Cette dernière s'affranchit de la dérivée de la fonction $f(x)$, en l'approchant par l'accroissement :

$$f^{'}(x^{k}) \approx \frac{f^{(k)} - f^{(k-1)}}{x^{(k)} - x^{(k-1)}} \tag{18}$$

Le schéma numérique de cette méthode est donné par :

$$x^{(k+1)} = x^{(k)} - f^{(k)} \times \frac{x^{(k)} - x^{(k-1)}}{f^{(k)} - f^{(k-1)}} \tag{19}$$

En observant l'équation (19), on constate des similitudes avec celle de `Newton`. Toutefois, la méthode de la `sécante` nécessite l'initialisation de deux valeurs approchées x_0 et x_1 de la racine exacte de l'équation $f(x) = 0$.

Énoncé du TP ❻ ☞ ®

Il vous est demandé de trouver la racine de l'équation : $f(x) = x - 0.2 \times sin(4x) - 1/2$. Prenez $x_0 = -1$ et $x_1 = 2$ comme valeurs initiales. Écrire un code matlab, portant sur l'implémentation de la méthode de la `sécante` en considérant une tolérance : $tol = 10^{-10}$.

1. Afficher la solution approchée x_n.
2. Afficher le nombre d'itérations conduisant à la solution approchée.
3. Afficher sur le même graphe, la fonction $f(x)$, la solution approchée et les approximations successives $x^{(k+1)}$.

✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎ Script Matlab ✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 10/11/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DE LA SECANTE

x(1) = -1 ; x(2) = 2; fun = x - 0.2.*sin(4.*x) - 1/2;
it = 0 ; tol = 1e-10 ; Nit = 20 ;

while it <  Nit

    it = it + 1;

```

```
    fun = eval('fun',x);
    var = x(2) - fun(2)*((x(2) - x(1))/(fun(2) - fun(1)));

    xn = var;
    x = [x(2), xn];

    if abs(xn -var) < tol

        sol = xn;
        approx(it) = xn;

        break
    end

end

fun = inline('x - 0.2.*sin(4.*x) - 1/2'); outPut = [sol, fun(sol)];
format long
%%%%%%%%%%%%%%%%%%%%%%% affichage graphique %%%%%%%%%%%%%%%%%%%%%%%

b = 4 ; a = -2 ; n = 1000 ; dx = (b - a)/n ; x = a:dx:b ;

figure('color', [1 1 1])
plot(x,fun(x),'LineWidth',1) ; hold on
plot(outPut(1),outPut(2),'ro','MarkerSize',12,'LineWidth',2) ;
hold on ; plot(outPut(1),outPut(2),'rx','MarkerSize',12,'LineWidth'
   ,2)
hold on ; plot(approx,fun(approx),'kx','MarkerSize',10,'LineWidth'
   ,1.5)
hold on ; plot(approx,fun(approx),'ko','MarkerSize',10,'LineWidth'
   ,1.5)
hold on ;

line(x, zeros(1, length(x)),'LineStyle','-.','Color','k','LineWidth'
   ,1)
xlabel('x') ; ylabel('f(x)') ;
title('RACINE DE f(x) PAR LA METHODE DE LA SECANTE')

%%%%%%%%%%%%%%%%%%%%% AFFICHAGE SUR LA FIGURE %%%%%%%%%%%%%%%%%%%%%
text('Interpreter','latex','String','$x - \: 0.2.*sin(4.*x) - \: 1/2
   $','Position',[-3/2 2],'FontSize',15)

text(sol,2*sol,['sol = ',num2str(sol)],'Position',[-1/2 1/2],...
 'BackgroundColor',[1 1 1]);
```

Les résultats des différents calculs sont portés sur la figure suivante

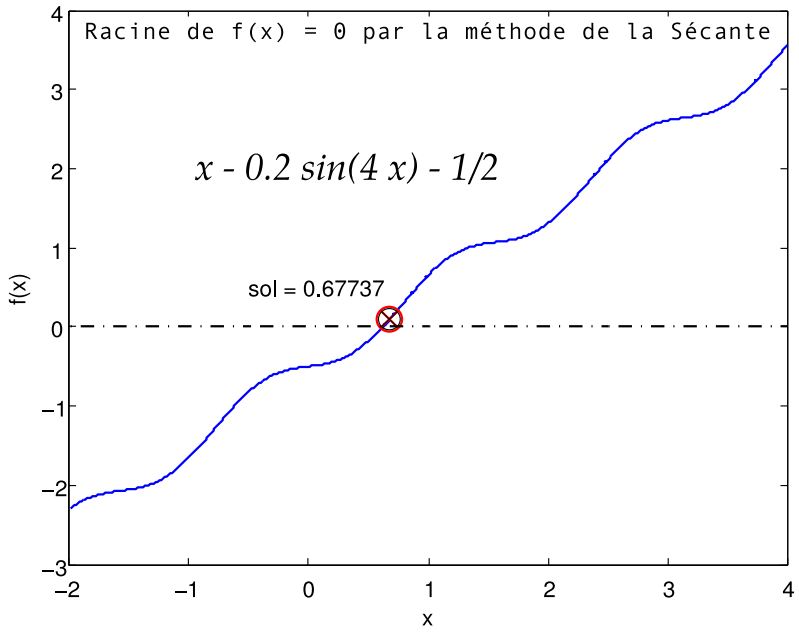


FIGURE 7: Racine de f par la méthode de la `Sécante`

4. Méthodes d'interpolation

Méthode de Lagrange

Introduction

Soient $n+1$ couples de données $\{x_i, y_i\}$, avec des nœuds différents x_i. On peut associer (relier) à ces données un seul et unique polynôme d'interpolation des y_i aux nœuds x_i, ayant un degré inférieur ou égal à n. Dans le cas général ce polynôme est donné par :

$$P_n(x) = \sum_{i=0}^{n} f(x_i)\, L_i(x), \quad \text{avec} \quad L_i(x) = \prod_{\substack{i=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \tag{20}$$

Cette relation est appelée formule d'interpolation de `Lagrange` et les polynômes L_i sont ses polynômes caractéristiques. Le but de l'interpolation consiste, entre autre, à substituer une fonction $f(x)$ (connue analytiquement ou non) par une fonction plus simple afin de procéder à une intégration numérique ou à un calcul de la dérivée. L'interpolation sert aussi à construire une représentation synthétique de données expérimentales quand leurs nombre devient très élevé.

Énoncé du TP ❼ ☞ Ⓡ

Construire, selon la méthode de `Lagrange`, le polynôme d'interpolation $P_2(x)$ de degré deux qui interpole les points : $(x_0, y_0) = (0, 1)$; $(x_1, y_1) = (1, 2)$ et $(x_2, y_2) = (2, 5)$.

1. Déterminer d'abord ce polynôme de façon analytique.
2. Écrire un algorithme Matlab permettant l'implémentation de la méthode de `Lagrange`. Déterminer ce polynôme.
3. Tracer, sur la même figure, le polynôme et les points d'interpolation.

Résolution analytique

Dans un premier temps, nous déterminerons analytiquement le polynôme $P_2(x)$. Ainsi, tenant compte de l'équation (20) et pour $n = 0, 1, 2$, il vient :

$$P_2(x) = y_0 \times L_0(x) + y_1 \times L_1(x) + y_2 \times L_2(x) \tag{21}$$

$$L_0(x) = \frac{x - x_1}{x_0 - x_1} \times \frac{x - x_2}{x_0 - x_2} \tag{22}$$

$$L_1(x) = \frac{x - x_0}{x_1 - x_0} \times \frac{x - x_2}{x_1 - x_2} \tag{23}$$

$$L_2(x) = \frac{x - x_0}{x_2 - x_0} \times \frac{x - x_1}{x_2 - x_1} \tag{24}$$

En substituant les équations (22), (23) et (24) dans (21), on obtient :

$$P_2(x) = y_0 \times \left(\frac{x - x_1}{x_0 - x_1} \times \frac{x - x_2}{x_0 - x_2}\right) + y_1 \times \left(\frac{x - x_0}{x_1 - x_0} \times \frac{x - x_2}{x_1 - x_2}\right) + y_2 \times \left(\frac{x - x_0}{x_2 - x_0} \times \frac{x - x_1}{x_2 - x_1}\right)$$

$$\Longrightarrow P_2(x) = 1 \times \left(\frac{(x-1)(x-2)}{(-1)(-2)}\right) + 2 \times \left(\frac{(x)(x-2)}{(1)(-1)}\right) + 5 \times \left(\frac{(x)(x-1)}{(2)(1)}\right)$$

$$\Longrightarrow P_2(x) = \left(\frac{(x^2 - 3\,x + 2)}{2}\right) - 2 \times \left(\frac{(x^2 - 2\,x)}{1}\right) + 5 \times \left(\frac{(x^2 - x)}{2}\right)$$

$$\Longrightarrow P_2(x) = \left(\frac{x^2 - \cancel{3x} + 2 - 4\,x^2 + \cancel{8x} + 5\,x^2 - \cancel{5x}}{2}\right)$$

$$\Longrightarrow P_2(x) = x^2 + 1 \tag{25}$$

Résolution algorithmique

À partir de ce calcul, on comprend que le calcul analytique montre ses limites pour des degrés n élevés. Par conséquent, il est nécessaire de développer un algorithme permettant l'implémentation de la méthode de `Lagrange` afin de déterminer les polynômes quelque que soit le degré n.

Script Matlab

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 02/12/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DE LAGRANGE POUR
% L'INTERPOLATION POLYNOMIALE
x(1) = 0 ; x(2) = 1 ; x(3) = 2; n = 3 ; % POLYNOME DE DEGRE 2
y(1) = 1 ; y(2) = 2 ; y(3) = 5; interv = 1000; dx = (x(3) - x(1))/
   interv ;
xvar = x(1) :dx: x(3) ; polyn = 0 ;

col = {'+k','+r','+m'};

for i = 1:n

    lag = 1;

    for j = 1 : n

        if (i~=j)

            lag = (xvar - x(j))./(x(i) - x(j)).*lag;

        end
    end

 figure(1) ; plot(x(i),y(i),col{i},'MarkerSize',12,'LineWidth',2);
   hold on ;
 polyn = polyn + lag.*y(i);
end

hold on
plot(xvar, polyn,'LineWidth',1) ; hold on ; xlabel('x') ; ylabel('y')
axis([-0.5 2.5 -0.5 5.5])

title('Interpolation : selon Lagrange')

p = 2 ; coeff = polyfit(xvar,polyn,p) % COEFFICIENTS DU POLYNOME D'
   INTERPOLATION. AINSI, % coeff(1) CONTIENT LE COEFFICIENT
   xpuissance(p) , coeff(2) CELUI DE xpuissance(p-1), ... % C'EST-A-
   DIRE, DANS NOTRE CAS coeff(1)*xpuissance(2) + coeff(2)*x + coeff
   (1) = xpuissance(2) + 1.
```

Ci-dessous, le graphique représentant le polynôme interpolant et les points d'interpolation.

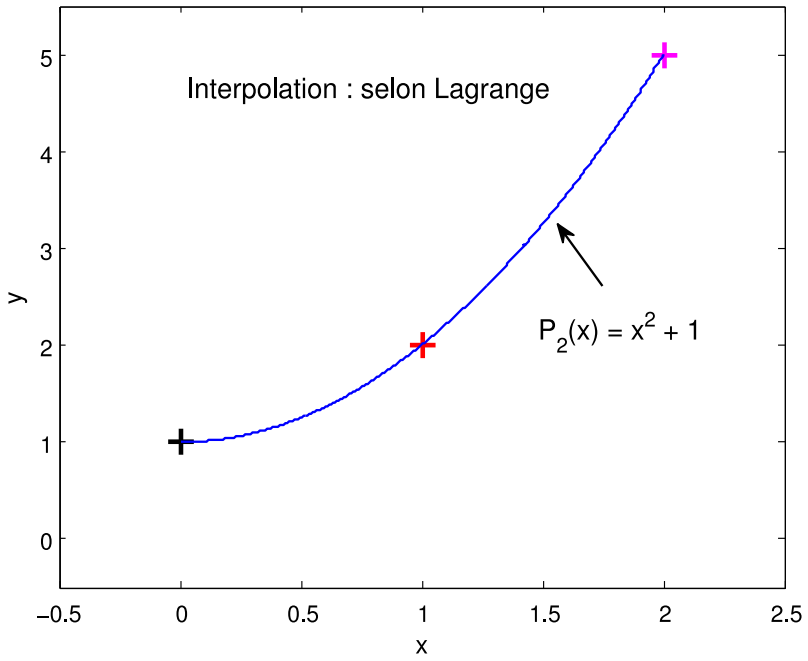


FIGURE 8: Interpolation selon `Lagrange`

5. Méthodes d'interpolation

Méthode de Hermite

Introduction

L'interpolation de `Hermite` est une généralisation de celle de `Lagrange` en faisant coïncider non seulement $f(x)$ et P_n aux nœuds x_i, mais également leurs dérivées d'ordre k_i aux nœuds x_i. Soient $x_0, x_1, ..., x_n$, (n + 1) points distincts de l'intervalle $[a\ b]$ et $y = f(x)$ une fonction définie sur le même intervalle admettant les dérivées $(y_0, y_0^{'}),\ (y_1, y_1^{'}), ...,\ (y_k, y_k^{'})$. Dans ce cas, il existe un seul et unique polynôme tel que $\prod_N(x_i) = y_i$ et $\prod_N^{'}(x_i) = y_i^{'}$, avec $N = 2n + 1$ et $i = 0, 1, 2, ..., n$. Le polynôme de `Hermite` est donné par :

$$P_n\,(x) = \sum_{i=0}^{n} \Big(y_i\, D_i(x)\ +\ y_i^{'}\,(x - x_i) \Big) \times (L_i(x))^2 \tag{26}$$

Avec,

$$L_i(x) = \prod_{\substack{i=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j} \qquad c_i = \sum_{\substack{i=0 \\ j \neq i}}^{n} \frac{1}{x_i - x_j} \qquad D_i(x) = 1 - 2(x - x_i) \times c_i \tag{27}$$

Signalons au passage que l'interpolation de `Lagrange` (voir le TP précédent) est un cas particulier de celle de `Hermite` ($k_0 = k_1 = ... = k_n = 0$).

Énoncé du TP ❽ ☞ ®

Construire, selon la méthode de `Hermite`, le polynôme d'interpolation $P_2(x)$ de degré deux qui interpole les points : $(x_0, y_0) = (0, 1)$; $(x_1, y_1) = (1, 2)$ et $(x_2, y_2) = (2, 5)$.

1. Écrire un algorithme Matlab permettant l'implémentation de la méthode de `Hermite`. Déterminer ce polynôme.
2. Tracer, sur la même figure, le polynôme et les points d'interpolation.

Script Matlab

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 03/12/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DE HERMITE POUR
% L'INTERPOLATION

x = [0, 1 , 2] ; n = 3 ; % POLYNOME DE DEGRE n - 1.
y = [1 , 2, 5] ; interv = 1000 ; dx = (x(3) - x(1))/interv ;
yder = [y(1), (y(2)-y(1))/(x(2)-x(1)), (y(3)-y(2))/(x(3)-x(2))];

xvar = x(1) : dx : x(3) ; polyn = 0 ;
col = {'+k','+r','+m'};

for i = 1:n
    lag = 1;
    ci = 0;

    for j = 1 : n

        if (i~=j)

            lag = (xvar - x(j))./(x(i) - x(j)).*lag;
            ci = ci + (1/(x(i) - x(j)));
       end
    end

Di = 1 - 2.*(xvar - x(i)).*ci;
polyn = polyn + (y(i).*Di + yder(i) .*(xvar - x(i))).*(lag).^2;
figure(2) ; plot(x(i),y(i),col{i},'MarkerSize',12,'LineWidth',2);
hold on ;
end

hold on
plot(xvar, polyn,'b','LineWidth',1) ; hold on ; xlabel('x') ; ylabel(
    'y')
axis([-0.5 2.5 -0.5 5.5])
title('Interpolation : selon Hermite')

p = 2 ;  coeff = polyfit(xvar,polyn,p) ; % EVALUATION DES
    COEFFICIENTS
evalp = polyval(coeff, xvar) ; % EVALUATION DU POLYNOME
```

Ci-dessous, le graphique représentant le polynôme interpolant et les points d'interpolation.

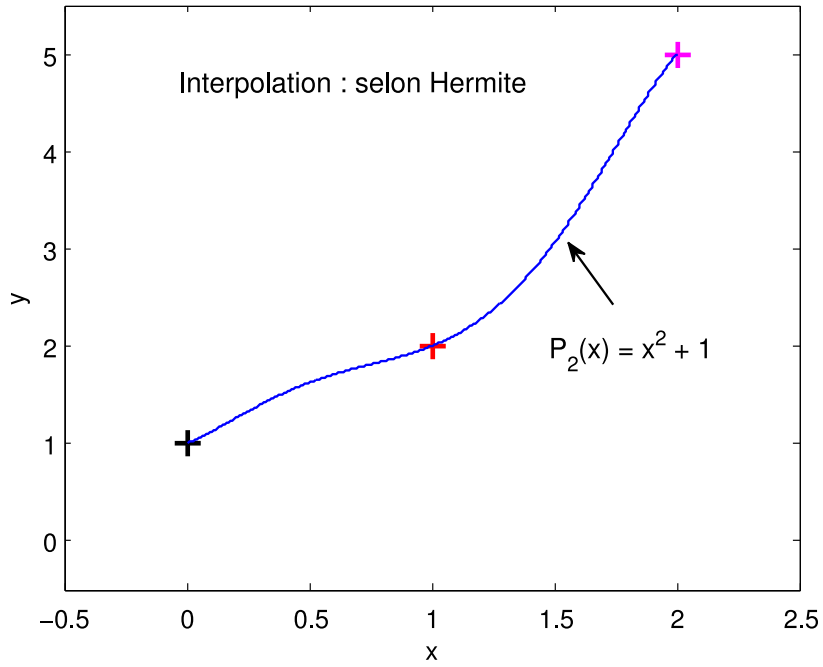


FIGURE 9: Interpolation selon `Hermite`

6. Méthodes d'interpolation

nœuds de Tchebychev

Introduction

Nous avons vu précédemment, dans le cas de l'interpolation de `Lagrange` et celle de `Hermite`, que l'utilisateur a la liberté de choisir les nœuds d'interpolation $\{x_i\}$. En revanche, l'interpolation de `Pafnouti Tchebychev`, impose un choix des nœuds (voir, Eqs. (29) ou (30)) dans l'intervalle $[a, b]$, appelés points de `Tchebychev`. Il s'agit donc d'une interpolation de `Lagrange` menée en des points particuliers. Tenant compte de cette répartition particulière des nœuds, on peut montrer que, si $f(x)$ est dérivable sur $[a, b]$, alors $P_n(x)$ converge vers $f(x)$ quand $n \longrightarrow \infty$ pour tout $x \in [a, b]$. Ainsi, les nœuds de `Tchebychev` sont déterminés par minimisation de la quantité :

$$max_{x\in[a,b]} \left| \prod_{i=0}^{n} (x - x_i) \right| \tag{28}$$

En effet, l'équation (28) est minimale pour les nœuds qui annulent les polynômes de `Tchebychev`, soit :

$$x_i = \left(\frac{b+a}{2}\right) + \left(\frac{b-a}{2}\right) \times \cos\left(\frac{(2\,(n-i)+1)\,\pi}{2\,n}\right) \quad \text{avec} \quad i = 0, 1, ..., n \tag{29}$$

Il existe également une autre répartition non uniforme des nœuds d'interpolation, ayant les mêmes caractéristiques de convergence que les nœuds de `Tchebychev`, définie par les nœuds de `Tchebychev-Gauss`, soit :

$$x_i = \left(\frac{b+a}{2}\right) - \left(\frac{b-a}{2}\right) \times \cos\left(\frac{(2\,i+1)\,\pi}{(n+1)\,2}\right) \quad \text{avec} \quad i = 0, 1, ..., n \tag{30}$$

L'attrait de choisir les nœuds de `Tchebychev`, est de se prémunir (lutter) contre le `phénomène de Runge`. Dans ce qui suit nous nous attellerons de rappeler ce phénomène.

Interprétation

Intuitivement on est tenté de croire que plus l'écart entre les points d'interpolation est réduit meilleure sera la convergence du polynôme de Lagrange. Les choses ne se présentent pas sur cet aspect, ainsi, Carl Runge a mis en évidence le fait que plus $n \longrightarrow \infty$ plus le polynôme de Lagrange diverge aux bords de l'intervalle $[a, b]$. On comprend dans ce cas, que la convergence n'est pas régulière. Cette irrégularité de convergence est résolue en procédant à l'interpolation aux points de Tchebychev. Nous allons illustrer ce phénomène de Runge par l'example suivant :

Soit la fonction $f(x) = \dfrac{1}{1+x^2}$, avec $x \in [-5, 5]$. Nous allons considérer les nœuds d'interpolation suivants : $x_0 = -5$, $x_1 = -4$, $x_2 = -3$, $x_3 = -2$, $x_4 = -1$, $x_5 = 0$, $x_6 = 1$, $x_7 = 2$, $x_8 = 3$, $x_9 = 4$, $x_{10} = 5$.

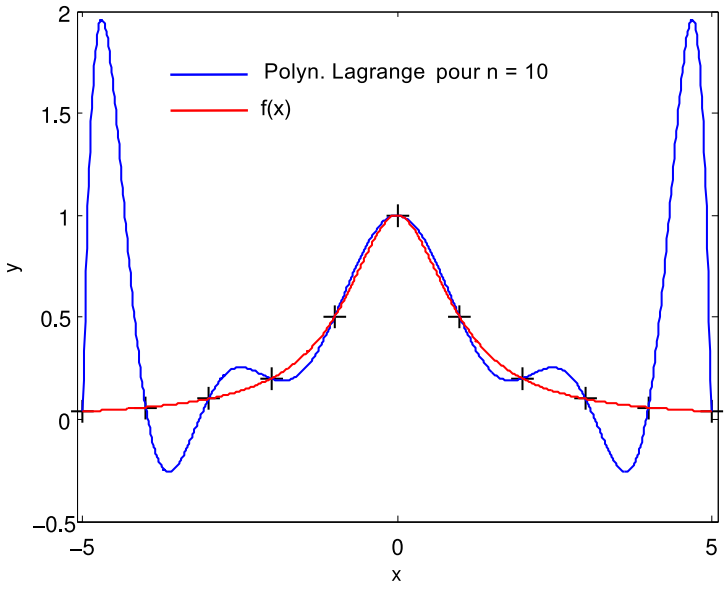


FIGURE 10: Illustration du phénomène de Runge

À partir de la fig. (10), on constate une divergence, au voisinage des bornes de l'intervalle, entre le polynôme interpolateur de Lagrange et la courbe $f(x)$. Par ailleurs, plus le degré n de ce polynôme est élevé plus la divergence (oscillations sur les bords de $[a, b]$) aux bords de l'intervalle est grande. Ceci est typique d'une convergence irrégulière. Cet exemple illustre très bien le phénomène de Runge. Ceci peut être démontré en calculant le maximum de $f(x)$ et de ses dérivées jusqu'à l'ordre 25 avec le code Matlab suivant :

```
clear all ; clc ; format long ;
xvar = -5:0.01:5 ; syms x ; n = 24 ; fun1 = 1/(1+ x.^2 ) ;
for ik = 1:n + 1
dfun1 = diff(fun1, ik) ; fun = subs(dfun1, xvar) ;
maxdfun1(ik) = max(abs(fun)) ; end
```

Après exécution, les maximums des valeurs absolues de $f(x)^{(k)}$ pour k = 22 à 25 sont :

```
maxdfun1([22, 23, 24, 25]) = 1.0e+25 *
0.00011240007277 0.00245438057646 0.06204484017332 1.48020670296177
```

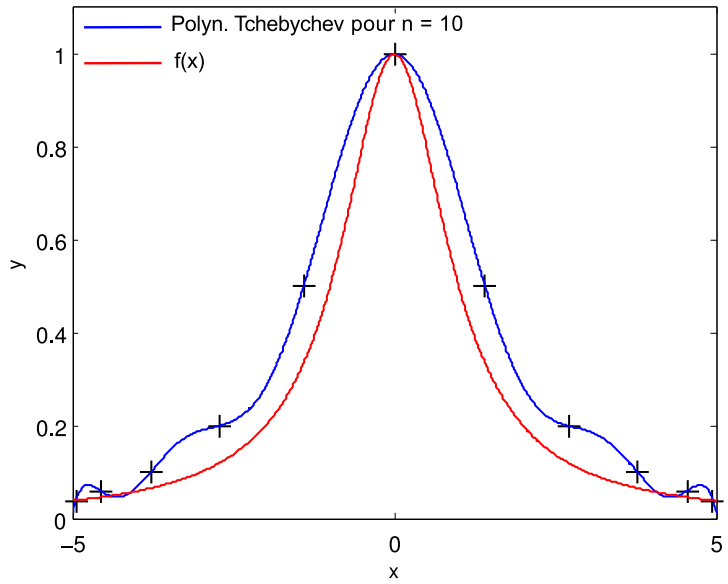


FIGURE 11: Atténuation du phénomène de `Runge` aux nœuds de `Tchebychev`

En revanche, dans l'interpolation de `Tchebychev` (fig. (11)), la convergence est uniforme. De plus, lorsque le nombre de nœuds d'interpolation augmente, la courbe polynomiale se confond avec la fonction (fig. (12)).

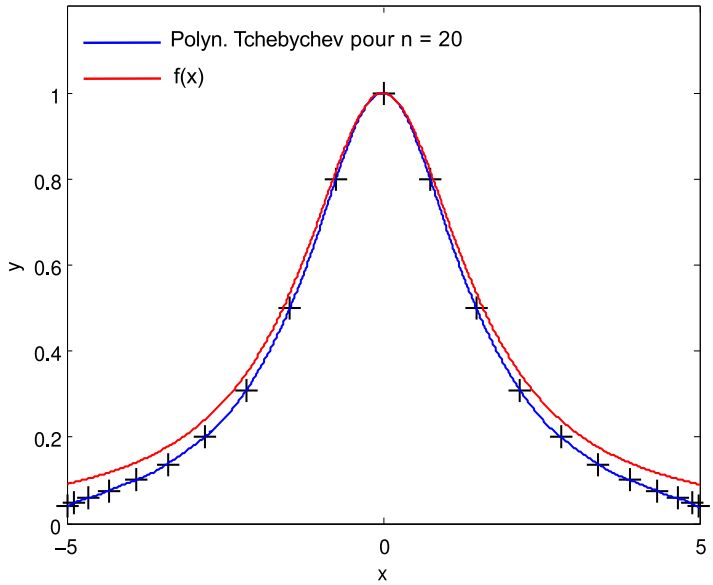


FIGURE 12: Effet du nombre de points d'interpolation selon `Tchebychev`

Énoncé du TP 9 ☞ ®

Construire, sous Matlab, le polynôme d'interpolation de Tchebychev de la fonction $f(x) = \dfrac{1}{1+x^2}$. Les nœuds d'interpolation sont les suivants : $x_0 = -5$, $x_1 = -4$, $x_2 = -3$, $x_3 = -2$, $x_4 = -1$, $x_5 = 0$, $x_6 = 1$, $x_7 = 2$, $x_8 = 3$, $x_9 = 4$, $x_{10} = 5$.

1. Déterminer ce polynôme pour $n = 10$ et $n = 20$. Conclure
2. Afficher les nœuds de Tchebychev
3. Tracer, sur la même figure, le polynôme de Tchebychev, les points d'interpolation et la fonction $f(x)$.

Script Matlab

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 03/12/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE LA METHODE DE TCHEBYCHEV
format long

x(1) = -5 ; x(2) = -4 ; x(3) = -3; n = 11 ; %POLYNOME DE DCEGRE n - 1
x(4) = -2 ; x(5) = - 1 ; x(6) = 0 ; x(7) = 1 ; x(8) = 2 ; x(9) = 3 ;
   x(10) = 4 ; x(11) = 5 ;  y = 1./(1 + x.^2);

interv = 1000; dx = (x(11) - x(1))/interv ;
xvar = x(1) :dx: x(11) ; polyn = 0 ;

x=(x(11)+x(1))/2+((x(11)-x(1))/2)*cos((2*(n-(1:n))+1)*pi/(2*n));
xChe =  x; % LES NOEUDS xi ANNULANT LES POLYNOMES DE TCHEBYCHEV :
   CECI AFIN DE MINIMISER L'ECART ENTRE Pn(x) ET LA FONCTION f(x) A
   INTERPOLER

for i = 1:n

    lag = 1;

    for j = 1 : n

        if (i~=j)

          lag = (xvar - xChe(j))./(xChe(i) - xChe(j)).*lag;

        end
```

```
27      end
28
29   figure(1) ; plot(xChe(i),y(i),'+k','MarkerSize',12,'LineWidth',1);
       hold on ;
30   polyn = polyn + lag.*y(i);
31  end
32
33  hold on
34  plot(xvar, polyn,'b','LineWidth',1) ; hold on ; xlabel('x') ; ylabel(
       'y')
35  title('Interpolation : selon Lagrange')
36  p = 10 ; coeff = polyfit(xvar,polyn,p) ;
37
38   evalp = polyval(coeff, xvar) ; hold on ;
39  plot(xvar, 1./(1 +xvar.^2),'r')
```

Les graphiques générés par ce script Matlab, sont représentés ci-dessous

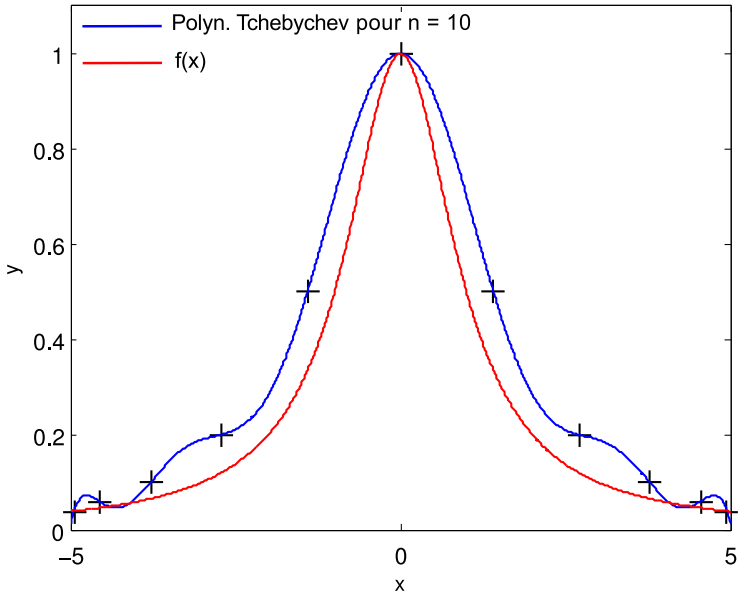


FIGURE 13: Interpolation aux nœuds de `Tchebychev` pour $n = 10$

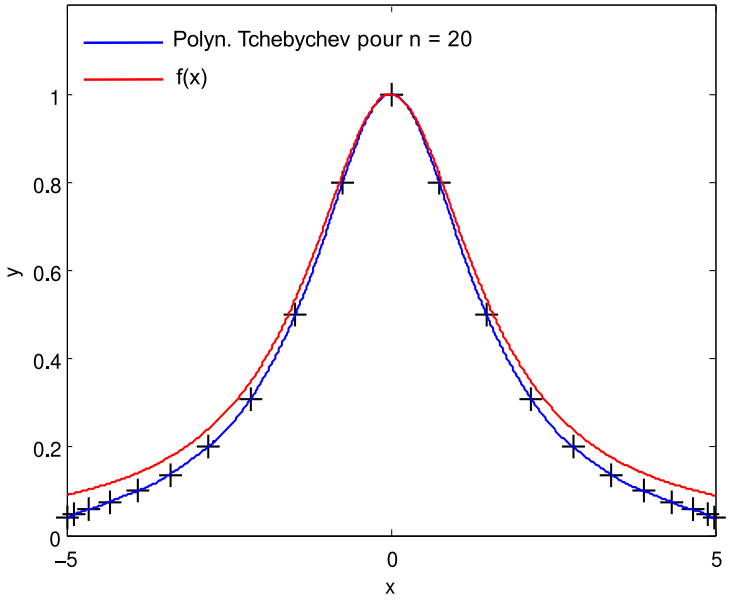


FIGURE 14: Interpolation aux nœuds de `Tchebychev` pour $n = 20$

7. Résolution numérique d'équations différentielles

Problème de Cauchy

Introduction

On désire calculer la solution $y(t)$ sur l'intervalle $I = [a, b]$ du problème de Cauchy

$$\dot{y}(t) = f(t, y(t)) \quad \text{avec} \quad y(t_0) = y_0 \tag{31}$$

On comprend, ainsi, qu'une équation différentielle est une équation dépendant d'une variable t et d'une fonction $y(t)$ et qui contient des dérivées de $y(t)$. Elle peut s'écrire comme :

$$F\left(t, y(t), y^{(1)}(t), y^{(2)}(t), ..., y^{(k)}(t)\right) = 0 \iff y^{(k)}(t) = \frac{d^k y(t)}{dt^k} \tag{32}$$

Pour $k = 2$, l'équation ci-dessus peut se mettre sous la forme différentielle :

$$\frac{d}{dt}\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = f(t, y(t)) = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{33}$$

L'existence d'une solution unique de l'équation différentielle est tributaire de l'imposition de certaines conditions limites sur $y(t)$ et de ses dérivées. Dans l'équation (32), les conditions initiales sont les valeurs de $y(a), y^{(1)}(a), y^{(2)}(a), ..., y^{(k-1)}(a)$. D'un point de vue pratique, l'inconnue $y(t)$ est un vecteur ayant k fonctions inconnues avec pour conditions initiales le vecteur $y_k(a)$ (ou $y_k(t_0)$). Par ailleurs, il faut noter que très souvent la solution analytique n'existe pas, et on doit par conséquent approcher la solution exacte $y(t)$ par des méthodes numériques.

7.1 Méthodes à un pas

Ces méthodes sont basées sur un développement en série de `Taylor` suivant un ordre plus au moins élevé. Elles sont qualifiées `à un pas`, car le calcul de y_{k+1} ne réclame que la valeur de y_k à l'instant précédent. Une méthode à `deux pas` utilisera à la fois y_k et y_{k-1}. Les schémas numériques des méthodes d'`Euler` explicite, de `Heun` (ou de

Runge-Kutta d'ordre 2) et de Runge-Kutta classique d'ordre 4 sont donnés dans ce qui suit

7.1.1 Méthode d'Euler

Afin d'atteindre la solution $y(t)$, sur l'intervalle $t \in [a, b]$, on choisit $n+1$ points dissemblables $t_0, t_1, t_2, ..., t_n$, avec $t_0 = a$ et $t_n = b$ et le pas de discrétisation est défini par $h = (b-a)/n$. La solution à estimer peut être approchée par un développement limité de Taylor

$$y(t_k + h) = y(t_k) + \frac{dy(t_k)}{dt}(t_{k+1} - t_k) + ... \tag{34}$$

Puisque $\frac{dy(t_k)}{dt} = f(t_k, y(t_k))$ et $h = t_{k+1} - t_k$, on obtient ainsi le schéma numérique d'Euler :

$$\begin{cases} y_0 = \text{valeurs initiales} \\ y_{k+1} = y_k + h\, f(t_k, y_k), \quad \text{avec} \quad k = 0, 1, ..., n-1. \end{cases} \tag{35}$$

Cette méthode est d'ordre 1, cela veut dire que l'erreur est proportionnelle au carré du pas (h) de discrétisation. Intuitivement, on comprend que pour améliorer la précision cette méthode, il suffira de réduire h. Cette réduction du pas de discrétisation aura pour incidence l'accroissement du temps de calcul ($\sim 1/h$). Par ailleurs, l'avantage de la méthode d'Euler, tire son origine du fait qu'elle réclame uniquement l'évaluation de la fonction f pour chaque pas d'intégration.

7.1.2 Méthode de Heun

La Méthode de Heun est une version améliorée de celle d'Euler. L'erreur du résultat généré par cette méthode est proportionnelle à h^3, meilleur que celui de la méthode d'Euler. Néanmoins, la méthode de Heun réclame une double évaluation de la fonction f.

$$\begin{cases} y_0 = \text{valeurs initiales} \\ y_{k+1} = y_k + \frac{h}{2}\left(f(t_k, y_k) + f(t_{k+1}, y_k + h\, f(t_k, y_k))\right), \quad \text{avec } k = 0, 1, ..., n-1. \end{cases} \tag{36}$$

Le schéma numérique de cette méthode résulte de l'application de la formule de quadrature du trapèze. Notons également que la méthode de Heun fait partie des méthodes de Runge-Kutta explicites d'ordre deux.

7.1.3 Méthode de Runge–Kutta, d'ordre 3

Les méthodes de type Runge-Kutta permettent d'obtenir une plus grande précision (elles génèrent des solutions numériques plus proches des solutions analytiques) que les deux méthodes précédentes.

$$\begin{cases} y_0 = \text{valeurs initiales} \\ y_{k+1} = y_k + \left(f(t_k, y_k) + 4\, f(t_k + \dfrac{h}{2}, y_k + y_k \dfrac{h}{2}) + f(t_k + h, y_k + (2\, y_{2k} - y_{1k})\, h \right) \\ \text{avec } k = 0, 1, ..., n-1 \text{ et} \end{cases} \tag{37}$$

$$\begin{cases} y_{1k} = f(t_k, y_k) \\ y_{2k} = f(t_k + \frac{h}{2}, y_k + y_{1k} \frac{h}{2}) \end{cases} \tag{38}$$

C'est la méthode de Runge-Kutta explicite à trois niveaux.

7.1.4 Méthode de Runge-Kutta, d'ordre 4

La méthode de `Runge-Kutta` (classique) d'ordre 4, est une méthode explicite très populaire. Elle calcule la valeur de la fonction en quatre points intermédiaires selon :

$$\begin{cases} y_0 = \text{valeurs initiales} \\ y_{k+1} = y_k + \dfrac{h}{6} \left(f(t_k, y_{1k}) + 2\, f(t_k + \dfrac{h}{2}, y_{2k}) + 2\, f(t_k + \dfrac{h}{2}, y_{3k}) + f(t_{k+1}, y_{4k}) \right) \\ \text{avec } k = 0, 1, ..., n-1 \text{ et} \end{cases} \tag{39}$$

$$\begin{cases} y_{1k} = y_k \\ y_{2k} = y_k + \frac{h}{2}\, f(t_k, y_{1k}) \\ y_{3k} = y_k + \frac{h}{2}\, f(t_k + \frac{h}{2}, y_{2k}) \\ y_{4k} = y_k + h\, f(t_k + \frac{h}{2}, y_{3k}) \end{cases} \tag{40}$$

Notons que le nombre de termes retenus dans la série de `Taylor` définit l'ordre de la méthode de `Runge-Kutta`. Il vient que la méthode `Runge-Kutta` d'ordre 4, s'arrête au terme $O(h^4)$ de la série de `Taylor`.

Énoncé du TP ⓾ ☞ ®

1. Résoudre numériquement, par le biais des méthodes de `Euler`, de `Heun` et de `Runge-Kutta` d'ordre 4, l'équation différentielle du premier ordre suivante :

$$\begin{cases} y' = \dfrac{-y\,t^2 - y^2 + 2\,t}{1 - t^3} \\ y(0) = 1 \end{cases} \tag{41}$$

2. Afficher sur la même figure, la solution des trois méthodes.
3. Analyser l'erreur en fonction du pas de discrétisation pour la méthode de `Runge-Kutta` d'ordre 4.
4. Connaissant la solution exacte de l'équation différentielle ci-dessous. Proposer une démarche permettant la détermination de l'ordre de convergence de la méthode de `Runge-Kutta`. On donne :

$$\begin{cases} y' = \dfrac{t - y}{2} \\ y(0) = 1 \end{cases} \tag{42}$$

   La solution exacte est : $3\,exp(-t/2) + t - 2$
5. Tracer le graphique donnant l'erreur relative, pour chaque pas de discrétisation, en fonction du nombre d'itérations.
6. Tracer le graphique donnant le maximum de l'erreur relative en fonction du pas de discrétisation.

Script Matlab

```
clear all ; close all ; clc ;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% @copyright 13/12/2015 Samir KENOUCHE : ALGORITHME PERMETTANT
% L'IMPLEMENTATION, SOUS MATLAB, DE METHODES NUMERIQUES (Euler, Heun,
% Runge-Kutta) POUR LA RESOLUTION D'EQUATIONS DIFFERENTIELLES

%%%%%%%%%%%%%%%%%%% METHODE DE Runge-Kutta d'ordre 4 %%%%%%%%%%%%%%%%

dydt = inline('(-y.*t.^2 - y.^2 + 2.*t)./(1 - t.^3)','t','y');
a = 0 ; b = 1 ; n = 70 ; h = (b-a)/n ; t = a :h: b;

epsilon = 0.0001 ; u(1) = 1 + epsilon ;

for i = 1:n-1
```

```
15
16    u1(i) = u(i)  ; u2(i) = u(i) + h/2*dydt(t(i),u1(i)) ;
17
18    u3(i) = u(i) + h/2*dydt(t(i) + h/2, u2(i)) ; u4(i) = u(i) + h*dydt
      (t(i) + h/2, u3(i)) ;
19
20    u(i+1) =u(i) + h/6*(dydt(t(i),u1(i)) + 2*dydt(t(i) + h/2,...
21    u2(i)) + 2*dydt(t(i) + h/2,u3(i)) + dydt(t(i+1),u4(i))) ;
22
23  end
24
25   figure('color',[1 1 1])
26   plot(t(1:end-1),u,'o-g')
27
28  %%%%%%%%%%%%%%%%%%%%%%%% METHODE DE Heun %%%%%%%%%%%%%%%%%%%%%%%%
29  clear all ; clc ;
30
31  dydt = inline('(-y.*t.^2 - y.^2 + 2.*t)./(1 - t.^3)','t','y');
32  a = 0 ; b = 1 ; n = 70 ; h = (b-a)/n ; t = a:h:b;
33
34  epsilon = 0.0001 ; u(1) = 1 + epsilon;
35
36  for i = 1:n - 1
37
38    u(i+1) =u (i) + h/2*(dydt(t(i),u(i)) + dydt(t(i + 1),...
39       u(i) + h*dydt(t(i),u(i))));
40  end
41
42  tn = t(1:end-1) ;
43  hold on ; plot(tn,u,'o-r') ;
44  %%%%%%%%%%%%%%%%%%%%%%%% METHODE D'Euler %%%%%%%%%%%%%%%%%%%%%%%%
45
46  clear all ; clc ;
47  dydt = inline('(-y.*t.^2 - y.^2 + 2.*t)./(1 - t.^3)','t','y');
48
49  a = 0 ; b = 1 ; n = 70 ; h = (b-a)/n ; t = a:h:b;
50  epsilon = 0.0001 ; u(1) = 1 + epsilon;
51
52  for i = 1:n - 1
53
54    u(i+1) =u (i) + h/2*(dydt(t(i),u(i))) ;
55
56  end
57
```

```
58 hold on
59 plot(t(1:end-1),u,'o-b')
60
61 legend('Runge-Kutta','Heun','Euler')
```

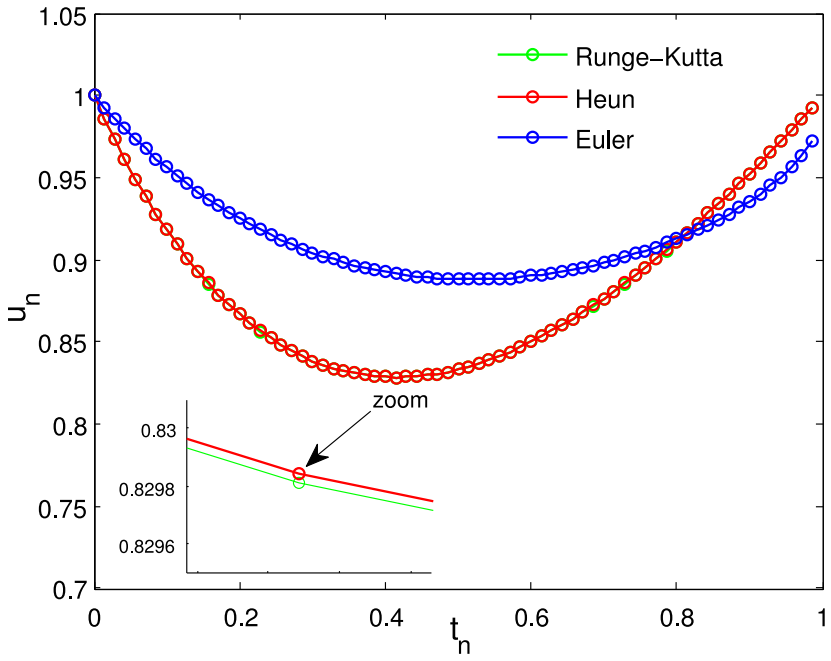


FIGURE 15: Solutions numériques obtenues par les méthodes de `Euler`, de `Heun` et de `Runge-Kutta d'odre 4`

✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎ Script Matlab ✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎✎

```
1
2  clear all ; close all ; clc ;
3  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4  % @copyright 14/12/2015 Samir KENOUCHE : ANALYSE DE L'ERREUR EN
5  % FONCTION DU PAS DE DISCRITISATION DANS LE CADRE DE LA METHODE DE
6  % Runge-Kutta D'ORDRE 4
7
8  dydt = inline('(t - y)/2','t','y');  % EQUATION DIFFERENTIELLE
9  funex = inline('3*exp(-t/2)- 2 + t','t') ; % SA SOLUTION EXACTE
10 a = 0 ; b = 6 ; ik = 0 ; epsilon = 0.0001 ; u(1) = 1  ;
11 col = {'k','b','r','m','c','g'};
12
13 for n = 60:10:100
14
15     h = (b-a)/n ; t = a :h: b;
16
17   for i = 1: n -1
```

```
18
19     u1(i) = dydt(t(i),u(i)) + 2*dydt(t(i) + h/2,...
20     u(i) + h/2*dydt(t(i),u(i))) ;
21
22     u2(i) = dydt(t(i) + h/2,u(i)+h/2*dydt(t(i)+h/2,...
23     u(i)+h/2*dydt(t(i),u(i))));
24
25     u3(i) = dydt(t(i+1), u(i) + h*dydt(t(i)+h/2,...
26     u(i)+h/2*dydt(t(i)+h/2,u(i) + h/2*dydt(t(i), u(i)))) ;
27
28     u(i+1) = u(i) + h/6*(u1(i) + 2*u2(i) + u3(i)) ;
29
30   end
31
32 ik = ik + 1;
33 t = t(1:end-1) ;
34 err  =  abs((funex(t) - u)./funex(t));
35
36 max_err(ik) = max(err) ;  pas(ik) = h ;
37 figure(1) ; hold on ; plot(err,col{ik}, 'LineWidth',1) ;
38 xlabel('NOMBRE D''ITERATIONS') ; ylabel('ERREUR RELATIVE')
39
40 end
41
42 figure(3) ; plot(t,u,'o-') ; hold on
43 plot(t, funex(t),'o-r')
44
45 for p = 1:6
46
47 [coeff, s] = polyfit(pas,max_err,p) ; % s ETANT L'ERREUR
48 % D'INTERPOLATION, POUR UN DEGRE p, GENEREE PAR LA FONCTION : polyfit
49 evalp = polyval(coeff, pas) ;
50
51 err_iterpolation(p) = max(abs(evalp - max_err));
52
53 end
54
55 [min_err_interp, degre_interp] = min(err_iterpolation);
56 coeff_opt = polyfit(pas,max_err,degre_interp);
57 eval_opt = polyval(coeff_opt, pas) ;
58
59 figure('color', [1 1 1])
60 plot(pas, max_err,'r+','MarkerSize',10,'LineWidth',1)
61 hold on ; plot(pas,eval_opt) ; axis([0.05 0.11 0 7e-08])
```

```
62 xlabel('PAS DE DISCRETISATION') ; ylabel('MAXIMUM DE L''ERREUR
      RELATIVE')
63
64 str1 = {'CETTE METHODE EST D''ORDRE :', num2str(degre_interp)}
65 uicontrol('Style','text','Position',[260 80 150 60],...
66           'BackgroundColor',[1 1 1],'String',str1);
```

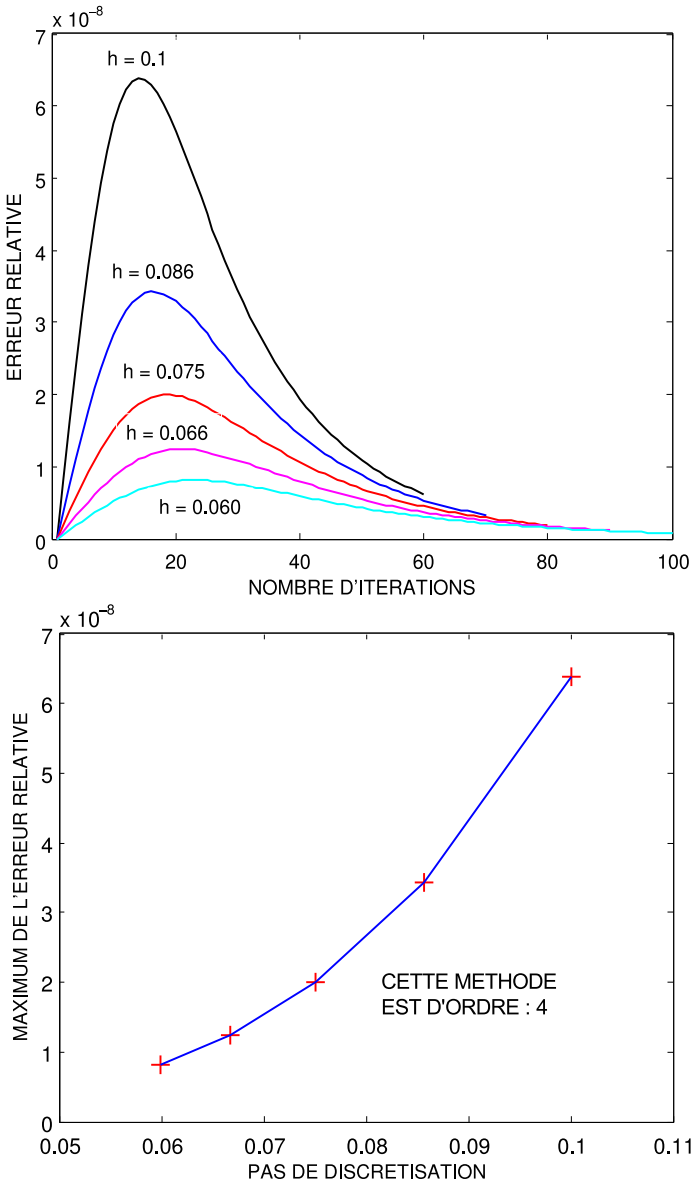


FIGURE 16: Évolution de l'erreur relative en fonction du pas de discrétisation

s(p = 1) : normr = 9.3851e-09 ; s(p = 2) : normr = 7.4149e-10 ; s(p = 3) : normr = 1.6060e-11 ; s(p = 4) : normr = 9.9262e-24 ; s(p = 5) : normr = 1.9921e-23 ; s(p = 6) : normr = 2.8892e-23.

D'où le choix `p = 4`, qu'on retrouve également avec une démarche similaire en posant `err_iterpolation = max(abs(evalp - max_err))` et `[min_err_interp, degre_interp] = min(err_iterpolation)`. Il en ressort que `degre_interp = 4`. Avec un raisonnement analogue on peut aisément obtenir l'ordre des méthodes `d'Euler` et de `Heun`.

7.2 Au moyen de commandes Matlab

Introduction

Matlab comprend un certain nombre de *solveurs* destinés à la résolution d'équations différentielles. Les plus utilisés sont `ode23` et `ode45`, qui sont basés sur la méthode de *Runge-Kutta* explicite à un pas. Le *solveur* `ode113`, utilise la méthode de *Adams-Bashforth-Moulton* multi-pas. Les autres *solveurs* sont `ode15s`, `ode23s`, `ode23t`, `ode23tb`. Ils ont tous la même syntaxe :

`[t, y] = solveur(eqs, [ti ; tf], yinit, opts)`. Cette syntaxe renvoie la solution `y` au temps `t`. L'argument `eqs` est le système d'équations différentielles. Ce système peut être défini de plusieurs manière. Soit à travers un fichier `m-file`, dans ce cas on doit rajouter l'identifiant `@eqs`. Soit à travers une commande `inline` ou bien au moyen de la *fonction anonyme*. Ce système d'équations différentielles est résolu sur l'intervalle `[ti ; tf]`, avec les conditions initiales `yinit = [y(ti) ; y(tf)]`. L'argument d'entrée `opts`, de type *srtucture*, compte les options d'optimisation indiquées dans `odeset`, sa syntaxe est donnée par :

`opts = odeset('Property1', value1, 'Property2', value2, ...)`. Ainsi, chaque propriété est suivie de sa valeur. À titre illustratif, la propriété `('Stats','on', ...)` affiche, à la fin de l'exécution, des statistiques relatives au calcul effectué. Pour plus d'informations sur les paramètres d'optimisation, taper `help odeset`, dans la fenêtre des commandes de Matlab.

Énoncé du TP ☞ Ⓢ

1. Résoudre symboliquement et numériquement au moyen du *solveur* `ode23`, l'équation différentielle du second ordre suivante :

$$\begin{cases} y''(t) + 3\,y = 4\,\sin(t) + \exp(-t) \\ y(0) = 1, \quad y'(0) = 1 \end{cases} \tag{43}$$

2. Afficher sur la même figure, la solution numérique et la solution analytique.

Script Matlab

```
clear all ; clc ; close all ;

%%%%%%%%%%%%%%%%%%%%%% SOLUTION SYMBOLIQUE %%%%%%%%%%%%%%%%%%%%%%
syms t

ySym = dsolve('D2y + 3*y = 4*sin(t) + exp(-t)','y(0) = 1','Dy(0) = 1'
    ,'t') ;

%%%%%%%%%%%%%%%%%%%%%% SOLUTION NUMERIQUE %%%%%%%%%%%%%%%%%%%%%%
a = 0 ; b = 10 ; n = 200 ; h = (b-a)/n ; tn = a :h: b ;
y = [1 ; 1] ;

eqs = @(tn,y) [y(2); - 3*y(1) + 4*sin(tn) + exp(-tn)] ;
opts = odeset('Stats','on','RelTol', 1e-3) ;
[tn, ysol] = ode23(eqs, [tn(1) ; tn(end)], y, opts) ; % SOLVEUR ode23

figure('color', [1 1 1]) ;
plot(tn, ysol(:,1),'o-','LineWidth',1) ; hold  on ;
plot(tn, double(subs(ySym, t, tn)),'o-r','LineWidth',1)
xlabel('t','FontSize',12) ; ylabel('y(t)','FontSize',12) ;
ih1 = legend('SOLUTION NUMERIQUE','SOLUTION ANALYTIQUE') ;
set(ih1,'Interpreter','none','Location','NorthWest','Box','off','
   Color','none')
```

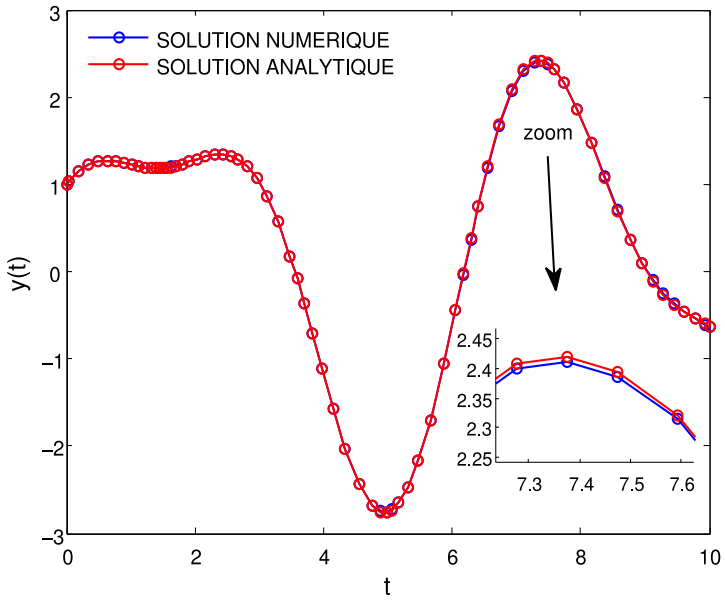


FIGURE 17: Comparaison entre la solution analytique et la solution numérique générée par le solveur `ode23`

Un autre façon de faire est de créer des fonctions imbriquées dans un fichier `m-file`, selon la syntaxe suivante :

```
function [] = eqs_sol(t,y)

tsol = [t(1) ; t(end)] ;

opts = odeset('Stats','on','RelTol', 1e-3) ;
[t , ysol] = ode23(@eqs, tsol, y, opts)  ;
plot(t,ysol(:,1))
function dydt = eqs(t, y)
dydt = [y(2) ; - 3*y(1) + 4*sin(t) + exp(-t)] ;

end
end
```

Qu'il faudra ensuite appeler, en tapant dans la fenêtre des commandes : `eqs_sol(t, [1 ; 1])`. Cette fonction accepte deux arguments en entrée, le vecteur `t` et les conditions initiales `[1 ; 1]`. Le fichier doit être sauvegardé sous le nom `eqs_sol.m`. Nous allons désormais résoudre une équation différentielle du second ordre avec paramètre variable, noté α :

Énoncé du TP ☞ Ⓢ

1. Résoudre numériquement au moyen du *solveur* `ode45`, l'équation différentielle du second ordre suivante :

$$\begin{cases} y''(t) - \alpha\,(1-y^2)\,y' = -y \\ y(0) = 1, \quad y'(0) = 0 \end{cases} \tag{44}$$

2. Afficher sur la même figure, la solution numérique pour différentes valeurs de α. Il en est de même pour les fonctions dérivées. Avec, le paramètre α qui prend ses valeurs de 1.5 à 4 par pas de 0.5.

Script Matlab

```
clc ; clear all ; close all ;
a = 1 ; b = 25 ; n = 100 ; h = (b-a)/n ; t = a :h: b ;
y = [1 ; 0] ; alpha = [1.5 ; 2.0 ; 2.50 ; 3.0 ; 3.5 ; 4.0] ;
col = {'o-k','o-m','-co','o-g','o-','o-r'} ;

for ik = 1 : numel(alpha)
eqs = @(t,y) [y(2); alpha(ik)*(1 - y(1).^2)*y(2) - y(1)] ;
opts = odeset('Stats','on','RelTol', 1e-3) ;
```

```
9  [t, ySol] = ode45(eqs, [t(1) ; t(end)], y, opts) ;
10
11 ysol = squeeze(ySol(:,1)) ; ysolDer = squeeze(ySol(:,2)) ;
12 figure(1) ; plot(t, ysol, col{ik}) ; hold on ; % SOLUTIONS
13 xlabel('t','FontSize',12) ; ylabel('y(t)','FontSize',12)
14 figure(2) ; plot(t, ysolDer, col{ik}) ; hold on ; % DERIVEES
15 xlabel('t','FontSize',12) ; ylabel('y^{''}(t)','FontSize',12)
16 end
```

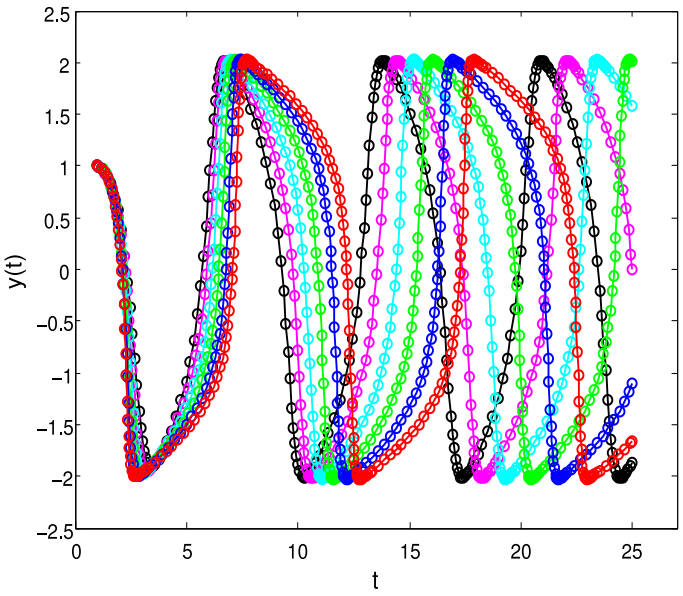


FIGURE 18: Solution numérique $y(t)$ pour différentes valeurs de α

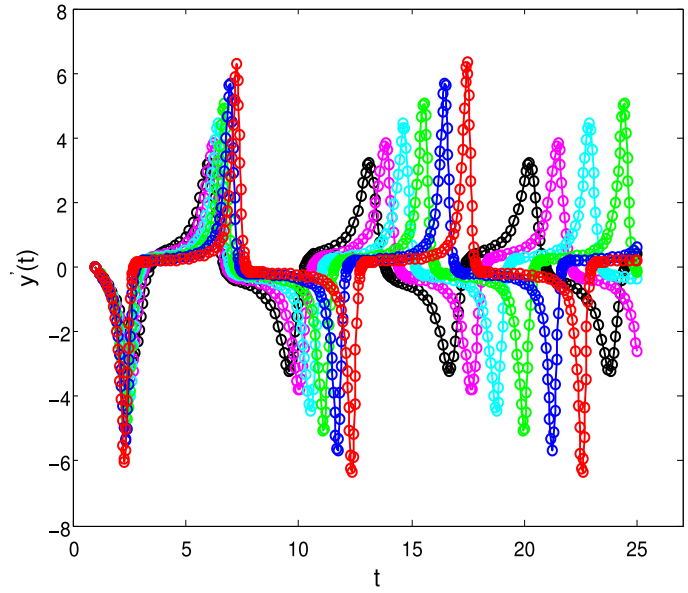


FIGURE 19: Dérivée première de la solution numérique $y(t)$ pour différentes valeurs de α