République Algérienne Démocratique et Populaire
Ministère de L'enseignement Supérieur et de la Recherche Scientifique
Université "Dr. Yahia FARES " de Médéa
Faculté des sciences et de la technologie
Département du Génie électrique et de l'informatique

Mémoire de fin d'études pour l'obtention du diplôme de Master en génie électrique

Option : Systèmes Intelligents et communicants

THEME

Factorisation polynomiale d'ordre supérieur par DSP à virgule fixe et flottante

Présenté par :
- Aberrahmane Elhacene
- Bendjebbas Aboubakr Essadik

Encadré par :
- Dr. BATEL Noureddine

Année universitaire 2011-2012

REMERCIEMENTS

Nous tenons à remercier notre encadreur *Dr. BATEL Noureddine* pour ses encouragements, ses conseils et sa disponibilité, qui nous ont été utile au cours de l'élaboration de notre étude.

A nos parents
A nos sœurs
A nos frères

A tous ceux qui nous ont aidés et encouragés.

RESUMES

ملخص

قمنا بوصف الطرق التكرارية لإيجاد حلول معادلات كثيرات الحدود ذات الأعلى ترتيب، و بالتحديد طريقتي لاجير و طريقة المصفوفة.و سنقوم بمحاكاة كل من الطريقتين في Matlab و LabVIEW, و تطبيقها على معالجين مختلفين DSP ذات النقطة الثابتة و اخرى متحركة.

الكلمات الرئيسية : طريقة لاجير، طريقة المصفوفة، DSP,LabVIEW,Matlab

Résume

Nous décrivons les méthodes itératives pour trouver les racines de polynômes d'ordre supérieurs et plus précisément, les deux méthodes matricielle et de Laguerre, et leurs simulations sous Matlab et LabVIEW, ainsi que leurs implémentations sous deux DSP différents virgule fixe et virgule flottante.

Mots clé : Méthode de Laguerre, méthode Matricielle, Matlab, LabVIEW, DSP.

Abstract

We describe iterative methods for finding polynomials roots of higher order and more specifically, the two methods Laguerre and matrix method, and their simulations in Matlab and LabVIEW, and their implementations in two different fixed-point DSP and floating point.

Key words: Laguerre method, Matrix method, Matlab, LabVIEW, DSP.

TABLE DES MATIÈRES

INTRODUCTION GENERALE 1

CHAPITRE 1 : RACINES D'ÉQUATION ALGEBRIQUES D'ORDRES SUPERIEURS ... 2

1.1. Introduction 3

1.2. Concepts importants 3

1.2.1. Convergence globale 3
1.2.2. Convergence inconditionnelle 4
1.2.3. Vitesse de Convergence 4
1.2.4. La stabilité numérique 4
1.2.5. Réduction polynomiale 4

1.3. Les méthodes de recherche des racines 5

1.3.1. Théorème des points fixes 5
1.3.4. Méthode de Newton-Raphson 6
1.3.5. Méthode de la sécante 7
1.3.6. Méthode de la bissection 8
1.3.7. Méthode de Halley 9
1.3.8. Méthode de Steffensen 9
1.3.9. L'algorithme Jenkins-Traub 9
1.3.10. Méthode de Laguerre 10
1.3.11. Méthode Matricielle 11

1.4. Algorithmes utilisés par certains logiciels 12

1.5. Conclusion 12

CHAPITRE2 :PROGRAMMATIONS DES METHODES LAGURRE ET CELLE MATRICIELLE UTILISANT MATLAB ET LABVIEW 13

2.1. Introduction 14

2.2. Les algorithmes 14

2.2.1. L'algorithme de la méthode de Laguerre 14
2.2.2. Algorithme de la méthode matricielle 18

2.3. Environnement du travail 19

2.4. Présentation de Matlab 19

2.4.1. Particularités de MATLAB 20
2.4.2. L'application Matlab 21

2.5. Présentation de LabVIEW 22

2.5.1. Utilisation de LabVIEW 22
2.5.2. L'application LabVIEW 23

2.6. Comparaison des différents essais 24

2.6.1. Discussion des résultats 29
2.6.2. Remarques 30

2.7. Conclusion 31

CHAPITRE 3 : IMPLEMENTATION DES METHODES LAGURRE ET MATRICIELLE SUR DIFFERENTS DSP 32

3.1. Introduction 33

3.2. Classification des DSPs 34

3.2.1. Virgule fixe ou flottante 34
3.2.2. Les DSPs à virgules flottantes 34
3.2.3. Les DSP à virgules fixes 35
3.2.4. Performance des DSP 37
3.2.5. Puissance de calcul d'un DSP 37

3.3. Description du DSP TMS320C6713 39

3.4. Description du DSP TMS320C5510 40

3.5. L'environnement intégré de développement (IDE) 41

3.5.1. CCS 41

3.6. Implémentation sous DSP 44

3.7. Comparaison des différents essais 44

3.7.1. discussion des résultats 47
3.7.2. Remarques 48

3.8. Conclusion : 49

CONCLUSION GENERALE 50

TABLE DES FIGURES

Figure 2.1: L'interface graphique Matlab. 21
Figure 2.2: L'interface graphique de l'application 21
Figure 2.3: L'interface Graphique LabVIEW 23
Figure 2.4: L'interface Graphique de l'application 24
Figure 2.5 : graphe du temps d'exécution selon le degré 30
Figure 3.1: Chaîne de traitement à base d'un DSP 33
Figure 3.2 : Représentation des nombres entiers codés sur 4 bits en complément à 2 35
Figure 3.3:Représentation des nombres fractionnaires codés sur 4bits en complément à 2 35
Figure 3.4: Evolution du temps d'exécution d'une opération MAC selon Texas Instruments 38
Figure 3.5 : Structure interne du DSP TMS320C6713 39
Figure 3.6 : Bloc-diagramme du TMS320C55x CPU 41
Figure 3.7 : Structure du système du développement. 42
Figure 3.8 : L'outil DSP/BIOS 43
Figure 3.9 : L'outil JTAG 43
Figure 3.10 : graphe du nombre de cycles en fonction de degré du polynôme 48
Figure 3.11 : Algorithme de décomposition QR 49

LISTE DES TABLEAUX

Tableau 1.1 : algorithmes utilisés par certains logiciels 12

Tableau 2.1 : polynôme de degré 10 avec racines réelles..... 25

Tableau 2.2 : polynôme de degré 10 avec racines complexes..... 26

Tableau 2.3 : polynôme de degré 12 avec racines réelles..... 27

Tableau 2.4 : polynôme de degré 12 avec racines complexes 28

Tableau 2.5 : le temps d'exécution des trois méthodes sous Matlab en fonction de l'ordre du polynôme(en secondes) 30

Tableau 3.1 : Comparaison entre diverses catégories de DSP 36

Tableau 3.2: Définitions des unités les plus courantes de mesures des performances des DSP 38

Tableau 3.3 : polynôme de degré 10 à racines réelles sous différent DSP..... 44

Tableau 3.4 : polynôme de degré 10 à racines complexes sous différent DSP..... 45

Tableau 3.5 : polynôme de degré 12 à racines réelles sous différent DSP..... 46

Tableau 3.6 : polynôme de degré 12 à racines complexes sous différent DSP..... 46

Tableau 3.7 : le nombre de Cycles selon l'ordre de polynôme..... 47

INTRODUCTION GENERALE

Ce mémoire décrit la problématique du développement d'algorithmes numériques adéquat pour la recherche des racines réelles et complexes propres à des polynômes de degré supérieur.

Ce problème n'est pas posé uniquement à notre ère, mais il l'était depuis les civilisations sumériennes puis égyptiennes pour chaque nouveau degré polynomial, de façon croissante.

La résolution de ce type de problème, repose sur une diversité des méthodes exposées dans les ouvrages mathématique spécialisés. La recherche des racines se scinde en deux grands types de méthodes existantes :

- Les méthodes dites exactes: elles sont précises mais valable à des degrés inferieurs ou fiables ordres polynomiaux.
- Les méthodes dites approximatives: elles sont appliquées principalement aux polynômes d'ordre supérieurs a quatre (>4). Mais avec des résultats approchés, manquant de précision.

Du côté électronique les circuits les plus utilisés au développement d'applications à base d'un traitement de signal nous trouvons les cartes FPGA, les PIC et plus spécialement les DSP.

La plupart des applications exigent des traitements en temps réel.

Ainsi, l'outil de développements d'application utilisé au niveau matériel est surtout à base de DSP « Digital Signal Processor » qui supporte le langage C et assembleur.

De notre côté, l'outil de développement utilisé est soit le PC, sur lequel sont installés les deux logiciels Matlab et LabVIEW, soit des cartes DSP où l'implémentation de nos applications se fait à travers l'environnement de développement intégré CCS de Texas Instrument.

Dans le premier chapitre, nous donnons un bref aperçu de certaines méthodes d'extraction des racines de polynômes. Il s'agit notamment du théorème de points fixes, la méthode de newton-Raphson, la méthode de la sécante, la méthode de bissection, la méthode de Halley, la méthode de Steffensen, l'algorithme de Jenkins-Traub, l'algorithme de Laguerre et la méthode Matricielle.

Dans le deuxième chapitre, nous implémentons la méthode de Laguerre et la méthode matricielle en utilisant Matlab et LabVIEW comme moyens logiciels. Des tests des deux algorithmes avec une comparaison des résultats sont entamés par la suite.

Dans le troisième chapitre, nous simulons les deux méthodes précédentes sur deux familles DSPs (TMS320C67XX et TMS320C55XX), et les résultats sont discutés.

CHAPITRE 1:

RACINES D'ÉQUATION ALGEBRIQUES D'ORDRES SUPERIEURS

1.1. Introduction

La théorie des équations algébriques repose sur le théorème fondamental de l'algèbre qui assure l'existence de solutions. Ce théorème encore appelé *théorème d'Alembert*, affirme que l'équation algébrique $P(x)$ où P est un polynôme de degré n, admet exactement n racines distinctes ou non, réelles ou complexes. Dans le cas complexe, les racines sont deux à deux conjuguées. Depuis l'Antiquité, l'homme a cherché des formules explicites donnant les valeurs des racines en fonction des coefficients du polynôme $P(x)$, à l'image de l'équation du second degré $x^2 + px + q = 0$ qui admet si $p^2 - 4q > 0$, deux racines réelles x_1 et x_2 vérifiant $p = x_1 + x_2$ et $q = x_1 x_2$. Au XVI^{e} siècle, Niccolo Tartaglia, Scipione Del Ferro et Antonio Fior ont cherché à résoudre des équations cubiques, mais c'est Girolamo Cardano (Cardan) qui donna en 1545 dans son *Ars magna*, les formules de résolution. L'équation générale du troisième degré exprimée sous forme alternée [1]

$$x^3 - ax^2 + bx - c = 0$$

Admet dans certains cas trois racines qui vérifient $a = x_1 + x_2 + x_3$, $b = x_1x_2 + x_2x_3 + x_1x_3$ et $c = x_1x_2x_3$.

L'équation du quatrième degré a été résolue par Luigi Ferrari, un disciple de Cardan. Exprimée sous forme alternée, l'équation

$$x^4 - ax^3 + bx^2 - cx + d = 0$$

Admet dans certains cas quatre racines x_1, x_2, x_3, x_4 qui vérifient $a = x_1 + x_2 + x_3 + x_4$ $b = x_1x_2 + x_2x_3 + x_1x_3 + x_1x_4 + x_2x_4 + x_3x_4$, $c = x_1x_2x_3 + x_1x_2x_4 + x_1x_3x_4 + x_2x_3x_4$ et $d = x_1x_2x_3x_4$.

L'équation générale de degré supérieur ou égal à 5 n'est pas résoluble par radicaux. Ce résultat a été démontré par N.H. Abel en 1824.

1.2. Concepts importants

Une méthode numérique pour déterminer un zéro d'un polynôme, prend généralement la forme d'une prescription de construire une ou plusieurs suites z_n de nombres complexes ou réelles censés converger vers un zéro du polynôme. Comme on pouvait s'y attendre, chaque algorithme à ses avantages et ses inconvénients, et par conséquent, le choix du meilleur algorithme pour un problème donné n'est jamais facile.

Tout algorithme raisonnable doit converger, c'est-à-dire, la suite qu'il génère devrait, dans des conditions appropriées, converge vers un zéro du polynôme donné. Nous allons voir d'autres propriétés souhaitables qu'un algorithme peut inclure.

1.2.1. Convergence globale

De nombreux algorithmes peuvent être garantis de manière à converger uniquement si la valeur de départ z_0 est suffisamment proche de la mise à zéro du polynôme. On dit qu'ils sont localement convergents. Les algorithmes qui ne nécessitent pas d'être suffisamment près de la valeur de départ sont globalement convergents.

1.2.2. Convergence inconditionnelle

Certains algorithmes ne convergent que si le polynôme donné à des propriétés spéciales, par exemple, tous les zéros sont simples. Ces algorithmes sont conditionnellement convergents. Si un algorithme est convergent (localement ou globalement) pour tous les polynômes, il est inconditionnellement convergent.

1.2.3. Vitesse de Convergence

La notion d'ordre est souvent utilisée comme une mesure de la vitesse de convergence d'un algorithme. L'ordre v est défini comme une borne supérieure de tous les nombres réels α tels que [2]:

$$\lim_{n\to\infty} sup \frac{|z_{n-1} - \xi|}{|z_n - \xi|^{\alpha}} < \infty. \qquad (1.1)$$

La méthode de Newton, par exemple, a un ordre de convergence $v = 2$, ce qui signifie asymptotiquement que le nombre de décimales correctes est doublé à chaque étape. Ainsi, l'ordre de convergence le plus élevé, est le plus rapide ; $|z_n - \xi|$ converge finalement à zéro.

1.2.4. La stabilité numérique

D'habitude, tous les calculs sont effectués sur un système à précision finie utilisant des nombres discrets et bornés de la machine. L'ensemble des chiffres fournis par arithmétique en virgule flottante est fini. Ainsi, les algorithmes conçus à l'origine pour travailler de façon portable, sont adaptés aux " nombres des machines " par les périphériques d'arrondissement et mise en l'échelle. Pas tous les algorithmes qui sont aussi insensibles à cette adaptation. Par la stabilité numérique, nous entendons le manque de sensibilité d'arrondissement et à la taille des opérations ou, plus précisément, la sensibilité de l'algorithme ne doit pas être plus grande que celle inhérente au mauvais conditionnement du problème à trouver les zéros.

1.2.5. Réduction polynomiale

La recherche de racines d'un polynôme se construit de la manière suivante :

Soit $P_n(x)$ un polynôme de degré n . Si on obtient une première racine, en peut écrire :

$$P_n(x) = (x - x_1)P_{n-1}(x) \qquad (1.2)$$

Où $P_{n-1}(x)$ un polynôme de degré $n-1$. Ainsi, théoriquement, une fois obtenue une première racine, on peut recommencer la recherche d'une autre racine pour un polynôme de degré strictement inférieur, on parle de factorisation du polynôme original. Successivement, on poursuit cette procédure jusqu'à l'obtention de l'ensemble des n racines de polynôme $P_n(x)$. Rappelons que les polynômes à coefficients complexes se factorisent en un produit de monômes de degré 1. Cette propriété exprime le fait que le polynôme à coefficients complexes, a l'ensemble de ses racines dans le plan complexe,

$$P_n(x) = \prod_{i=1}^{n}(x - x_i) \qquad (1.3)$$

1.3. Les méthodes de recherche des racines

- **Localisation des racines**

En pratique, la mise en œuvre d'un algorithme de recherche de solution d'équations suppose que nous connaissons une région dans laquelle se trouve cette solution. La théorie donne quelques critères de localisation lorsque l'équation est une équation polynomiale. Le théorème de Rolle (1690) affirme qu'entre deux racines de l'équation $P(x) = 0$, où P est un polynôme, il existe au moins une racine de l'équation dérivée $P'(x) = 0$. La règle de Descartes affirme que le nombre de racines positives d'un polynôme de la forme :

$$P(x) = a_0 + a_1 x + \cdots + a_n x^n \qquad (1.4)$$

est inférieur au nombre de changements de signes de la suite $(a_0, a_1, \ldots a_n)$.

Le théorème de Sturm (1829) donne un algorithme pour déterminer le nombre de racines d'un polynôme entre deux réels.

Notons enfin le critère de Routh-Hurwitz [1] qui établit une condition nécessaire et suffisante pour qu'un polynôme défini par l'expression (1.4) où tous les coefficients réel, ait toutes ses racines à parties réelles négatives.

- **Approximations successives**

Dans les méthodes d'approximations successives, l'équation $f\,(x) = 0$ est remplacée par l'étude d'une suite numérique convergente [1] :

$$x_{n+1} = \varphi(x_n)$$

qui permet d'obtenir en un nombre fini d'itérations, une solution approchée de l'équation. En général, on pose $\varphi(x) = x - f(x)$. Dans la méthode de Lagrange, on remplace la fonction f par le segment de droite passant par les points $(a, f(a))$ et $(b, f(b))$:

$$\varphi(x) = a - f(a)\frac{x - a}{f(x) - f(a)} \qquad (1.5)$$

Dans la méthode de Newton, on remplace la fonction f entre les points d'abscisse a et b par la tangente à la courbe en ces points:

$$\varphi(x) = x - \frac{f(x)}{f'(x)} \qquad (1.6)$$

1.3.1. Théorème des points fixes

Trouver le zéro réel d'une fonction $f\,(x)$ est la même que la recherche de l'intersection de $f\,(x)$ avec l'axe x. Considérons la fonction $G(x) = f(x) + x$, le point d'intersection de $G(x)$ et $y = x$ ou bien, $G(x) = x$ est exactement le même que le zéro réel de $f(x)$.

Toute solution x^* de l'équation $G(x) = x$, est appelée un point fixe de la fonction $G(x)$.

Soit $G(x)$ est une fonction continue sur $[a;\ b]$ où $|G'(x)| \leq k < 1$ pour tout $x \in (a;\ b)$[3]. Si $x_{n+1} = G(x_n)$, Alors :

$|x_{n+1} - x_n| = |G(x_n - x_{n-1})| \approx |G'(x)||x_n - x_{n-1}| \leq k|x_n - x_{n-1}| \cdots < k^n.|x_1 - x_0| < |x_1 - x_0|$

Ainsi $\lim_{n\to\infty}|x_{n+1} - x_n| = 0$, d'où $G(x)$ a un unique point fixe x^* sur (a, b).

a) Algorithme

Étape 1. Définir $x^* > 0$ comme zéro, N le nombre maximal d'itération, et Choisir x_n proche de x^*. $n = 1$;

Étape 2. Trouver $x_{n+1} = G(x_n)$;

Étape 3. Si $|x_{n+1} - x_n| = x^*$ ou $n > N$, Alors $x^* = x_n$ et arrêter ;

Étape 4. Incrémenter $n = n + 1$ et passer à l'Étape 2 ;

b) Avantage : nécessite une fonction d'évaluation. Lorsque $G'(x) \leq k < 1$ sur (a, b), la convergence vers la solution unique est garantie. La convergence est quadratique, lorsque $G"(x)$ est continue sur (a, b).

c) Inconvénient : pour trouver la racine de la fonction $f(x)$, il n'est pas toujours évident de construire $G(x)$ pour une convergence rapide. Si la plage de $G'(x)$ sur (a, b) n'est pas connue, alors la convergence est pas garantie.

1.3.4. Méthode de Newton-Raphson

La méthode de Newton-Raphson, encore appelée méthode des tangentes a été exposée par Newton vers 1669 et complétée par Joseph Raphson en 1690.

Cette méthode est l'une des plus puissantes méthodes numériques pour résoudre le problème de recherche des racines F(x)=0 [3].

Supposons que $f(x)$ est en permanence deux fois dérivable dans un certain intervalle de point x^*, telle que $f(x^*) = 0$, $f'(x^*) \neq 0$ et $f(x)$ est quasi-linéaire proche de x^* (c'est-à-dire, $|f''(x)| < \epsilon$ proche de x^*).

Soit $x_0$un point proche de x^*, considérons polynôme de Taylor de $f(x)$, développé autour de x_0,

$$f(x) = f(x_0) + (x - x_0)f'(x) + f''(\xi(x)) \qquad (1.7)$$

Où $\xi(x)$ se situe entre x et x_0. Puisque $F(x) = 0$ et $F(x)$ est quasiment linéaire de x, nous avons

$$0 \approx f(x_0) + (x^* - x_0)f'(x^*)$$

Ou $$x^* \approx x_0 - \frac{f(x_0)}{f\prime(x_0)}$$

Donc $$x_1 \approx x_0 - \frac{f(x_0)}{f'(x_0)} \qquad (1.8)$$

Devrait être une meilleure approximation de x à x_0.

a) Algorithme :

Étape 1. Définir $x^* > 0$ comme zéro, N le nombre maximal d'itération, et Choisir x_0 proche de x^* et $n = 0$;

Étape 2. Si $|f(x)| = x^*$ ou $n > N$, Alors $x^* = x_n$ et arrêter ;

Étape 3. Définir $x_{n+1} = x_n - \frac{f(x_0)}{f\prime(x_0)}$ puis incrémenter $n = n + 1$ et passer à l'Étape 2 ;

b) A**vantage :** La convergence est très rapide.

c) Inconvénient : Nécessite souvent une bonne supposition initiale de x_0. Nécessite deux évaluations de la fonction ($f(x_n)$ et $f'(x_n)$) à chaque itération. Si $f(x)$ est un polynôme, alors la méthode de Horner peut être utilisée pour obtenir les deux fonctions $f(x_n)$ et $f'(x_n)$.

1.3.5. Méthode de la sécante

La méthode de la sécante, encore appelée $méthode\ de\ la\ fausse\ position$ ou "$regula\ falsi$ ", a été employée au XVI[e] siècle par Viète puis, plus tard par Descartes.

La méthode sécante est une variante de la méthode de Newton. Nous avons besoin de faire les mêmes hypothèses de $f(x)$ comme nous l'avons fait pour la méthode de Newton [3].

Supposons que x_0 et x_1 sont deux points proches de x^*. Si $x_1 - x_0 < \epsilon$, Alors

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

Considérons le polynôme de Taylor de $f(x)$, développé au tour de x_1,

$$f(x) = f(x_1) - (x_1 - x_0)f'(x) + \frac{(x_1 - x_0)^2}{2} f''(\xi(x)) \qquad (1.9)$$

Où $\xi(x)$ se situe entre x et x_1. En remplaçant $f'(x_1)$ par $\frac{f(x_1) - f(x_0)}{x_1 - x_0}$ dans la formule (1.9) et en utilisant le fait que $f(x^*) = 0$ et $f(x)$ est quasiment linéaire de x^*,

$$0 \approx f(x_1) + (x^* - x_0)\frac{f(x_1) - f(x_0)}{x_1 - x_0} \qquad (1.10)$$

$$x^* \approx x_1 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_1)$$

Par conséquent en choisissant :

$$x_2 \approx x_1 - \frac{x_1 - x_0}{f(x_1) - f(x_0)} f(x_1) \qquad (1.11)$$

Nous devrions obtenir une meilleure approximation de x^* que x_1.

a) Algorithme :

Étape 1. Définir $x^* > 0$ comme zéro, N le nombre maximal d'itération, et Choisir x_0 et x_1 proche de x^*. $n = 1$;

Étape 2. Si $f(x) = x^*$ ou $n > N$, Alors $x^* = x_n$ et arrêter ;

Étape 3. Définir $x_{n+1} = x_n - \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} f(x_n)$ puis incrémenter $n = n + 1$ et passer à l'Étape 2;

b) Avantage: Convergence rapide et une seule fonction d'évaluation**.**

c) Inconvénient : Nécessite des calculs par itération, des soins doivent être pris pour éviter une divergence.

1.3.6. Méthode de la bissection

Dans la *méthode de dichotomie* ou *méthode de la bissection*, l'intervalle de recherche de la solution est coupé en deux à chaque pas d'itération [1]. On détermine progressivement un intervalle de plus en plus fin dans lequel se trouve la solution cherchée.

Soit f une fonction numérique et continue sur un intervalle $[a_0, b_0]$, et $f(a_0)f(b_0) \leq 0$. Alors il existe $x^* \in [a_0, b_0]$ tel que $f(x) = 0$.

a) Algorithme A :

Étape 1. Définir $x^* > 0$ comme zéro, N le nombre maximal d'itération et mettre en $T = a_0$;

Étape 2. Donnée $x_0 = b_0$ et $n = 1$;

Étape 3. Trouver $x_n = \frac{T + x_{n-1}}{2}$;

Étape 4. Si $|f(x)| = x^*$ ou $n > N$, Alors $x^* = x_n$ et arrêter ;

Étape 5. Si $S = f(x_n)f(x_{n-1}) < 0$, Alors $n = n + 1, T = x_{n-1}$ et passer à L'Étape 3;

Étape 6. Si $S = (x_n)f(x_{n-1}) > 0$, Alors $n = n + 1$, et passer à L'Étape 3 ;

Nous avons ainsi :

$$x^* = \lim_{n \to \infty} x_n$$

$$|x_n - x^*| \leq \frac{b_0 - a_0}{2^n} = 2^{-n}(b_0 - a_0) \qquad (1.12)$$

Pour une précision $\epsilon = 10^{-r}$, et en acceptant x_N comme x^*, nous avons besoin de choisir un entier N qui satisfait la condition de $\log 2^{-N}(b_0 - a_0) \leq \log 10^{-r}$. Cela implique que le nombre de pas d'itérations est :

$$N \geq \frac{\mathrm{r} + \log(b_0 - a_0)}{\log 2} \qquad (1.13)$$

b) Algorithme B :

Étape 1. Définir $x^* > 0$ comme zéro, N le nombre maximal d'itération et mettre en n à zéro;

Étape 2. Trouver $x_n = \frac{a_n+b_n}{2}$;

Étape 3. Si $|f(x)| = 0^*$ ou $n > N$, Alors $x^* = x_n$ et arrêter ;

Étape 4. Si $S = f(a_n)f(b_n) < 0$, Alors $a_{n+1} = a_n$, $b_{n+1} = x$ et passer à L'Étape 2;

Étape 5. Si $S = (a_n)f(b_n) > 0$, Alors $a_{n+1} = x$, $b_{n+1} = b_n$ et passer à L'Étape 2;

Nous avons donc [3]:

$$x^* = \lim_{n\to\infty} a_n = \lim_{n\to\infty} b_n$$

$$N \geq r + \log(b_0 - a_0)$$

c) Avantage : Implémentation facile et garanti de la convergence.

d) Inconvénient : exige a_0 et b_0 pour satisfaire $f(a_0)\, f(b_0) < 0$. Très lente convergence. En général, on cherche la solution exacte, mais seulement voir si l'un des critères d'arrêt, tels que :

$|f(x)| < \epsilon$, $|x_n - x_{n-1}| < \epsilon$, Ou $N \geq \frac{\mathrm{r}+\log(b_0-a_0)}{\log 2}$ sont satisfaits.

1.3.7. Méthode de Halley

On pose :

$$x_{n+1} = x_n - \frac{1}{a_n} \quad \text{où} \quad a_n = \frac{f\prime(x_n)}{f(x_n)} - \frac{1}{2}\left(\frac{f\prime\prime(x_n)}{f\prime(x_n)}\right) \qquad (1.14)$$

Cette méthode est plus lente que la méthode de Newton, mais implique moins d'arithmétique, et est considérée comme de très bonne qualité pour trouver de véritables racines de polynômes de Horner [3].

1.3.8. Méthode de Steffensen

Pour améliorer la convergence de la méthode des approximations successives, on pose [3] :

$$x_{n+1} = x_n - \frac{f(x_n)}{S(x_n)}, \qquad \text{où} \quad S(x) = \frac{f\big(x+f(x)\big)-f(x)}{f(x)} \qquad (1.15)$$

Cette méthode est aussi rapide que la méthode de Newton et ne nécessite pas l'évaluation de $f\,'(x)$.

1.3.9. L'algorithme Jenkins-Traub

La méthode Jenkins-Traub est une méthode qui calcule les zéros du polynôme en trois étapes [2].
Une première étape est d'aucun changement: l'objectif principal de cette étape est d'accentuer les petits zéros.

Étape deux, est celle du "fixe" : le but de cette étape est de séparer des zéros d'égaux ou à peu près de même grandeur.
Étape trois, est celle du " Variable " : on termine lorsque la valeur calculée du polynôme correspond à un possible zéro inférieur ou égal à un.

1.3.10. Méthode de Laguerre

La méthode de Laguerre utilise le fait que les dérivées logarithmiques successives d'un polynôme divergent au voisinage d'une racine. En prenant le logarithme de l'équation $P_n(x) = \prod_{i=1}^{n}(x - x_i)$ (Réduction polynomiale), on obtient

$$\ln(|P_n(x)|) = \sum_{i=1}^{n} \ln(|x - x_i|) \qquad (1.16)$$

En dérivant l'équation (1.16), on obtient :

$$\frac{d\ln(|P_n(x)|)}{dx} = \sum_{i=1}^{n} \frac{1}{x - x_i}$$

$$= \frac{P_n{}'(x)}{P_n(x)} = G \qquad (1.17)$$

En dérivant l'équation (1.17), il vient

$$-\frac{d^2\ln(|P_n(x)|)}{dx^2} = \sum_{i=1}^{n} \frac{1}{(x - x_i)^2}$$

$$= \left[\frac{P_n{}'(x)}{P_n(x)}\right]^2 - \frac{P_n{}''(x)}{P_n(x)} = H \qquad (1.18)$$

Soit la racine x_1à déterminer, on suppose que la valeur de départ x est située à une distance a de x_1 et que l'ensemble des autres racines sont situées à une distance supposée identique et qui vaut b

$$x - x_1 = a$$
$$x - x_i = b \qquad i \in [2, n]$$

En insérant les équations $x - x_1 = a$, et $x - x_i = b$ dans les équations G et H, on en déduit respectivement les relations suivantes :

$$\frac{1}{a} + \frac{n-1}{b} = G$$
$$\frac{1}{a^2} + \frac{n-1}{b^2} = H$$

Après élimination de b, la valeur de a est :

$$a = \frac{n}{G \pm \sqrt{(n-1)(nH - G^2)}} \qquad (1.19)$$

Le signe placé devant la racine du dénominateur est choisi tel que le dénominateur soit le plus grand possible. $(x - a)$ devient alors la nouvelle valeur de départ et on itère le processus. En combinant cette méthode avec celle de la réduction polynomiale, on peut calculer l'ensemble des racines. En pratique, comme chaque racine n'est déterminée qu'avec une précision finie, il est nécessaire d'ajouter une procédure dite de lissage pour éviter les problèmes d'instabilité numérique [9].

1.3.11. Méthode Matricielle

La $méthode\ matricielle$, encore appelée méthode de la matrice associée ou de la matrice compagnon, permet de déterminer les zéros d'un polynôme en résolvant un problème de détermination de valeurs propres. Soit le polynôme $P(x)$:

$$P(x) = \sum_{i=0}^{n} a_i x^i \qquad (1.20)$$

Et A la matrice compagnon

$$A = \begin{pmatrix} -\frac{a_{n-1}}{a_n} & -\frac{a_{n-2}}{a_n} & \cdots & -\frac{a_1}{a_n} & -\frac{a_0}{a_n} \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & & & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix} \qquad (1.21)$$

Comme,

$$P(x) = (-1)^n \det(A - xI) \qquad (1.22)$$

Les zéros de $P(x)$ sont les valeurs propres de A. Il suffit donc de savoir déterminer les valeurs propres d'une matrice pour résoudre une équation polynomiale. Si le polynôme est de la forme

$$Q(x) = b_n x^n + b_{n-1} x^{n-1} + \cdots + b_1 x + b_0 \qquad (1.23)$$

Avec $b_n \neq 0$, il suffit de diviser chaque coefficient par b_n, et d'appliquer la méthode Matricielle pour trouver les racines de Q qui coïncident avec les racines de Q/b_n.

Calcul des valeurs propres :
Il s'agit de calculer les valeurs propres de matrices carrées d'ordre n à coefficients réels ; cependant, les méthodes proposées peuvent s'appliquer aux matrices à éléments complexes.
Diverses méthodes donnent des résultats plus ou moins intéressants selon la taille de la matrice A ou son conditionnement. Le problème est donc la résolution de l'équation caractéristique ou séculaire :

$$AX = \lambda X$$

Où X s'appelle vecteur propre et λ valeur propre.

1.4. Algorithmes utilisés par certains logiciels

Le tableau (1.1) récapitule les algorithmes utilisés par certains des langages de programmation plus populaires.

Logiciel	**Nom de Fonction**	**Méthode utilisée**
NAG F77 library	C02AFF	Laguerre modifié
MATLAB	Roots	Valeurs propres de la matrice compagnon équilibrée
HSL	PA16	Madsen-Reid
IMSL	CPOLY	Jenkins-Traub
Mathematica	NSolve	Jenkins-Traub
Numerical Recipes	Zroots	Laguerre
Labview	Polynomial Roots	Newton (racines réelles) Bairstow (racines complexe)

Tableau 1.1 : algorithmes utilisés par certains logiciels

1.5. Conclusion

Les méthodes citées dans ce chapitre sont des méthodes itératives appliquées pour résoudre le problème de recherche des racines d'un polynôme quelconque, mais surtout d'ordre supérieur, à une précision donnée.

Le chapitre deux, nous implémentons les deux méthodes *Laguerre* et *Matriciel* sous Matlab et LabVIEW.

CHAPITRE 2

PROGRAMMATIONS DES METHODES LAGURRE ET CELLE MATRICIELLE UTILISANT Matlab ET LabVIEW

2.1. Introduction

Dans ce chapitre, on va illustrer la méthode de Laguerre (basée essentiellement sur la condition de convergence) et la méthode Matricielle (à travers ses valeurs propres). Les outils de programmation sont les logiciels, Matlab et LabVIEW.

2.2. Les algorithmes

2.2.1. L'algorithme de la méthode de Laguerre

Cette méthode contient quatre fonctions qui sont l'évaluation polynômiale, la déflation polynômiale, la fonction de Laguerre, et la fonction Polyroots.

2.2.1.1. La fonction d'évaluation polynomiale

Le polynôme est évalué selon l'équation:

$$P(x) = a_1 x^n + a_2 x^{n-1} + \cdots + a_n x + a_{n+1} \qquad (2.1)$$

de la gauche vers la droite par l'algorithme suivant (nous supposons que les coefficients sont stockés dans le vecteur $A = (a_1\ a_2 \ldots a_{n+1})^t$:

P=0 ;

n= la taille du polynôme ;

Pour i=1 à n+1 :

$P = P + A(i) * x^{(n-i+1)}$

Fin.

On remarque que $a_n x$ et une seule multiplication, $a_{n-1}x^2$ nécessite deux multiplication, etc…. . Nous en déduisons que le nombre de multiplications dans cet algorithme est :

$$1 + 2 + 3 + \cdots + n - 1 = \frac{1}{2} n(n-1) \qquad (2.2)$$

Si n est grand, le nombre de multiplications est grand, mais il peut être considérablement réduit si nous évaluons le polynôme de la droite vers la gauche. Exemple:

$$P_4(x) = a_1 x^4 + a_2 x^3 + a_3 x^2 + a_4 x + a_5$$

$$= a_5 + x\left(a_4 + x\left(a_3 + x\left(a_2 + a_1 x\right)\right)\right)$$

D'où la séquence efficace de calcul pour évaluer le polynôme :

$$P_0(x) = a_1$$

$$P_1(x) = a_2 + x\, P_0(x)$$

$$P_2(x) = a_3 + x P_1(x)$$

$$P_3(x) = a_4 + x\,P_2(x)$$

$$P_4(x) = a_5 + x\,P_3(x)$$

Pour un polynôme de degré n, la procédure peut être résumée comme :

$$P_0(x) = a_1$$

$$P_i(x) = a_{n+i} + xP_{i-1}(x)\,,\ i = 1,2,3 \ldots n. \qquad (2.3)$$

Conduisant à l'algorithme :

```
p = a (1);
Pour i = 1 à n
    p = p*x + a (i+1)
Fin.
```

Le dernier algorithme implique qu'il y a n multiplications, et se rend plus efficace pour $n > 3$. Mais l'économie de temps calcul n'est pas la raison principale pour laquelle cet algorithme doit être utilisé. La procédure avec un nombre minimum de multiplications cumule une plus petite erreur d'arrondi.

L'algorithme d'évaluation de la première et de la deuxième dérivé de $P_n(x)$, est toujours le même. Nous obtenons par différenciation:

$$P_0{}'(x) = 0;\quad P_i{}'(x) = P_{i-1}(x) + xP_{i-1}{}'(x); i = 1,2, \ldots, n \qquad (2.4a)$$

$$P_0{}''(x) = 0;\quad P_i{}''(x) = 2P'_{i-1}(x) + xP''_{i-1}(x); i = 1,2, \ldots, n \qquad (2.4b)$$

```
Evalpoly ( )
Donner Polynôme, x ;
Début
 valeur_Dérive1<= 0 ;
 valeur_Dérive2<= 0 ;
 Valeur_Polynôme <= Polynôme (1) ;
 Pour (i=1 à la taille de (Polynôme-1)) :
     valeur_Dérive2<= valeur_Dérive2*x+2.0* valeur_Dérive1 ;
     valeur_Dérive1<= valeur_Dérive1*x+ Valeur_Polynôme;
     Valeur_Polynôme <= valeur_Polynôme *x+ Polynôme (i+1);
 Fin ;
 Retourner (Valeur_Polynôme, valeur_Dérive1, valeur_Dérive2) ;
Fin ;
```

2.2.1.2. Fonction de Déflation polynomiale

Une fois une racine r de $P_n(x) = 0$ calculée, il est souhaitable de factoriser le polynôme:

$$P_n(x) = (x - r)P_{n-1}(x) \qquad (2.5)$$

Cette procédure, connue sous le nom de déflation ou division synthétique, calcule les coefficients de $P_{n-1}(x)$. Comme les zéros restants de $P_n(x)$ sont également les zéros de $P_{n-1}(x)$, la procédure de recherche des racines peut maintenant être appliquée à $P_{n-1}(x)$. La déflation rend ainsi progressivement plus facile de trouver les racines, parce que le degré du polynôme est réduit à chaque fois qu'une racine est trouvée. De plus, en éliminant les racines qui ont déjà été trouvés, le temps de calcule devient plus en plus rapide.

Si nous écrivons :

$$P_{n-1}(x) = b_1 x^{n-1} + b_2 x^{n-2} + \cdots + b_{n-1}x + b_n$$

L'expression (2.5) devient:

$$a_1 x^n + a_2 x^{n-1} + \cdots + a_n x + a_{n+1}$$

$$= (x - r)(b_1 x^{n-1} + b_2 x^{n-2} + \cdots + b_{n-1}x + b_n)$$

Nous obtenons donc les coefficients de $P_{n-1}(x)$:

$$b_1 = a_1 ; \quad b_2 = a_2 + r\, b_1; \ldots; b_n = a_n + r b_{n-1} \qquad (2.6)$$

Ce qui conduit à l'algorithme de déflation Horner:

```
Defpoly ( )
 Donner Polynôme, Racine ;
 Début
 Nv_Polynôme (1) =Polynôme(1);
  Pour (i=2 à la taille de (Polynôme)-1) :
    Nv_Polynôme(i) = Polynôme(i) +Racine*Nv_Polynôme (i-1);
  Fin;
 Retourner Nv_Polynôme ;
Fin.
```

2.2.1.3. Fonction de Laguerre

La procédure à suivre pour trouver un zéro d'un polynôme par la formule générale de Laguerre est la suivante:

- Supposons que x est une approximation de la racine de $P_n(x) = 0$ (n'importe laquelle).
- Évaluer $P_n(x)$, $P'_n(x)$ et $P''_n(x)$ en utilisant la fonction d'évaluation polynomiale.
- Calculer G (x) et H (x) par les équations (1.17) et(1.18).

- Déterminer la meilleure racine r de l'équation(1.19) par le choix convenable du signe au dénominateur.
- Prendre $x = r$ et répéter les étapes 2 à 5 jusqu'à ce que $|Pn(x)| < \varepsilon$ ou $|x - r| < \varepsilon$, tel que ε est la tolérance d'erreur.

Une belle propriété de la méthode de Laguerre est qu'elle converge vers une racine, à quelques exceptions près, à partir de n'importe quelle valeur de départ de x.

```
Laguerre ( )
 Donner Polynôme, tolérance ;
 Début
  Racine = nombre aléatoire ;
  N =la taille de Polynôme;
  Pour (i=1 à 30)
       Evalpoly (Polynôme, Racine) ;
       Si |valeur_Polynôme|<tolérance Alors exit ;
       G=valeur_Dérive1 / valeur_Polynôme ;
       H = G*G- valeur_Dérive2/ valeur_Polynôme ;
       F =√((N − 1) ∗ (N ∗ H − G ∗ G))
       Si |G+F|=<|G-F| Alors
         x = N/ (G+F);
       Sinon
         x = N/ (G-F) ;
       Fin Si ;
       Racine =Racine-x ;
       Si |x|<tolérance Alors exit ;
  Fin;
  Retourner Racine ;
 Fin.
```

2.2.1.4. Fonction Polyroots

La fonction Polyroots dans ce module calcule toutes les racines de $P_n(x) = 0$, où le polynôme $P_n(x)$ est défini par ses coefficients $Coef = [a1, a2, a3, \ldots, an + 1]$. Après la première racine calculée par la fonction Laguerre, le polynôme est déflaté à l'aide de Defpoly et le prochain zéro est cherché en appliquant Laguerre pour le polynôme déflaté. Ce processus est répété jusqu'à ce que toutes les racines sont trouvées. Si une racine calculée a une très petite partie imaginaire, il est très probable que cela représente une erreur d'arrondi.

```
    Polyroots ()
     Donner Polynôme;
     Début
      Tolérance=1.0 e-6 ;
      n=la taille de Polynôme-1 ;
       Pour i=1 à n :
           Laguerre (Polynôme, Tolérance);
             (nb_complexe de racines) < Tolérance Alors
              Racine = nb_ réel de racine ;
           Racines(i)=Racine ;
           Defpoly (Polynôme, Racine) ;
      Fin;
      Retourner Racines ;
     Fin ;
```

2.2.2. Algorithme de la méthode matricielle

Dans une première étape, nous allons convertir le polynôme en une matrice qui s'appelle la matrice compagnon.

La deuxième étape est le calcul des valeurs propres de la matrice compagnon.

Les valeurs propres sont les racines du polynôme.

```
Matricielle ( )
 Donner le Polynôme;
 Début
  N= la taille de Polynôme ;
  Pour i=1 à N-1 :
       Matrice Compagnon (1, i)= -Polynôme(i)/polynôme(n);
       Pour j=2 à N-1 :
         Matrice Compagnon (j, i)= 0;
         j=j+1 ;
       Fin;
      Si (i<n-1) Alors
        Matrice Compagnon (i+1, i) = 1;
      i=i+1;
 Fin;
  Racines = valeurs propres (Matrice Compagnon) ;
  Retourner Racines.
Fin.
```

2.3. Environnement du travail

Pour réaliser notre programme de recherche les racines Polynomiales, nous avons utilisé un lap-top ayant les caractéristiques suivantes :

- Fabricant : HP
- Modèle : Personal Computer Probook 4530s
- Processeur : Intel ® Core™ i3-2350M CPU @ 2.30GHz
- Mémoire vive : 4,00 Go DDR III
- Type de système : Système d'exploitation 32bits
- Disque dur : 500 Go.

2.4. Présentation de Matlab

MATLAB est une abréviation de *Matrix LABoratory* [4]. Écrit à l'origine, en Fortran, par *C. Moler*, MATLAB était destiné à faciliter l'accès au logiciel matriciel développé dans les projets LINPACK et EISPACK. La version actuelle, écrite en C par the MathWorks Inc., existe en version professionnelle et en version étudiant. Sa disponibilité est assurée sur plusieurs plates-formes : Sun, Bull, HP, IBM, compatibles PC (DOS, Unix ou Windows), Macintoch, iMac et plusieurs machines parallèles.

MATLAB est un environnement puissant, complet et facile à utiliser destiné au calcul scientifique. Il apporte aux ingénieurs, chercheurs et à tout scientifique un système interactif intégrant calcul numérique et visualisation. C'est un environnement performant, ouvert et programmable qui permet de remarquables gains de productivité et de créativité.

MATLAB est un environnement complet, ouvert et extensible pour le calcul et la visualisation. Il dispose de plusieurs centaines (voire milliers, selon les versions et les modules optionnels autour du noyau Matlab) de fonctions mathématiques, scientifiques et techniques. L'approche matricielle de MATLAB permet de traiter les données sans aucune limitation de taille et de réaliser des calculs numériques et symboliques de façon fiable et rapide. Grâce aux fonctions graphiques de MATLAB, il devient très facile de modifier interactivement les différents paramètres des graphiques pour les adapter selon nos souhaits.

L'approche ouverte de MATLAB permet de construire un outil sur mesure. On peut inspecter le code source et les algorithmes des bibliothèques de fonctions (Toolboxes), modifier des fonctions existantes et ajouter d'autres.

MATLAB possède son propre langage, intuitif et naturel qui permet des gains de temps de CPU spectaculaires par rapport à des langages comme le C, le TurboPascal et le Fortran. Avec MATLAB, on peut faire des liaisons de façon dynamique, à des programmes C ou Fortran, échanger des données avec d'autres applications (via la DDE : MATLAB serveur ou client) ou utiliser MATLAB comme moteur d'analyse et de visualisation.

MATLAB comprend aussi un ensemble d'outils spécifiques à des domaines, appelés Toolboxes (ou Boîtes à Outils). Indispensables à la plupart des utilisateurs, les Boîtes à Outils sont des collections de fonctions qui étendent l'environnement MATLAB pour résoudre des catégories spécifiques de problèmes. Les domaines couverts sont très variés et comprennent notamment le traitement du signal, l'automatique, l'identification de systèmes, les réseaux de neurones, la logique floue, le calcul de structure, les statistiques, etc.

MATLAB fait également partie d'un ensemble d'outils intégrés dédiés au Traitement du Signal. En complément du noyau de calcul MATLAB, l'environnement comprend des modules optionnels qui sont parfaitement intégrés à l'ensemble :

- une vaste gamme de bibliothèques de fonctions spécialisées (*Toolboxes*).

- *Simulink*, un environnement puissant de modélisation basée sur les *schémas-blocs* et de simulation de systèmes dynamiques linéaires et non linéaires.
- Des bibliothèques de blocs *Simulink* spécialisés (*Blocksets*).
- D'autres modules dont un *Compilateur*, un *générateur* de *code C*, et un *accélérateur*.
- Un ensemble d'outils intégrés dédiés au Traitement du Signal : le *DSP Workshop*.

2.4.1. Particularités de MATLAB

MATLAB permet le travail interactif soit en mode commande, soit en mode programmation ; tout en ayant toujours la possibilité de faire des visualisations graphiques. Considéré comme un des meilleurs langages de programmations (C ou Fortran), MATLAB possède les particularités suivantes par rapport à ces langages :

- programmation facile,
- continuité parmi les valeurs entières, réelles et complexes,
- gamme étendue des nombres et leurs précisions,
- bibliothèque mathématique très compréhensive,
- outil graphique qui inclut les fonctions d'interface graphique et les utilitaires,
- possibilité de liaison avec les autres langages classiques de programmation (C ou Fortran).

Dans MATLAB, aucune déclaration n'est à effectuer sur les nombres. En effet, il n'existe pas de distinction entre les nombres entiers, les nombres réels, les nombres complexes et la simple ou double précision. Cette caractéristique rend le mode de programmation très facile et très rapide. En Fortran par exemple, une subroutine est presque nécessaire pour chaque variable simple ou double précision, entière, réelle ou complexe. Dans MATLAB, aucune nécessité n'est demandée pour la séparation de ces variables.

La bibliothèque des fonctions mathématiques dans MATLAB donne des analyses mathématiques très simples. En effet, l'utilisateur peut exécuter dans le mode commande n'importe quelle fonction mathématique se trouvant dans la bibliothèque sans avoir à recourir à la programmation.

Pour l'interface graphique, des représentations scientifiques et même artistiques des objets peuvent être créées sur l'écran en utilisant les expressions mathématiques. Les graphiques sur MATLAB sont simples et attirent l'attention des utilisateurs, vu les possibilités importantes offertes par ce logiciel.

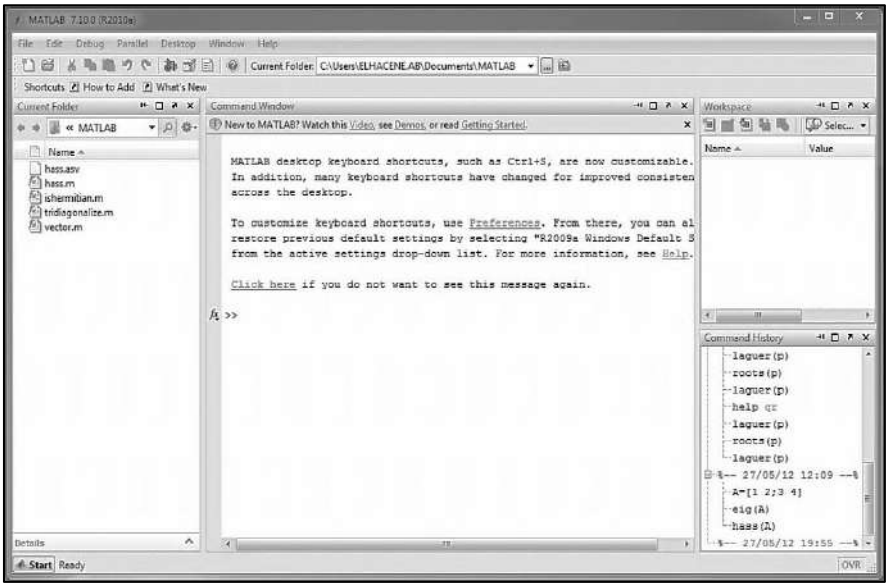


Figure 2.1: L'interface graphique Matlab

2.4.2. L'application Matlab

Nous avons développé une application sous Matlab figure (2.2), dont le but est de résoudre le problème des racines polynomiales réelles ou complexes avec les deux méthodes précédentes.

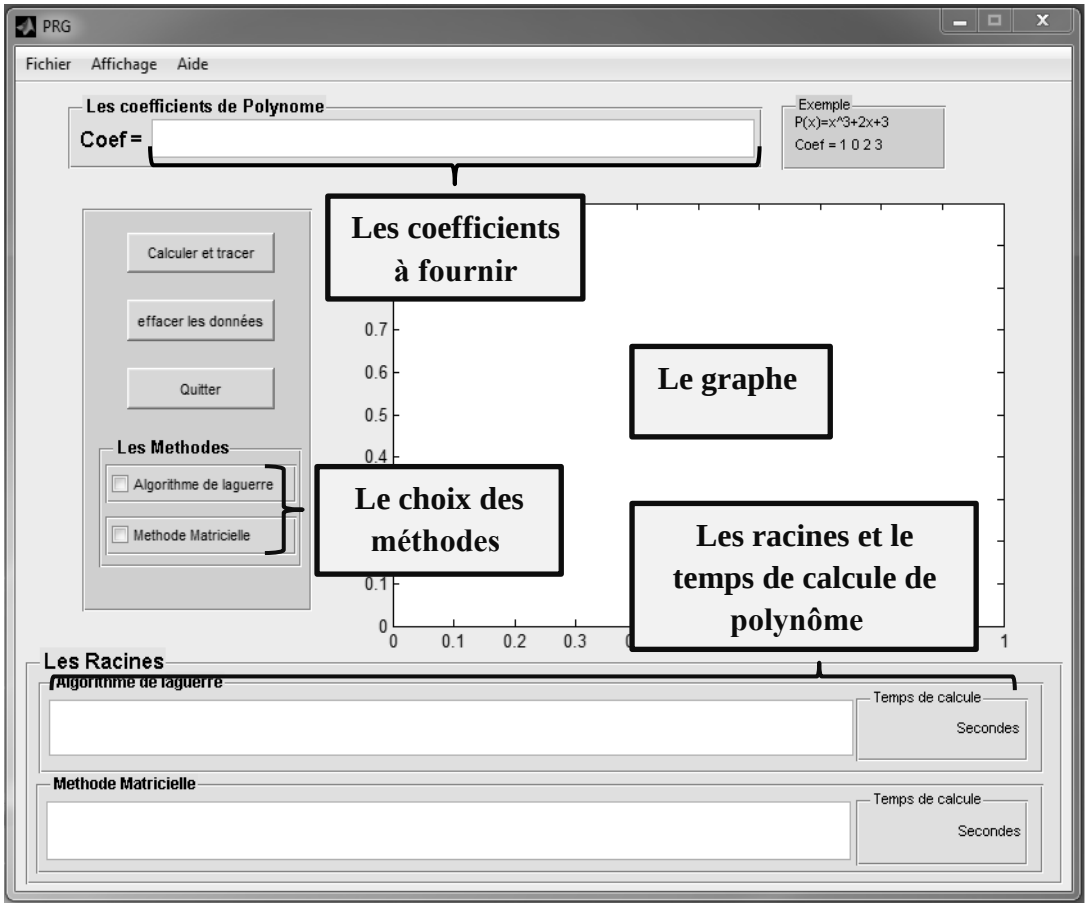


Figure 2.2: L'interface graphique de l'application

Les entrées de cette application sont les coefficients du Polynôme dans la zone Coef. Exemple : pour chercher les zéros du polynôme $P(x) = 2x^4 + x^3 - 3x^2 + \frac{1}{2}x^1 + 2$, taper seulement les coefficients du polynôme : Coef= 2 1 -3 0.5 2, et pour lancer la recherche de racines, choisir l'une des deux méthodes ou les deux à la fois.

2.5. Présentation de LabVIEW

LabVIEW (*Laboratory Virtual Instrument Engineering Workbench*) est un langage de programmation graphique qui utilise des icônes au lieu de lignes de texte pour créer des applications [5]. À l'inverse des langages de programmation textuels, où les instructions déterminent l'exécution du programme, LabVIEW utilise une programmation par flux, où le flux de données détermine l'exécution.

Dans LabVIEW, une interface utilisateur appelée instrument virtuel, est réalisé à l'aide d'un ensemble d'outils et d'objets. On pout ensuite ajouter du code en utilisant les représentations graphiques de fonctions pour commander les objets de la face-avant. Le diagramme contient ce code. D'une certaine manière, le diagramme ressemble à un organigramme.

Vous pouvez utiliser plusieurs add-on d'outils logiciel pour développer des applications spécialisées. Tous les toolsets s'intègrent de façon transparente dans LabVIEW.

LabVIEW est pleinement intégrée pour la communication avec le matériel tel que GPIB, VXI, PXI, RS-232, RS-485, et de commande d'acquisition de données, vision et appareils de commande. LabVIEW possède également des fonctions intégrées pour connexion de votre application à Internet, en utilisant le serveur web de LabVIEW et standards de logiciels tels que TCP/IP.

L'utilisation de LabVIEW, peut passer par des applications compilées à 32 bits donnant rapidité d'exécution, nécessaire pour l'acquisition des données, test, mesure, et des solutions de contrôle. On peut également créer des exécutables et les bibliothèques partagées, comme les DLL.

LabVIEW contient des bibliothèques pour la collecte de données, l'analyse, la présentation et le stockage. Il inclut également des outils de développement. On peut définir des points d'arrêt, animer l'exécution d'un programme, afin de faciliter son débogage et son développement.

LabVIEW fournit aussi de nombreux mécanismes pour la connexion de code externe ou par logiciel dll. En outre, de nombreux outils sont disponibles pour un large éventail de besoins applicatifs.

2.5.1. Utilisation de LabVIEW

LabVIEW permet de construire ses propres solutions scientifiques pour systèmes d'ingénierie.

LabVIEW offre la flexibilité et les performances d'un puissant langage de programmation sans difficulté de connexion.

LabVIEW offre aux utilisateurs, un moyen plus rapide de programmation en instrumentation, acquisition de données, et des systèmes de contrôle.

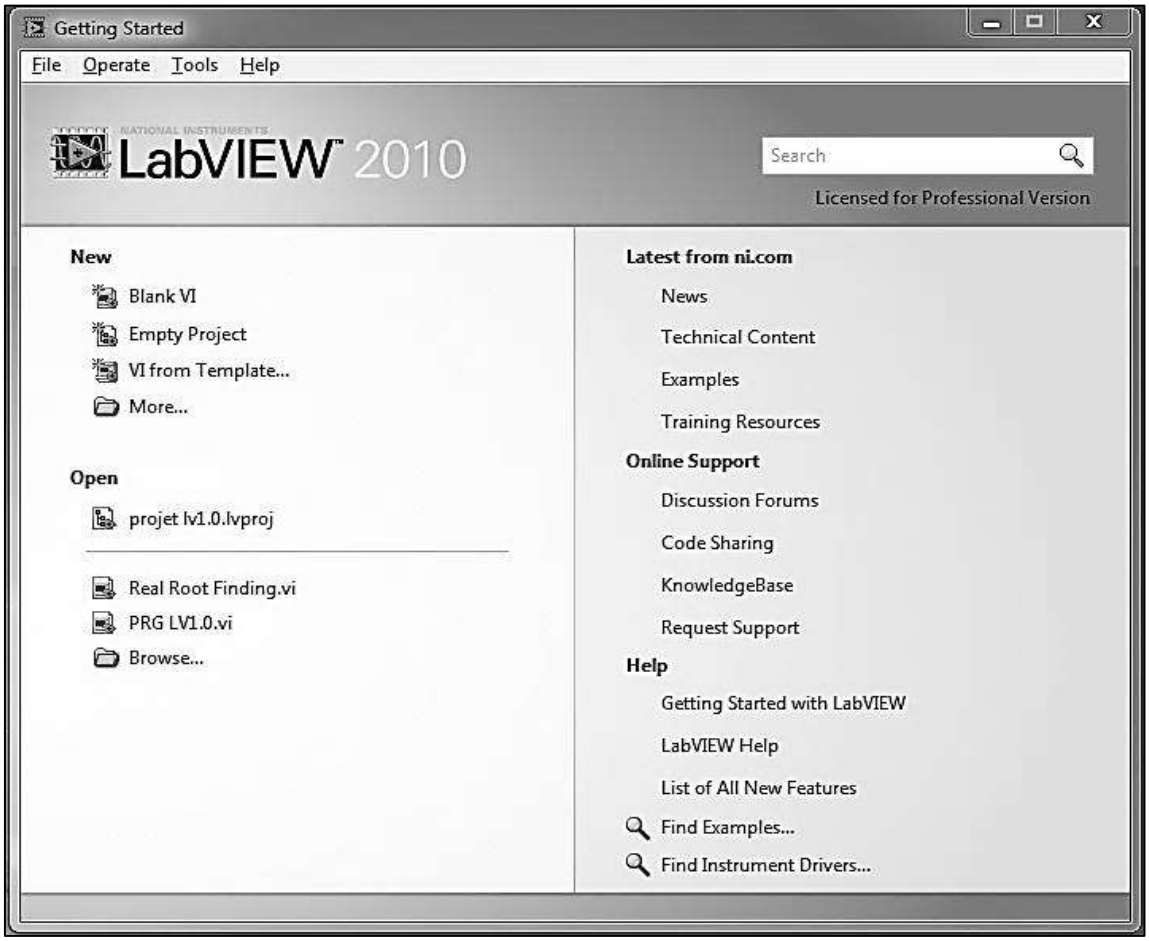


Figure 2.3: L'interface Graphique LabVIEW

2.5.2. L'application LabVIEW

Nous avons développé une application sous LabVIEW Pour la recherche des racines polynomiales, comme à l'application Matlab.

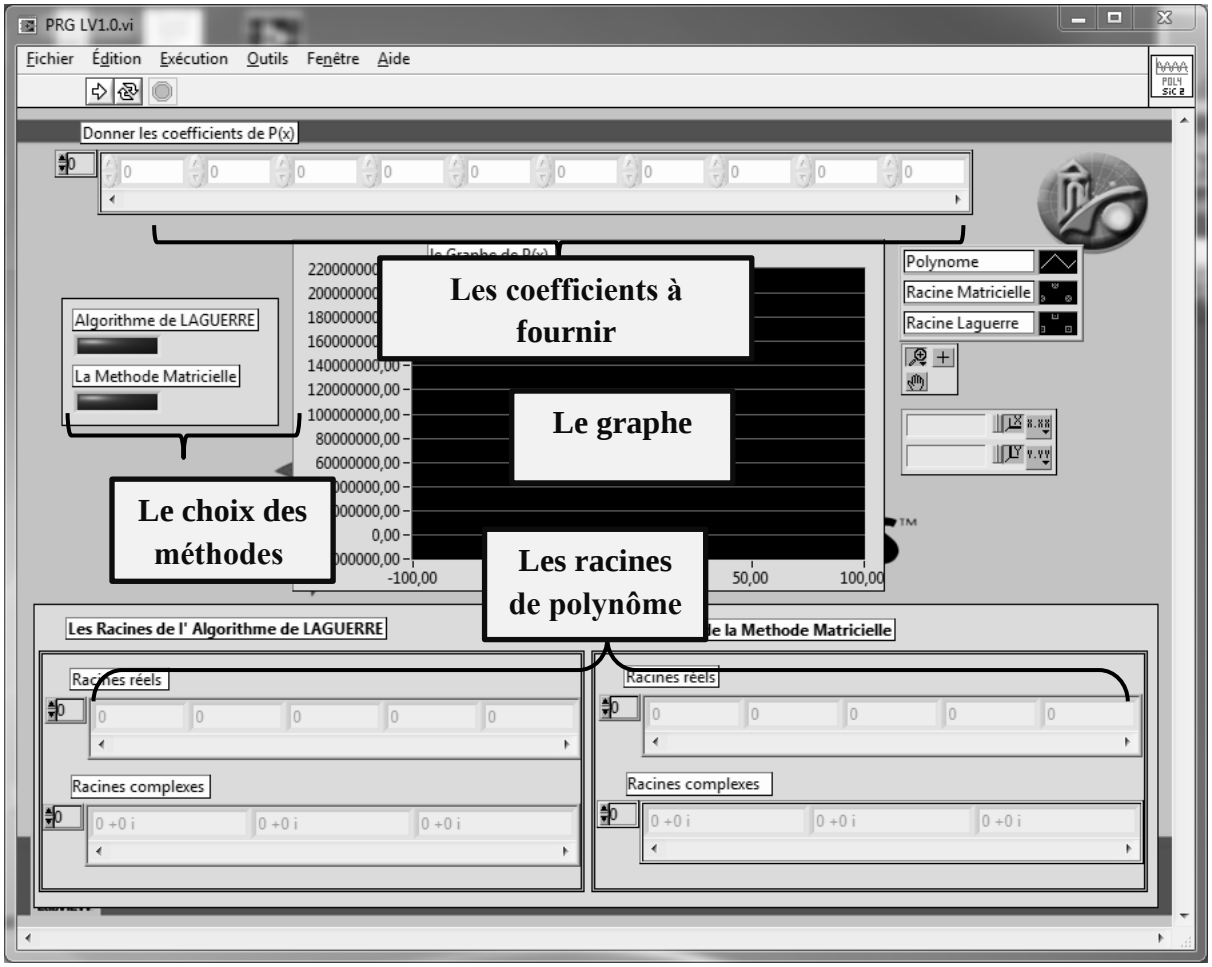


Figure 2.4: L'interface Graphique de l'application

Les entrées de cette application sont les coefficients du Polynôme dans le champ *Donner les Coefficients de P(x).*

Exemple : pour chercher les zéros du polynôme $P(x) = 2x^4 + x^3 - 3x^2 + \frac{1}{2}x^1 + 2$, taper seulement les coefficients de polynôme : *Coefficients* = 2 1 -3 0.5 2, et pour lancer la recherche, choisir l'une des deux méthodes ou les deux à la fois.

2.6. Comparaison des différents essais

Les tableaux 2.1 à 2.4 illustrent quelques résultats fournis après factorisation de quatre polynômes fixés au choix.

Deux polynômes de degré 10 et deux de degré 12 sont choisis telle que les types de racines polynomiales sont soit réelles ou complexes.

Les racines réelles sont pour deux Polynômes de degrés différents, de même, les deux autres polynômes fournissent des racines de type complexe.

Ces tableaux représentent une comparaison des résultats obtenus à partir des fonctions prédéfinis des deux logiciels Matlab et LabVIEW, et notre travail, en termes d'algorithmes de Laguerre et matriciel, écrits aussi sous Matlab et LabVIEW.

	Fonction roots (Matlab/LabVIEW)	Méthode de Laguerre	Méthode Matricielle
Polynôme	$P(x) = x^{10} - 9x^9 - 124x^8 + 1026x^7 + 5289x^6 - 39141x^5 - 81526x^4 + 585324x^3 + 198760x^2 - 2685600x + 2016000$		
Matlab	10.0000 8.0000 5.0000 4.0000 2.0000 1.0000 -3.0000 -5.0000 -6.0000 -7.0000	10 8 5 4 2 1 -3 -5 -6 -7	10 8 5 4 2 1 -3 -5 -6 -7
Temps d'exécution (Matlab)	**0,0002001 Secondes**	**0,00070467 Secondes**	**0,00011481 Secondes**
LabVIEW	9,99999 8,00003 4,99987 4,00013 1,99998 1 -2,99999 -5,00008 -5,9999 -7,00004	9,99999 8,00003 4,99987 4,00013 1,99998 1 -2,99999 -5,00008 -5,9999 -7,00004	9,99999 8,00003 4,99987 4,00013 1,99998 1 -2,99999 -5,00008 -5,9999 -7,00004

Tableau 2.1 : polynôme de degré 10 avec racines réelles

	Fonction roots (Matlab/LabVIEW)	Méthode de Laguerre	Méthode Matricielle
Polynôme	$P(x) = 2x^{10} - 1x^8 + 1x^6 - 0.5x^4 + 3x^3 + 1x^2 + 2$		
Matlab	1.0717 + 0.4407i 1.0717 - 0.4407i 0.4466 + 0.7799i 0.4466 - 0.7799i 0.1647 + 0.8527i 0.1647 - 0.8527i -0.7226 + 0.8695i -0.7226 - 0.8695i -0.9604 + 0.1846i -0.9604 - 0.1846i	1.0717+0.4407i 1.0717-0.4407i 0.44663+0.77989i 0.44663-0.77989i 0.16468+0.85268i 0.16468-0.85268i -0.72262-0.86952i -0.72262+0.86952i -0.96039-0.18465i -0.96039+0.18465i	1.0717+0.4407i 1.0717-0.4407i 0.44663+0.77989i 0.44663-0.77989i 0.16468+0.85268i 0.16468-0.85268i -0.72262+0.86952i -0.72262-0.86952i -0.96039+0.18465i -0.96039-0.18465i
Temps d'exécution (Matlab)	**0,0002113 Secondes**	**0,00108628 Secondes**	**0,00011504 Secondes**
LabVIEW	1,0717 +0,440705 i 1,0717 -0,440705 i 0,446628 +0,779887 i 0,446628 -0,779887 i 0,164677 +0,852678 i 0,164677 -0,852678 i -0,722616 +0,869522 i -0,722616 -0,869522 i -0,960392 +0,184646 i -0,960392 -0,184646 i	1,0717 +0,440705 I 1,0717 -0,440705 i 0,446628 +0,779887 i 0,446628 -0,779887 i 0,164677 +0,852678 i 0,164677 -0,852678 i -0,722616 +0,869522 i -0,722616 -0,869522 i -0,960391 +0,184645 i -0,960392 -0,184644 i	1,0717 +0,440705 i 1,0717 -0,440705 i 0,446628 +0,779887 i 0,446628 -0,779887 i 0,164677 +0,852678 i 0,164677 -0,852678 i -0,722616 +0,869522 i -0,722616 -0,869522 i -0,960392 +0,184646 i -0,960392 -0,184646 i

Tableau 2.2 : polynôme de degré 10 avec racines complexes

	Fonction roots (Matlab/LabVIEW)	Méthode de Laguerre	Méthode Matricielle
Polynôme	$P(x) = 1x^{12} - 12\,x^{11} - 97\,x^{10} + 1398x^{9} + 2211x^{8} - 55008x^{7} + 35897x^{6} + 829902x^{5} - 1557212x^{4} - 3281880x^{3} + 10072800x^{2} - 6048000\,x$		
Matlab	10.0000 8.0000 5.0000 4.0000 3.0000 2.0000 1.0000 0 -3.0000 -5.0000 -6.0000 -7.0000	10 8 5 4 3 2 1 0 -3 -5 -6 -7	10 8 5 4 3 2 1 0 -3 -5 -6 -7
Temps d'exécution (Matlab)	**0,0003315 Secondes**	**0,00157346 Secondes**	**0,00021557 Secondes**
LabVIEW	10 8,00001 4,99993 4,00013 2,99991 2,00002 0,999999 0 -3 -4,99999 -6,00001 -7	10 8 5 4 3 2 1 1,02815E-16 ≃ 0 -3 -5 -6 -7	10 8 5 4 3 2 1 0 -3 -5 -6 -7

Tableau 2.3 : polynôme de degré 12 avec racines réelles

	Fonction roots (Matlab/LabVIEW)	Méthode de Laguerre	Méthode Matricielle
Polynôme	$P(x) = 2x^{12} + 3x^{10} + 0.5x^9 + 1x^6 - 0.5x^4 + 3x^3 + 1x^2 + 2$		
Matlab	0.9294 + 0.3638i 0.9294 - 0.3638i 0.4448 + 0.7895i 0.4448 - 0.7895i 0.2192 + 0.8720i 0.2192 - 0.8720i -0.0105 + 1.3115i -0.0105 - 1.3115i -0.7160 + 0.7729i -0.7160 - 0.7729i -0.8669 + 0.2011i -0.8669 - 0.2011i	0.92941+0.3638i 0.92941-0.3638i 0.44483+0.78948i 0.44483-0.78948i 0.21916+0.87204i 0.21916-0.87204i -0.010534+1.3115i -0.010534-1.3115i -0.71601+0.77293i -0.71601-0.77293i -0.86686+0.20113i -0.86686-0.20113i	0.92941+0.3638i 0.92941-0.3638i 0.44483+0.78948i 0.44483-0.78948i 0.21916+0.87204i 0.21916-0.87204i -0.010534+1.3115i -0.010534-1.3115i -0.71601+0.77293i -0.71601-0.77293i -0.86686+0.20113i -0.86686-0.20113i
Temps d'exécution (Matlab)	**0,0002135 Secondes**	**0,00137265 Secondes**	**0,00013819 Secondes**
LabVIEW	0,929414 +0,363797 I 0,929414 -0,363797 i 0,444829 +0,78948 i 0,444829 -0,78948 i 0,219157 +0,872037 i 0,219157 -0,872037 i -0,0105335 +1,31148 i -0,0105335 -1,31148 i -0,716009 +0,772933 i -0,716009 -0,772933 i -0,866858 +0,201128 i -0,866858 -0,201128 i	0,929414 +0,363797 I 0,929414 -0,363797 i 0,444829 +0,78948 i 0,444829 -0,78948 i 0,219157 +0,872037 i 0,219157 -0,872037 i -0,0105335 +1,31148 i -0,0105335 -1,31148 i -0,716009 +0,772933 i -0,716009 -0,772933 i -0,866858 +0,201129 i -0,866858 -0,201129 i	0,929414 +0,363797 i 0,929414 -0,363797 i 0,444829 +0,78948 i 0,444829 -0,78948 i 0,219157 +0,872037 i 0,219157 -0,872037 i -0,0105335 +1,31148 i -0,0105335 -1,31148 i -0,716009 +0,772933 i -0,716009 -0,772933 i -0,866858 +0,201128 i -0,866858 -0,201128 i

Tableau 2.4 : polynôme de degré 12 avec racines complexes

2.6.1. Discussion des résultats

Les racines des quatre polynômes pris comme exemple, trouvées par les fonctions prédéfinies propres à Matlab et LabVIEW coïncident avec celles fournies par nos algorithmes basées sur la méthode de Laguerre et celle matricielle.

a) Tableau 2.1

Le polynôme de degré 10 à racines réelles, est traité bonne précision sous Matlab. Le temps écoulé pour l'exécution par la méthode matricielle est meilleur que les autres méthodes. La fonction roots de Matlab arrive en deuxième position, puis la méthode de Laguerre.

b) Tableau 2.2

Le polynôme de degré 10 à racines complexes, est traité avec la même précision sous Matlab et sous LabVIEW. La précision obtenue par notre travail est meilleure que celle de Matlab à 10^{-2}. Le temps d'exécution de la méthode matricielle est toujours plus efficace que les autres méthodes.

c) Tableau 2.3

Le polynôme de degré 12 à racines réelles, est traité avec la même précision sauf pour la fonction roots de LabVIEW où les résultats sont peu précis.

Le temps écoulé reste du même ordre pour les toutes méthodes sous Matlab.

d) Tableau 2.4

Le polynôme de degré 12 à racines complexes, est traité avec les résultats suivants:

- Matlab : la précision des fonctions programmées est meilleure que celle de la fonction roots.

  En revanche, le temps d'exécution est meilleur chez la méthode matricielle puis roots puis Laguerre.

- Labview : les résultats des trois méthodes sont les mêmes avec une bonne précision par rapport à Matlab.

Nous notons aussi que notre travail offre de bons résultats au niveau de la précision.

Le tableau 2.5 présente la relation entre l'ordre du polynôme et le temps d'exécution des trois méthodes sous Matlab.

Les polynômes choisis ont des racines mélangées et sont les suivant :

$P(x) = -x^{14} - x^{13} + 3x^{11} + 5x^{8} - 3x^{5} + 3x^{3}$

$P(x) = 4x^{14} + x^{13} - 1.5\,x^{8} + 4.75x^{6} + 3x^{5} + 0.75x^{3} + 1$

$P(x) = -7x^{16} + x^{15} - 4x^{13} - x^{10} + x^{8} + 6x^{7} + 4x^{6}$

$P(x) = 2x^{16} + x^{15} - 4x^{13} - x^{10} + 5\,x^{8} + x^{7} + 3x^{6} + 0.75x^{2} + 2x$

$P(x) = 2x^{18} + x^{17} - 4x^{15} - x^{12} + x^{10} + 5\,x^{8} + x^{7} + 3x^{6} + 0.75x^{2} + 2x + 0.3$

$P(x) = 0.5x^{18} + x^{17} + x^{16} + 2x^{15} + 4x^{12} + 5x^{11} + 6x^{10} - 5\,x^{9} - 5\,x^{7} - 3x^{6} - 0.5x^{5} + 1$

$$P(x) = x^{20} + 2x^{19} + 5x^{18} + 4x^{16} + 0.5x^{15} - 0.5x^{11} - x^9 + 4x^8 + 5x^6 + 6x^5 + 0.8x^4 - x^3 - 3x^2$$

$$P(x) = x^{20} + 2x^{19} + 3x^{18} + 4x^{16} + 5x^{15} - 2x^{13} + 5x^{11} - 0.5x^{10} + x^9 + x^8 + 0.4x^7 + x^3 + 2x^2 + 3x + 0.15$$

	Roots	Laguerre	Matricielle
degré 10	0,0002057	0,00089547	0,00011492
degré 12	0,0002725	0,00147306	0,00017688
degré 14	0,0002035	0,00170065	0,00013837
degré 16	0,0002545	0,00217659	0,00017512
degré 18	0,0003423	0,00240822	0,00020423
degré 20	0,0003513	0,00285512	0,00022949

Tableau 2.5 : le temps d'exécution des trois méthodes sous Matlab en fonction de l'ordre du polynôme(en secondes)

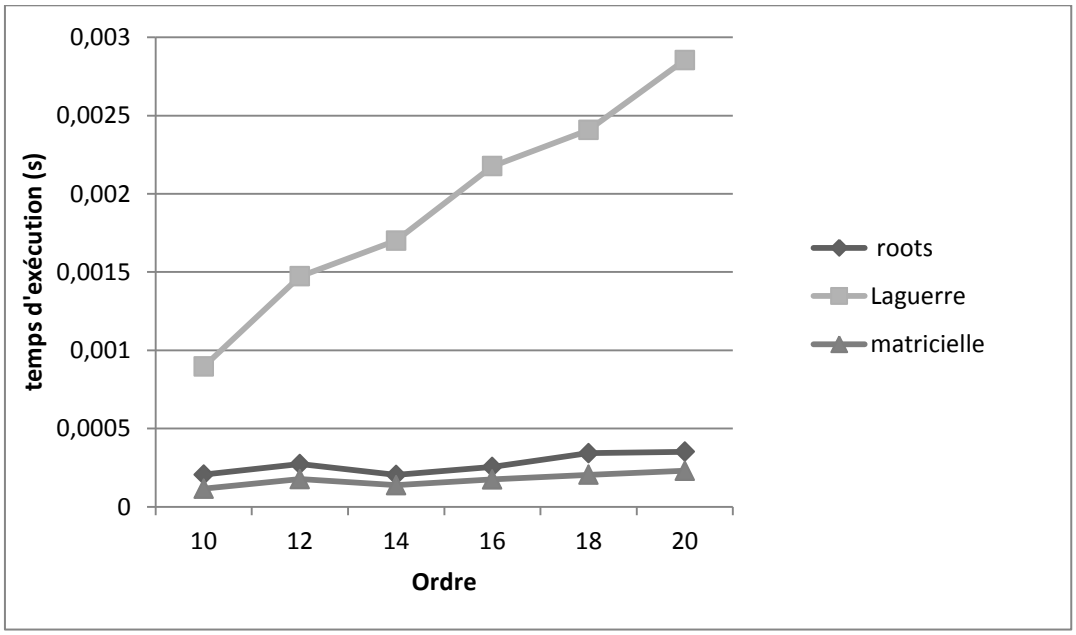


Figure 2.5 : graphe du temps d'exécution selon le degré

Le graphe de la figure 2.5 représente les valeurs illustrées au tableau 2.5. Le graphe nous permet de distinguer la bonne performance la méthode matricielle par rapport à la fonction roots de Matlab.

La méthode de Laguerre prend plus de temps dépendant de l'ordre du polynôme. Car cette méthode est itérative et utilise plusieurs fonctions.

2.6.2. Remarques

a) L'outil LabVIEW est limité au niveau de calcul temporaire, au millier de la seconde (ms). Ça implique une difficulté de détermination du temps écoulé au traitement de chaque opération.

b) Les valeurs du temps d'exécution des quatre polynômes d'exemple (voir tableaux 2.1, 2.2, 2.3, 2.4) sont déterminées comme moyenne de dix exécutions de chaque polynôme.
c) Le graphe de la figure 2.5 comporte 5 degrés de polynômes. Pour chaque degré, nous choisissons deux polynômes où chaque polynôme est exécuté pendant cinq essais. Le temps est pris en moyenne.

2.7. Conclusion

Dans ce chapitre nous avons discuté notre travail implémenté sur Matlab et LabVIEW, en ne considérant par l'aspect d'exécution en temps réel.

Les différentes méthodes développées présentent une même précision que celles prédéfinies dans les logiciels Matlab et LabVIEW.

Au chapitre suivant, nous implémentons nos algorithmes sur différents DSP, en prêtant plus d'attention à l'exécution en temps réel.

CHAPITRE 3:

IMPLEMENTATION DES METHODES LAGURRE ET MATRICIELLE SUR DIFFERENTS DSP

3.1. Introduction

Certaines applications nécessitent des calculs en temps réel sur les échantillons d'entrée, c'est le cas par exemple pour :

- le filtrage numérique,
- la reconnaissance de la parole,
- le contrôle (moteur, etc.).

Pour ces applications on utilise des processeurs spécifiques dédiés au traitement du signal appelés DSP (Digital Signal Processor).

Les DSPs sont similaires aux microprocesseurs d'usage général à 1'exception d'être mieux adaptés pour effectuer les opérations d'addition et de multiplication et ceci d'une manière plus optimale. Les DSPs ont aussi l'avantage de la faible consommation d'énergie et du coût moindre.

La plupart du temps, on retrouve le DSP dans une chaîne de traitement du signal, Figure 3.1 composée essentiellement d'un CAN (Convertisseur Analogique Numérique) pour numériser le signal, et d'un CNA (convertisseur Numérique Analogique) pour restituer le signal après traitement.

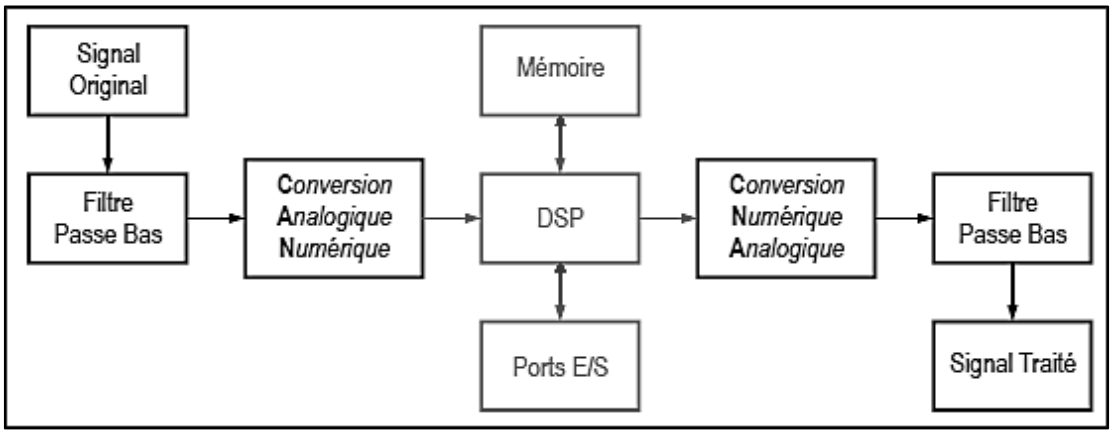


Figure 3.1: Chaîne de traitement à base d'un DSP

Vu d'ensemble, on peut classer les DSPs en deux grandes familles:

- les DSPs à virgule fixe: qu'on retrouve dans la plupart des applications où le coût est un facteur important (ils sont moins chers que les DSP à virgule flottante), cependant ils sont plus compliqués à programmer.
- les DSPs à virgule flottante : sont plus faciles et plus souples à programmer que les DSPs à virgule fixe, dans ce cas on fait intervenir une mantisse et un exposant et on bénéficie d'une dynamique plus importante. On retrouve ce type de DSP dans les applications Audio.

Dans le cadre de ce projet, les DSPs utilisées sont, le TMS320C5510 et le TMS320C6713 de la firme Texas Instrument [6].

3.2. Classification des DSPs

Il est impossible d'effectuer une classification « définitive » des DSP, car chaque constructeur met sur le marché tous les ans, un nouveau composant qui surclasse les anciens ou les concurrents par la puissance de calcul, la rapidité (gestion du pipeline et fréquence d'horloge), le nombre de registres, des Timers, de ports série…

3.2.1. Virgule fixe ou flottante

Un point essentiel des DSP est la représentation des nombres (les données) qu'ils peuvent manipuler. Il est possible de distinguer deux familles :

- Les DSP à virgule fixe : les données sont représentées comme étant des nombres fractionnaires à virgule fixe, (exemple -1.0 à +1.0), ou comme des entiers classiques. La représentation de ces nombres fractionnaires s'appuie la méthode du « complément à deux ». L'avantage de cette représentation (qui n'est qu'une convention des informaticiens) est d'éviter la débordements en cas de multiplications successives.

- Les DSP à virgule flottante : les données sont représentées en utilisant une mantisse et un exposant. La représentation de ces nombres s'effectue selon la formule suivante : $n = mantisse * 2^{exposant}$. Généralement, la mantisse est un nombre fractionnaire (-1.0 à +1.0), et l'exposant est un entier indiquant la place de la virgule en base 2 (c'est le même mécanisme qu'en base 10).

3.2.2. Les DSPs à virgules flottantes

Les DSP à virgule flottante sont plus souples et plus faciles à programmer que les DSP à virgule fixe. Un DSP comme le TMS320C30 [7] manipule des nombres formés avec une mantisse de 24 bits et un exposant de 8 bits (taille de la donnée en mémoire : 32 bits). Les valeurs intermédiaires des calculs sont mémorisées dans des registres avec un format de 32 bits de mantisse et un exposant de 8 bits (taille du registre : 32 + 8 bits supplémentaires). La dynamique disponible est très grande, elle va de $-1 * 2^{128}$ à $(1 - 2^{-23}) * 2^{127}$, toutefois la résolution reste limitée à 24 bits au mieux. Outre les nombres fractionnaires, ce DSP sait également manipuler les entiers avec une précision de 32 bits.

La très grande dynamique proposée par les DSP à virgule flottante permet virtuellement de ne pas se soucier des limites des résultats calculés lors de la conception d'un programme. Cet avantage a cependant un prix, à savoir qu'un système basé sur un DSP à virgule flottante a un coût de fabrication supérieur par rapport à un système basé sur DSP à virgule fixe. La puce d'un DSP à virgule flottante nécessite à la fois une surface de silicium plus importante (cœur plus complexe), et un nombre de broches supérieur, car la mémoire externe est elle aussi au format 32 bits. Le système revient donc plus cher (exemple : 2 x 32 broches ne serait-ce que pour les bus de données externes avec une architecture Harvard de base).

Un DSP à virgule flottante est plutôt adapté (sans être impératif) à des applications dans lesquelles :

- Les coefficients varient dans le temps (exemple: les filtres adaptatifs),
- Le signal et les coefficients ont besoin d'une grande dynamique,

- La structure mémoire est importante (exemple: traitement d'image),
- La précision est recherchée sur toute une gamme dynamique importante (exemple : traitements audiophoniques de qualité professionnelle).

De part, leurs facilités de programmation, ils peuvent également se justifier dans des projets ou le temps et la facilité de développement sont des facteurs importants. On les trouve également dans des produits de faible volume de production, pour lesquels le prix du DSP n'est pas significatif.

3.2.3. Les DSP à virgules fixes

Un DSP à virgule fixe est un peu plus compliqué à programmer qu'un DSP à virgule flottante.

Poids des bits	MSB			LSB	
	-2^3	2^2	2^1	2^0	
Valeurs	-8	4	2	1	
	0	1	0	1	= 4+1 = +5
	1	1	0	1	= -8+4+1 = -3
Le moins positif	0	0	0	1	= 1 = +1
Le plus positif	0	1	1	1	= 4+2+1 = +7
Le moins négatif	1	1	1	1	= -8+4+2+1 = -1
Le plus négatif	1	0	0	0	= -8 =-8

Figure 3.2 : Représentation des nombres entiers codés sur 4 bits en complément à 2

Poids des bits	MSB			LSB	
	-2^0	2^{-1}	2^{-2}	2^{-3}	
Valeurs	-1	½	¼	1/8	
	0	1	0	1	= 0.5+0.125 = +0.625
	1	1	0	1	= -1+0.5+0.125 = -0.375
Le moins positif	0	0	0	1	= 1 = +0.125
Le plus positif	0	1	1	1	= 0.5+0.25+0.1251= +0.875
Le moins négatif	1	1	1	1	= -1+0.5+0.25+0.125=-0.125
Le plus négatif	1	0	0	0	= -1 =-1

Figure 3.3: Représentation des nombres fractionnaires codés sur 4bits en complément à 2

Dans un DSP à virgule fixe typique comme le TMS320C25, les nombres sont codés sur 16 bits.

Toutefois, sur ce DSP, les calculs sont effectués avec des accumulateurs de 32 bits. Lorsque les résultats doivent être stockés en mémoire, les 16 bits les moins significatifs sont perdus. Ceci permet de limiter les erreurs d'arrondis cumulatives. Il est toujours possible de stocker séparément en mémoire, les 16 bits faibles puis les 16 bits forts s'il n'y a plus de registres libres lors d'une étape de calcul.

Cette particularité n'est pas toujours disponible sur tous les DSP. Dans ce cas, les calculs requérant absolument une double précision sont réalisés en chaînant deux à deux des

instructions spéciales manipulant des données 16 bits en simple précision, au détriment du temps d'exécution.

La précision des calculs est un point critique des DSP à virgule fixe, car le concepteur de programmes doit rester vigilant à chaque étape d'un calcul. Il doit rechercher la plus grande dynamique possible (c.à.d. exploiter au mieux la gamme des nombres disponibles), pour conserver une bonne précision des calculs, tout en évitant autant que faire ce peu les débordements du ou des accumulateurs. Les bits supplémentaires des accumulateurs (les bits de garde) prévus à cet effet permettent de réduire cette contrainte.

Les programmeurs contournent les limites des DSP à virgule fixe en déterminant à l'avance, et avec soins, la précision et la dynamique nécessaire (par méthode analytique ou avec des outils de simulation) pour réaliser leurs projets.

Il est également possible d'effectuer des opérations en virgule flottante dans un DSP à virgule fixe par le biais de routines logicielles adéquates. Cette approche est néanmoins pénalisante en temps d'exécution, même sur un DSP à virgule fixe très rapide.

En termes de rapidité, les DSP à virgule fixe se placent d'ordinaire devant leurs homologues à virgule flottante, ce qui constitue un critère de choix important.

Les DSP à virgule fixe sont les plus utilisés, car ils sont moins chers que les DSP à virgule flottantes. On les trouve dans tous les produits de grande diffusion où le coût est un facteur important. Il peut cependant exister des exceptions, certains DSP à virgule fixe se présentant comme des micro contrôleurs perfectionnés plus chers qu'un DSP à virgule flottante de base.

Voyons le cas des DSP fabriqués par Texas Instruments (désignés par TMS) et Analog Devices (désignés par ADSP). Le classement du tableau 3.1 est effectué selon le nombre de bits du bus de données et le temps d'exécution d'un cycle, puis d'une opération complexe, comme la transformée de Fourier rapide à 1024 points de calcul.

Nom	Critère de choix	Données avec virgule fixe ou flottante	Durée d'une instruction élémentaire [ns]	Durée du calcul FFT [us]
ADSP2105	Faible cout	16 bits-Fixe	100	3.46
TMS320C2X	Faible cout	16 bits-Fixe	80	9.01
ADSP2101	Haute performance	16 bits-Fixe	60	2.07
TMS320C5X	Haute performance	16 bits-Fixe	35	2.97
ADSP2199X	Haute performance	16 bits-Fixe	6	0.4
TMS320C67X	Haute performance	32 bits-Flottante	5	0.4
ADSP 21010	Faible cout	32 bits-Flottante	80	1.54
TMS320C3X	Faible cout	32 bits-Flottante	50	3.08
ADSP21020	Haute performance	32 bits-Flottante	40	0.77
TMS320C4X	Haute performance	32 bits-Flottante	40	1.55

Tableau 3.1 : Comparaison entre diverses catégories de DSP

- Les TMS320Clx, C2x, C5x ou ADSP2105, 21 01 sont à 16 bits à virgule fixe.
- Les TMS320Clx sont utilisés pour le contrôle des disques durs dans les ordinateurs.

- Les TMS320C2x ou ADSP-2105 servent au fonctionnement des fax.
- Les TMS320C5x ou ADSP-21 01 sont utilisés dans les modems.
- Les TMS320C3x, C4x ou ADSP-21010, 21020 sont à 32 bits à virgule flottante.
- Les TMS320C3x ou ADSP-21010 sont utilisés pour les systèmes Hi-Fi, à synthèse vocale, et dans les processeurs graphiques à 3 dimensions.
- Les TMS320C4x, TMS320C6x ou ADSP-21020 sont conçus pour le fonctionnement en parallèle, avec d'autres systèmes processeurs (applications réalité virtuelle et la reconnaissance d'images).

3.2.4. Performance des DSP

Plus que pour un microprocesseur classique, les performances d'un DSP conditionnent son domaine d'application.

La plupart des DSP sont particulièrement destinés à des applications « temps réel » et spécialisées, c'est à dire des applications où le temps de traitement est bien sûr primordial, mais où la diversité des événements à traiter n'est pas notablement importante. De ce point de vue, l'approche DSP s'apparente plus à une étude « électronique » visant à réaliser une ou des fonctions de traitements de signal, que d'une approche informatique temps réel et/ou multitâche traditionnelle.

Il existe cependant des applications où le DSP assure à la fois des fonctions de traitements numériques du signal et les fonctions générales d'un microprocesseur au cœur d'un système informatique classique.

Dans tous les cas, les performances du DSP sont critiques. Le concepteur d'un système à base de DSP doit évaluer d'une part la « puissance » nécessaire pour réaliser les traitements numériques voulus, et d'autre part les performances des DSP disponibles pour réaliser son application.

3.2.5. Puissance de calcul d'un DSP

C'est un autre critère de classification des DSP. Cette puissance de calcul dépend de la rapidité de l'exécution des instructions, et donc de l'horloge. Dans un DSP, le MAC (multiplicateur et accumulateur) calcule le produit de deux entrées codées sur N bits, dans un temps « record » de 7ns à 150ns. Un seul cycle d'horloge. La multiplication est obtenue de manière asynchrone. Le résultat est chargé dans un accumulateur à 2 x N bits. L'utilisateur choisit de garder seulement les N bits de poids fort en simple précision, et effectue donc une troncature ou alors l'ensemble du résultat en double précision. Bien entendu, si le DSP est à virgule flottante, l'effet de la troncature est moins gênant.

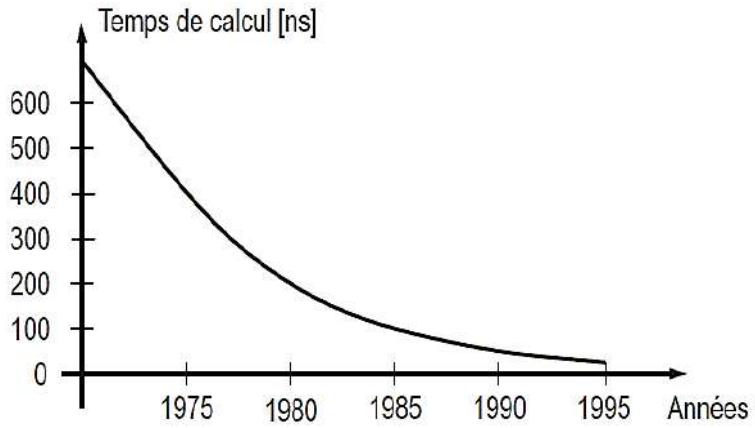


Figure 3.4: Evolution du temps d'exécution d'une opération MAC selon Texas Instruments

La méthode classique pour évaluer les performances d'un DSP est de se baser sur sa vitesse d'exécution. Encore faut-il trouver une bonne définition de ce qu'est la vitesse d'exécution, ce qui n'est pas forcément simple.

Cette méthode de base consiste donc à compter le nombre d'instructions effectuées par seconde. Un obstacle apparaît alors, car une instruction ne signifie pas forcément la même chose d'une famille de DSP à l'autre. Le Tableau 1-6 résume les principales définitions en usage.

Acronyme anglais	Définition
MFLOPS Million FLoating-point Operations Per Second	Mesure le nombre d'opérations à virgule flottante (multiplications, additions, soustractions, etc.) que le DSP à virgule flottante peut réaliser en une seconde.
MOPS Million Operations Per Second	Mesure le nombre total d'opérations que le DSP peut effectuer en une seconde. Par opération, il faut comprendre non seulement le traitement des données, mais également les accès DMA, les transferts de données, les opérations d'E/S, etc. Cette définition mesure donc les performances globales d'un DSP plutôt que ses seules capacités de calcul.
MIPS Million Instructions Per Second	Mesure le nombre de codes machine (instructions) que le DSP peut effectuer en une seconde. Bien que cette mesure s'applique à tous de DSP, le MFOPS est préféré dans le cas d'un DSP a virgule flottante.
MBPS Mega-Bytes Per Second	Mesure la largeur de bande d'un bus particulier ou d'un dispositif d'E/S, c.-à-d. son taux de transfert.

Tableau 3.2: Définitions des unités les plus courantes de mesures des performances des DSP

Une autre méthode consiste à définir une fois pour toute une opération de référence comme un « MAC », puisqu'il s'agit d'une fonction commune à tous les DSP. Il ne reste plus qu'à compter le nombre de « MAC » par seconde.

Cependant cette définition n'apporte pas beaucoup d'informations sur les performances des DSP modernes. En effet, un « MAC » est exécuté en un seul cycle. Sachant que sur les DSP récents la plupart des instructions sont également exécutées en un cycle, cela revient donc à mesurer les MIPS du DSP. Il faut également tenir compte du fait que certains DSP en font plus dans un seul « MAC » (nombre, format et taille des opérandes traités) que d'autres.

3.3. Description du DSP TMS320C6713

Le TMS320C6713 est un processeur à virgule flottante de la famille TMS320C6x, c'est une nouvelle génération de DSPs développé par Texas Instrument, qui utilise une version améliorée de l'architecture VLIW (Very Long Instruction Word), le TMS320C6713 est composé de parties principales suivantes : l'unité centrale (CPU), les mémoires et les périphériques, le tout est relié par un bus interne, Figure 3.5.

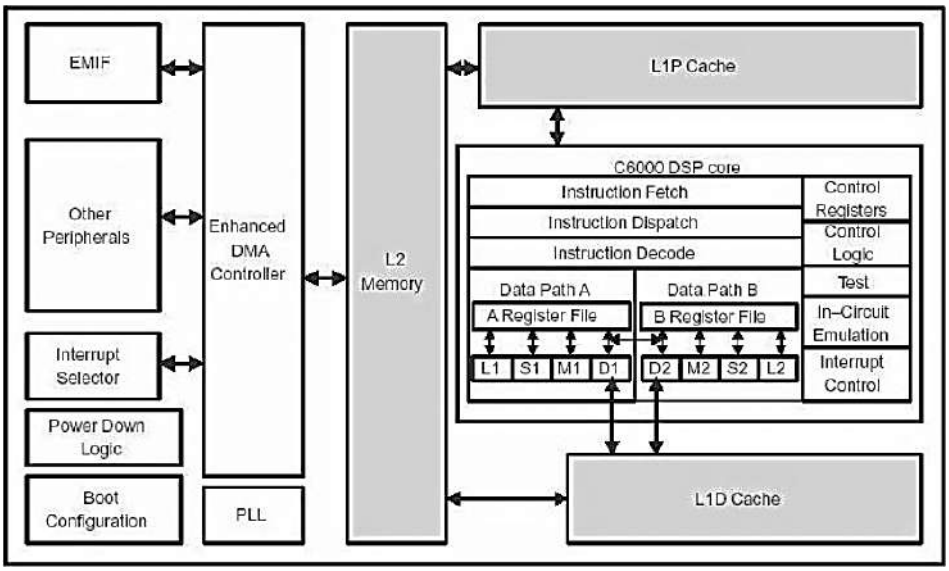


Figure 3.5 : Structure interne du DSP TMS320C6713

L'unité centrale est composée principalement des éléments suivants:

L'unité de contrôle du programme : contient trois composantes (Instruction Fetch, Instruction Dispatch, Instruction Decode) nécessaires à l'exécution d'instructions en pipeline. C'est un moyen qui permet de gagner du temps lors de l'exécution en effectuant plusieurs tâches en même temps en les parallélisant.

Les unités fonctionnelles : l'architecture VLIW permet le traitement par paquets de 8 instructions pouvant être traitées par les unités fonctionnelles (huit unités: 6 UALs et 2 Multiplieurs). Ces unités sont organisées en deux blocs d'unités (Data Path A, Data Path B), chaque bloc contient les éléments suivants :

- l'unité (.M): Utilisée pour les opérations de multiplications.
- l'unité (.L) : Utilisée pour les opérations logiques et arithmétiques.
- l'unité (.S) : Utilisée pour effectuer un branchement ou pour la manipulation des bits.

- l'unité (.D) : Utilisée pour rechercher ou stocker (load/store) des données, également utilisée pour les opérations arithmétiques.
- Les registres : on compte plusieurs registres dont 32 registres à usages générale à 32 bits, 16 registres pour l'unité A (A0 jusqu'à A15), et 16 pour l'unité B (B0 jusqu'à B15).
  On compte aussi 13 registres à usage spéciale, appelés registres de contrôle dont huit registres dédiés à la gestion et le contrôle d'interruptions.

3.4. Description du DSP TMS320C5510

Le TMS320C5510 est un processeur à virgule fixe de la famille TMS320C5x.

Le processeur C5510 est conçu pour faible consommation d'énergie, des performances optimales, et densité de code élevée. En outre, le code source de C5510 est compatible avec le C54x. Cela réduit considérablement le coût de la migration des systèmes basés sur C54x populaires aux C55x. Certaines fonctionnalités essentielles du C5510 sont énumérées ci-dessous:

- source-code compatible avec tous les TMS320C54x.
- 64 octet de file d'attente, tampon d'instructions qui fonctionne comme un cache du programme et met en œuvre efficacement les opérations régulières de blocs.
- Deux bits par 17 bits d'unités MAC peut se multiplient et s'accumulent dans un cycle de signal.
- Une unité ALU de 40 bits exécute une opération arithmétique de haute précision et des ALU supplémentaires à 16 bits pour des opérations logiques et effectuant des opérations arithmétiques parallèles à l'ALU principal.
- Quatre accumulateurs de 40 bits pour stocker les résultats de calcul afin de réduire l'accès à la mémoire.
- Huit registres auxiliaires étendus pour adressage des données, plus quatre registres de données temporaires pour faciliter de prescriptions en matière de traitement des données.
- Mode circulaire d'adressage qui prend jusqu'à cinq tampons circulaires.
- Une Instruction unique de répétition par bloc facilitant la construction de boucle.

Le CPU C55x se compose de quatre unités de traitement: une unité tampon d'instructions (UI), une unité de flux de programme (PU), une unité de flux de données d'adresse (UA), et une unité de calcul de données (UA). Ces unités sont reliées à 12 différents bus d'adresses et de données [8].figure 3.6

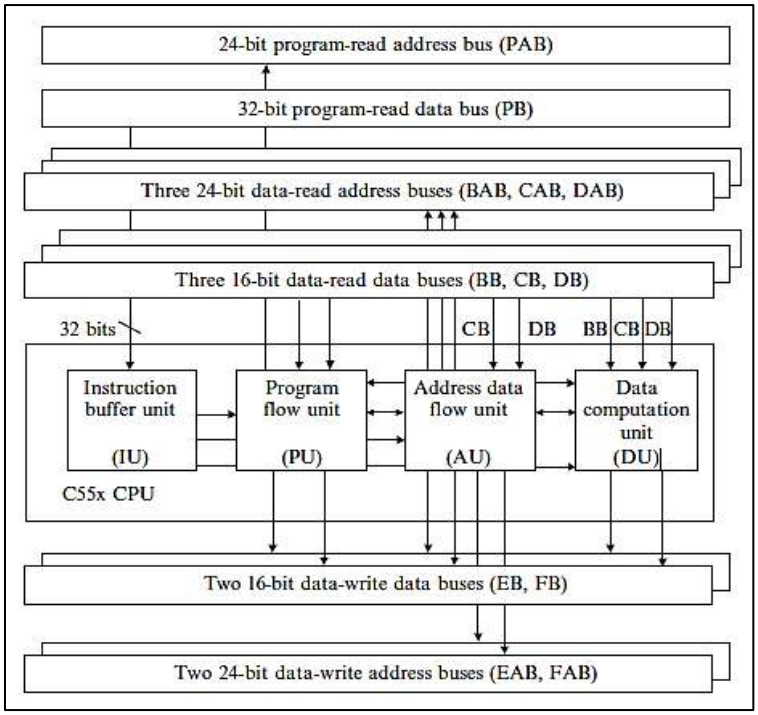


Figure 3.6 : Bloc-diagramme du TMS320C55x CPU

3.5. L’environnement intégré de développement (IDE)

Pour les besoins de notre application, nous avons utilisé 1'outil de développement de Texas Instrument appelé CCS (Code Composer Studio).

3.5.1. CCS

L'évaluation des performances des algorithmes sur le DSP est effectuée en utilisant le CCS, dont le fonctionnement est illustré par la Figure 3.7.

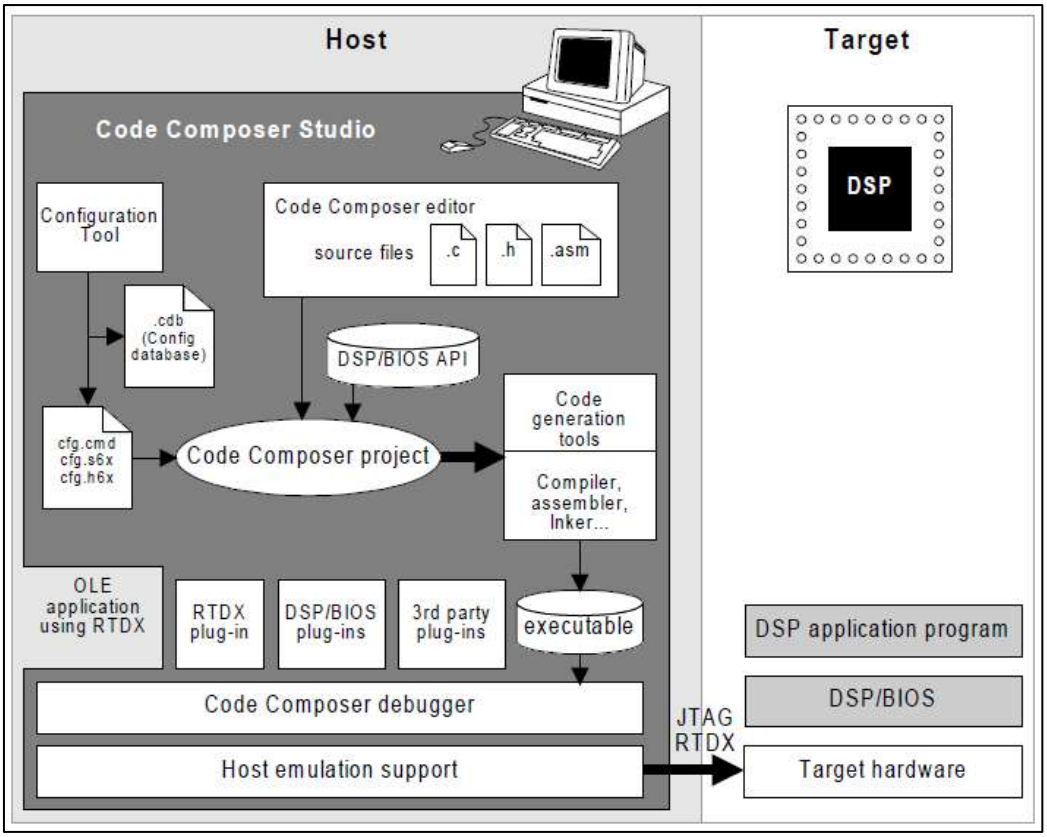


Figure 3.7 : Structure du système du développement

Code Composer Studio est un logiciel de développement qui:

Permet l'édition, la compilation, et le débogage des programmes destinés au DSP. Il inclut aussi :

- Les outils de génération du code qui sont le compilateur, l'assembleur et l'éditeur de lien. Le compilateur C/C++ permet de compiler le programme source en c et le convertit en assembleur en asm. L'assembleur reçoit le fichier asm et le convertit en langage machine ou fichier objet (obj). Enfin l'éditeur de liens (linker) combine les fichiers objet et les fichiers librairies et le fichier .cmd pour produire un fichier exécutable avec une extension .out, c'est ce fichier qui sera chargé sur le processeur du DSP pour être exécuter.
- le DSP/BIOS : est un outil d'analyse en temps réel. Pour s'en servir, on doit créer un fichier de configuration cdb, où seront définis les objets utilisés par l'outil DSP/BIOS, voir Figure 3.8. Ce fichier permet aussi de faciliter l'organisation de la mémoire et la gestion du vecteur des interruptions, en offrant la possibilité de les faire sur un environnement graphique via la section de gestion de la mémoire MEM (Memory Section Manager), et via HWI (Hardware Interrupt Service Routine Manager) pour les interruptions.

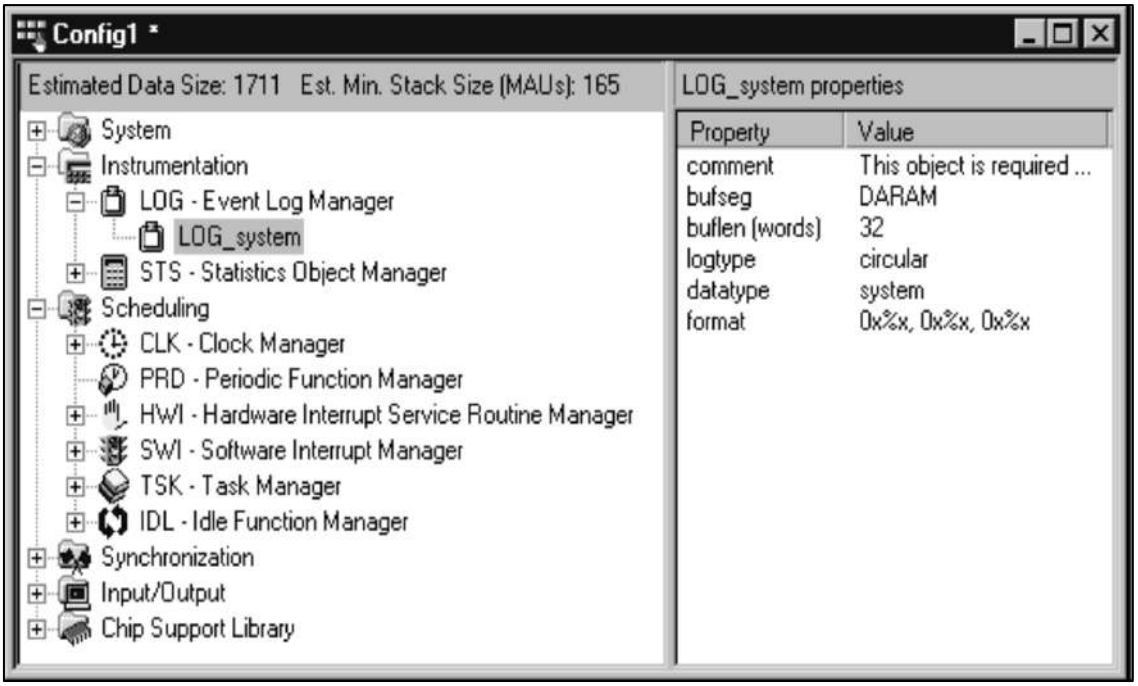


Figure 3.8 :L'outil DSP/BIOS

- Lien JTAG (Joint Team Action Group) et le RTDX (Real Time Data Exchange), Figure 3.8, ce dernier permet un échange de données en temps réel entre l'hôte (PC par exemple) et la destination (la carte DSK dans notre cas). Il permet aussi l'analyse et la visualisation des données au cours de l'exécution du programme. Le lien JTAG est utilisé pour atteindre l'émulateur (qui se trouve à l'intérieure du DSP), c'est ce dernier qui permet au CCS de contrôler en temps réel l'exécution du programme.

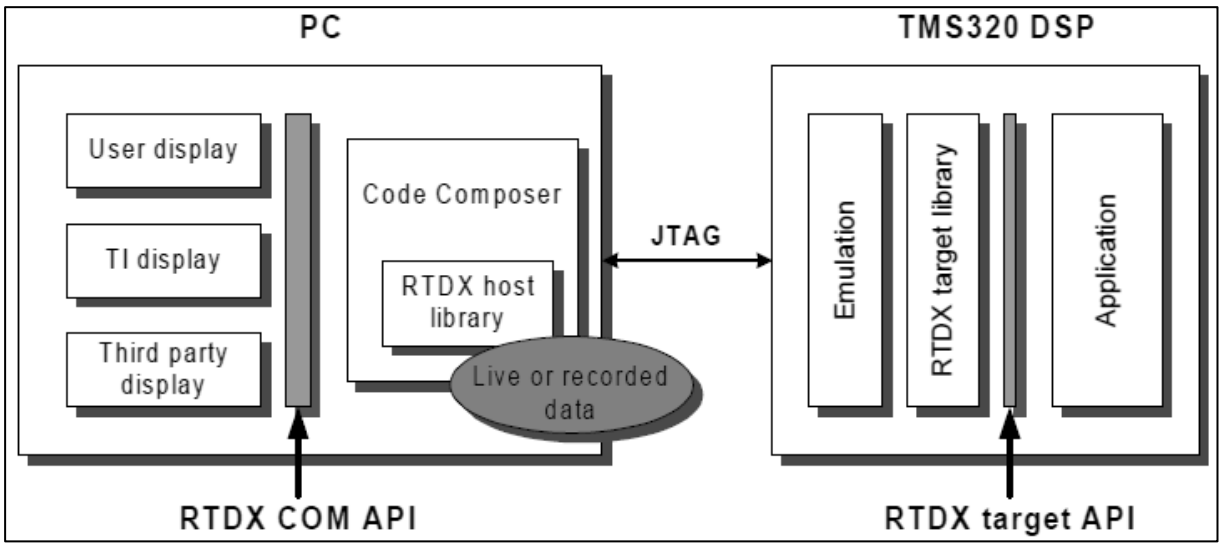


Figure 3.9 : L'outil JTAG

- un simulateur intégré : en effet, CCS offre la possibilité de tester des programmes pour DSPs, sans utiliser la carte DSK à l'aide d'un simulateur intégré.

3.6. Implémentation sous DSP

Nous avons essayés d'implémentés les algorithmes des deux méthodes Laguerre et matricielle vue précédemment, sous les deux DSP TMS32C6713 et TMS32C5510, en langage C.
Nous avons trouvés des problèmes dans les nombres complexes, qui ne sont pas des types prédéfinis dans le langage C, à cet effet nous avons programmes une nouvelle bibliothèque (comlex.h) celle où l'on a définis les nouveau type et sa manipulation.

3.7. Comparaison des différents essais

Les tableaux 3.3 à 3.6 illustrent quelques résultats fournis pour quatre polynômes de choix.
Deux polynômes de degré 10 et deux de degré 12, dont les types de racines polynomiales sont soit réels ou complexes.
Les racines réelles sont pour deux polynômes de degrés différents, ainsi que pour les deux autres polynômes qui sont à racines complexes.
Ces tableaux représentent une comparaison des résultats sur deux familles de DSP TMS320C6713 et TMS320C5510, du point de vue précision des résultats et nombre de cycle d'horloge consommé.

	Méthode de Laguerre	**Méthode Matricielle**
Polynôme	$\boldsymbol{P(x) = x^{10} - 9\,x^9 - 124x^8 + 1026\,x^7 + 5289\,x^6 - 39141\,x^5 - 81526\,x^4 + 585324\,x^3 + 198760\,x^2 - 2685600\,x + 2016000}$	
TMS320C6713	10.000002 7.999998 5.000001 3.999999 2.000000 1.000000 -3.000000 -4.999996 -6.000014 -6.999990	10.000054 7.999946 5.000009 3.999999 2.000000 1.000000 -3.000000 -5.000000 -5.999999 -7.000003
Temps d'exécution	**293422 Cycles**	**333206 Cycles**
TMS320C5510	10.000002 7.999998 5.000001 3.999999 2.000000 1.000000 -3.000000 -4.999996 -6.000014 -6.999990	10.000120 7.999897 5.000015 3.999989 2.000000 1.000000 -3.000000 -4.999999 -6.000004 -7.000017
Temps d'exécution	**1011418 Cycles**	**2390130 Cycles**

Tableau 3.3 : polynôme de degré 10 à racines réelles sous différent DSP

	Méthode de Laguerre	**Méthode Matricielle**
Polynôme	$P(x) = 2x^{10} - 1x^8 + 1x^6 - 0.5x^4 + 3x^3 + 1x^2 + 2$	
TMS320C6713	1.071703 + 0.440705i 1.071703 - 0.440705i 0.446628 + 0.779888i 0.446628 - 0.779887i 0.164677 + 0.852678i 0.164677 - 0.852678i -0.722616 + 0.869522i -0.722616 - 0.869522i -0.960392 + 0.184646i -0.960392 - 0.184646i	1.071703 +0.440705 i 1.071703 - 0.440705 i 0.446628 + 0.779887 i 0.446628 - 0.779887 i 0.164677 + 0.852679 i 0.164677 - 0.852679 i -0.722616 + 0.869522 i -0.722616 - 0.869522 i -0.960392 + 0.184646 i -0.960392 - 0.184646 i
Temps d'exécution	**383637 Cycles**	**478203 Cycles**
TMS320C5510	1.071703 + 0.440705i 1.071703 - 0.440705i 0.446628 + 0.779888i 0.446628 - 0.779887i 0.164677 + 0.852678i 0.164677 - 0.852678i -0.722616 + 0.869522i -0.722616 - 0.869522i -0.960392 + 0.184646i -0.960392 - 0.184646i	1.071703 +0.440705 i 1.071703 - 0.440705 i 0.446628 + 0.779887 i 0.446628 - 0.779887 i 0.164677 + 0.852678 i 0.164677 - 0.852678 i -0.722615 + 0.869522 i -0.722615 - 0.869522 i -0.960392 + 0.184646 i -0.960392 - 0.184646 i
Temps d'exécution	**2093245 Cycles**	**3432120 Cycles**

Tableau 3.4 : polynôme de degré 10 à racines complexes sous différents DSP

	Méthode de Laguerre	**Méthode Matricielle**
Polynôme	$P(x) = 1x^{12} - 12x^{11} - 97x^{10} + 1398x^9 + 2211x^8 - 55008x^7 + 35897x^6 + 829902x^5 - 1557212x^4 - 3281880x^3 + 10072800x^2 - 6048000x$	
TMS320C6713	9.999995 8.000007 4.999992 4.000005 3.000001 1.999999 1.000000 0.000000 -3.000000 -5.000005 -5.999991 -7.000003	10.000010 7.999991 5.000021 3.999987 3.000000 2.000001 1.000000 0.000000 -2.999999 -4.999996 -6.000015 -6.999997

Temps d'exécution	**358019 Cycles**	**481271 Cycles**
TMS320C5510	9.999995 8.000007 4.999992 4.000005 3.000001 1.999999 1.000000 0.000000 -3.000000 -5.000005 -5.999991 -7.000003	10.000010 7.999991 5.000021 3.999987 3.000000 2.000001 1.000000 0.000000 -2.999999 -4.999996 -6.000015 -6.999997
Temps d'exécution	**191798 Cycles**	**3068103 Cycles**

Tableau 3.5 : polynôme de degré 12 à racines réelles sous différents DSP

	Méthode de Laguerre	**Méthode Matricielle**
Polynôme	$P(x) = 2x^{12} + 3x^{10} + 0.5x^9 + 1x^6 - 0.5x^4 + 3x^3 + 1x^2 + 2$	
TMS320C6713	0.929414 + 0.363797 i 0.929414 - 0.363797 i 0.444829 + 0.789480 i 0.444829 - 0.789480 i 0.219157 + 0.872037 i 0.219157 - 0.872038 i -0.010534 + 1.311480 i -0.010534 - 1.311480 i -0.716009 + 0.772933 i -0.716009 - 0.772933 i -0.866858 + 0.201128 i -0.866858 - 0.201128 i	0.929414 + 0.363797 i 0.929414 - 0.363797 i 0.444829 + 0.789480 i 0.444829 - 0.789480 i 0.219157 + 0.872038 i 0.219157 - 0.872038 i -0.010534 + 1.311481 i -0.010534 - 1.311481 i -0.716008 + 0.772933 i -0.716008 - 0.772933 i -0.866857 + 0.201128 i -0.866857 - 0.201128 i
Temps d'exécution	**513256 Cycles**	**657287 Cycles**
TMS320C5510	0.929414 + 0.363797 i 0.929414 - 0.363797 i 0.444829 + 0.789480 i 0.444829 - 0.789480 i 0.219157 + 0.872037 i 0.219157 - 0.872038 i -0.010534 + 1.311480 i -0.010534 - 1.311480 i -0.716009 + 0.772933 i -0.716009 - 0.772933 i -0.866858 + 0.201128 i -0.866858 - 0.201128 i	0.929415 + 0.363797 i 0.929415 - 0.363797 i 0.444829 + 0.789480 i 0.444829 - 0.789480 i 0.219157 + 0.872038 i 0.219157 - 0.872038 i -0.010533 + 1.311481 i -0.010533 - 1.311481 i -0.716009 + 0.772933 i -0.716009 - 0.772933 i -0.866859 + 0.201128 i -0.866859 - 0.201128 i
Temps d'exécution	**2787867 Cycles**	**4825901 Cycles**

Tableau 3.6 : polynôme de degré 12 à racines complexes sous différents DSP

3.7.1. discussion des résultats

a) Tableau 3.3 :

Le polynôme est de degré 10 avec racines réelles. Les résultats de la méthode de Laguerre sont plus précis que ceux de la méthode matricielle, et les résultats entre les deux DSP sont identiques avec les deux méthodes.

La méthode Laguerre en terme de temps d'exécution est plus rapide que celle de matricielle, de même, le DSP TMS320C6713 est plus rapide que TMS320C5510**.**

b) Tableau 3.4 :

Le polynôme est de degré 10 avec racines complexes.

La précision des résultats est identique au tableau 3.3.

La méthode de Laguerre reste plus rapide en terme de temps d'exécution que la méthode matricielle.

Le DSP TMS320C6713 est également plus rapide que TMS320C5510.

c) Tableau 3.5 :

Le polynôme est de degré 12 avec racines réelles.

La précision des résultats et la différence en temps d'exécution du tableau 3.5 sont similaires à celles du tableau 3.3.

d) Tableau 3.6 :

Le polynôme est de degré 12 avec racines complexes.

La précision des résultats et la différence en temps d'exécution du tableau 3.6 sont similaires à celles du tableau 3.4.

Le tableau 3.7 présente la relation entre l'ordre du polynôme et le nombre de cycles des deux méthodes avec C6713 et C5510. (avec polynômes a racines mélangées vue précédemment)

	Laguerre C6713	matricielle C6713	Laguerre C5510	matricielle C5510
degré 10	**338529**	**405704**	**1552331**	**2906125**
degré 12	**435637**	**569245**	**1989832**	**3947002**
degré 14	**515543**	**753707**	**2218426**	**5550356**
degré 16	**610381**	**945907**	**3161979**	**6942929**
degré 18	**911850**	**1528270**	**4910156**	**11585151**
degré 20	**1054455**	**1699659**	**5688889**	**12922212**

Tableau 3.7 : le nombre de Cycles selon l'ordre de polynôme

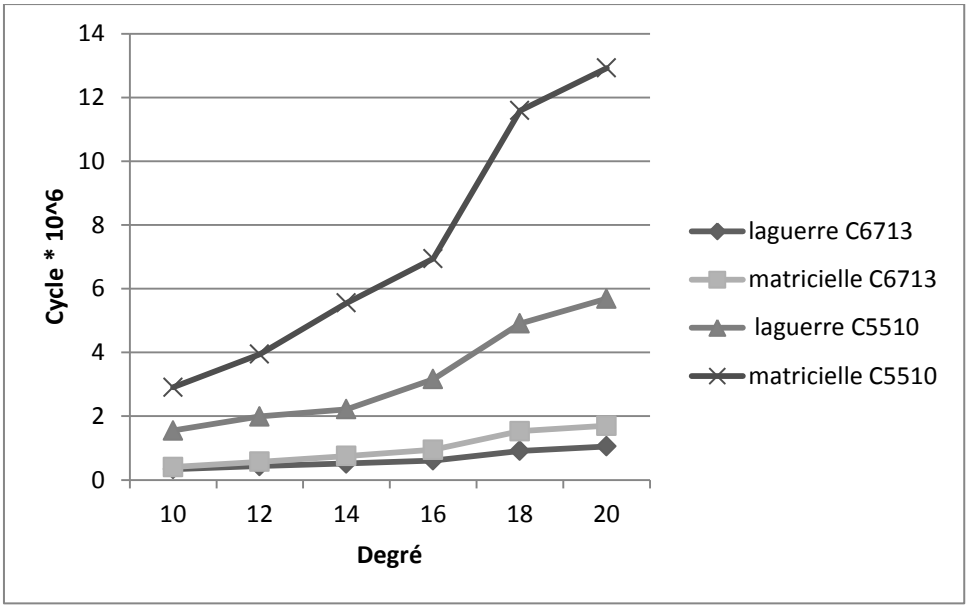


Figure 3.10 : graphe du nombre de cycles en fonction de degré du polynôme

La figure 3.10 représente l'évolution du temps d'exécution en terme du nombre de cycles d'horloge par rapport au degré du polynôme, suivant les deux méthodes, à savoir Laguerre et matricielle.

Les courbes sont paramétrées par le type DSP employé.

On constate que chaque fois le degré augmente, le temps d'exécution augmente également.

3.7.2. Remarques

a) Nous avons essayé de travailler sur l'algorithme de la décomposition QR, illustrée en figure 3.11 pour tirer les valeurs propres de la matrice polynomiale.

Données : A
début
  $H \leftarrow I$
  pour *k=1,…, n-1* **faire**
    $\alpha \leftarrow \sqrt{\sum_{i=k}^{m} A_{ik}^2}$
    $\beta \leftarrow \alpha^2 - \alpha A_{kk}$
    // construction du vecteur v
    $v_k \leftarrow A_{kk} - \alpha$
    pour *i=k+1,…, m* **faire**
      $v_i \leftarrow A_{ik}$
    pour *j=k,…, n* **faire**
      // Construction de A^{k+1}
      $c \leftarrow \frac{1}{\beta} \sum_{i=k}^{m} v_i A_{ij}$
      pour *i=k,…, m* **faire**
        $A_{ij} \leftarrow A_{ij} - c v_i$
    pour *j=1,…, m* **faire**
      // Construction de $H = H_k \cdots H_2 H_1$
      $c \leftarrow \frac{1}{\beta} \sum_{i=k}^{m} v_i H_{ij}$
      pour *i=k,…, m* **faire**
        $H_{ij} \leftarrow H_{ij} - c v_i$
  $Q \leftarrow {}^{t}H$
fin
retourner *Q, R = A*

Figure 3.11 : Algorithme de décomposition QR

Sauf que les résultats obtenus n'ont pas été à la hauteur de nos attentes, car les racines complexes ne figurent pas dans ces résultats.

Nous avons dû utiliser une fonction toute prête, trouvée dans un livre d'analyse numérique, mais même cette dernière ne donne pas des résultats fiables.

b) Le temps d'exécution est calculé en nombre de cycles.
c) Les choix des polynômes sont faits au hasard et peuvent fausser la comparaison.

3.8. Conclusion :

On constate que le DSP TMS320C6713 en virgule flottante est meilleur en terme de temps d'exécution que son homologue, le DSP TMS320C5510 en virgule fixe. La précision des résultats est la même pour les deux types de DSP.

CONCLUSION GENERALE

Le thème de notre étude insinue la résolution d'un problème mathématique lié par exemple à la stabilité d'un système défini par sa fonction de transfert représentée sous forme d'un polynôme d'ordre supérieur, ou bien, tirer les formants de la parole à partir de son polynôme prédicteur, etc…

Ainsi différents cas pratiques reviennent à chercher les racines d'un modèle polynomial, souvent de degré supérieur. Plusieurs solutions existent, mais deux principales sont envisagées dans notre cas : l'une à base d'algorithme de Laguerre et l'autre à base matricielle aboutissant aux valeurs propres du polynôme. Ces deux algorithmes ont été implémentés en langage C sur deux types différents de DSP : TMS320C5510 à virgule fixe, et le TMS320C6713 à virgule flottante. Les résultats ont été comparés et représentés sous forme de tableaux de mesure et graphes de tendance. La vérification des résultats a été confiée à deux logiciels puissants : Matlab et LabVIEW. Les résultats obtenus sont très satisfaisants et peuvent être utiles comme indicateurs de performance pour chacun des deux types de DSPs utilisés : C5x et C6x.

ANNEXE

- **Code sources Matlab**

d) Code sources de l'Algorithme de Laguerre

a. Fonction d'évaluation (`evalpoly.m`)

```
function [p,dp,ddp] = evalpoly(a,x)
n = length(a) - 1;
p = a(1); dp = 0.0; ddp = 0.0;
for i = 1:n
  ddp = ddp*x + 2.0*dp;
  dp = dp*x + p;
  p = p*x + a(i+1);
end
```

b. Fonction de déflation (`deflpoly.m`)

```
function b = deflpoly(a,r)
n = length(a) - 1;
b = zeros(n,1);
b(1) = a(1);
for i = 2:n
  b(i) = a(i) + r*b(i-1);
end
```

c. Fonction de Laguerre(`laguerre.m`)

```
function x = laguerre(P,tol)
x = randn;
n = length(P) - 1;
for i = 1:30
  [p,dp,ddp] = evalpoly(P,x);
  if abs(p) < tol; return;
  end
  g = dp/p; h = g*g - ddp/p;
  f = sqrt((n - 1)*(n*h - g*g));
  if abs(g + f) >= abs(g - f); dx = n/(g + f);
  else dx = n/(g - f);
  end
  x = x - dx;
  if abs(dx) < tol; return;
  end
end
error('Too many iterations in laguerre')
```

d. Fonction Polyroots (`Polyroots.m`)

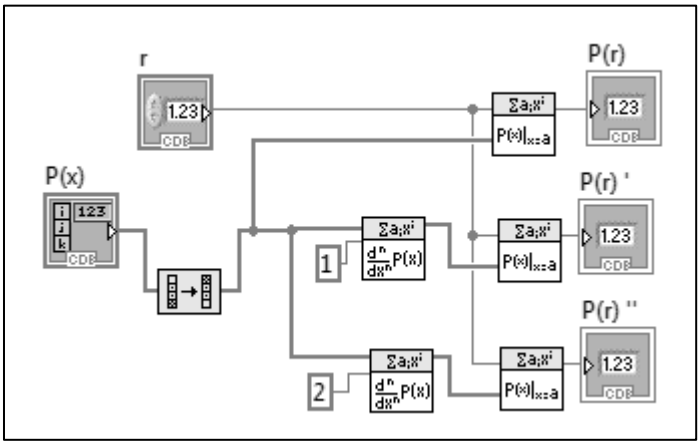
```
function root = polyroots(P)
tol = 1.0e-6;
n = length(P) - 1;
root = zeros(n,1);
for i = 1:n
x = laguerre(P,tol);
if abs(imag(x)) < tol; x = real(x);
end
root(i) = x;
P = deflpoly(P,x); end
```

e) Code sources de la méthode Matricielle (`m_matrice.m`)

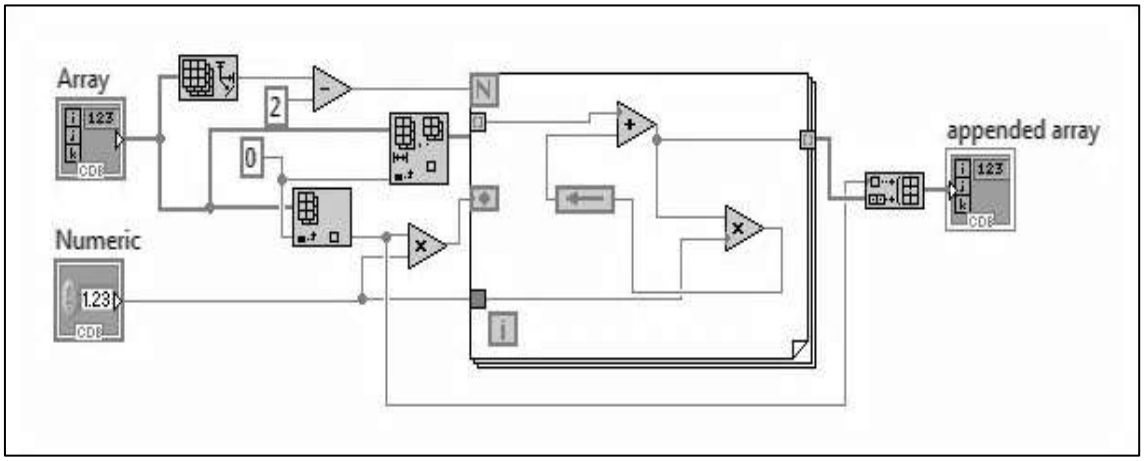
```
function [R]=m_matrice(P)
if length(P)<2
    R=[];
else
n=length(P);
A=diag(ones(n-2,1),-1);
A(1,:)=-P(2:n)./P(1);
R=eig(A);
end
```

- **Blocs diagramme LabVIEW**
  - **Blocs diagramme d'algorithme de Laguerre**

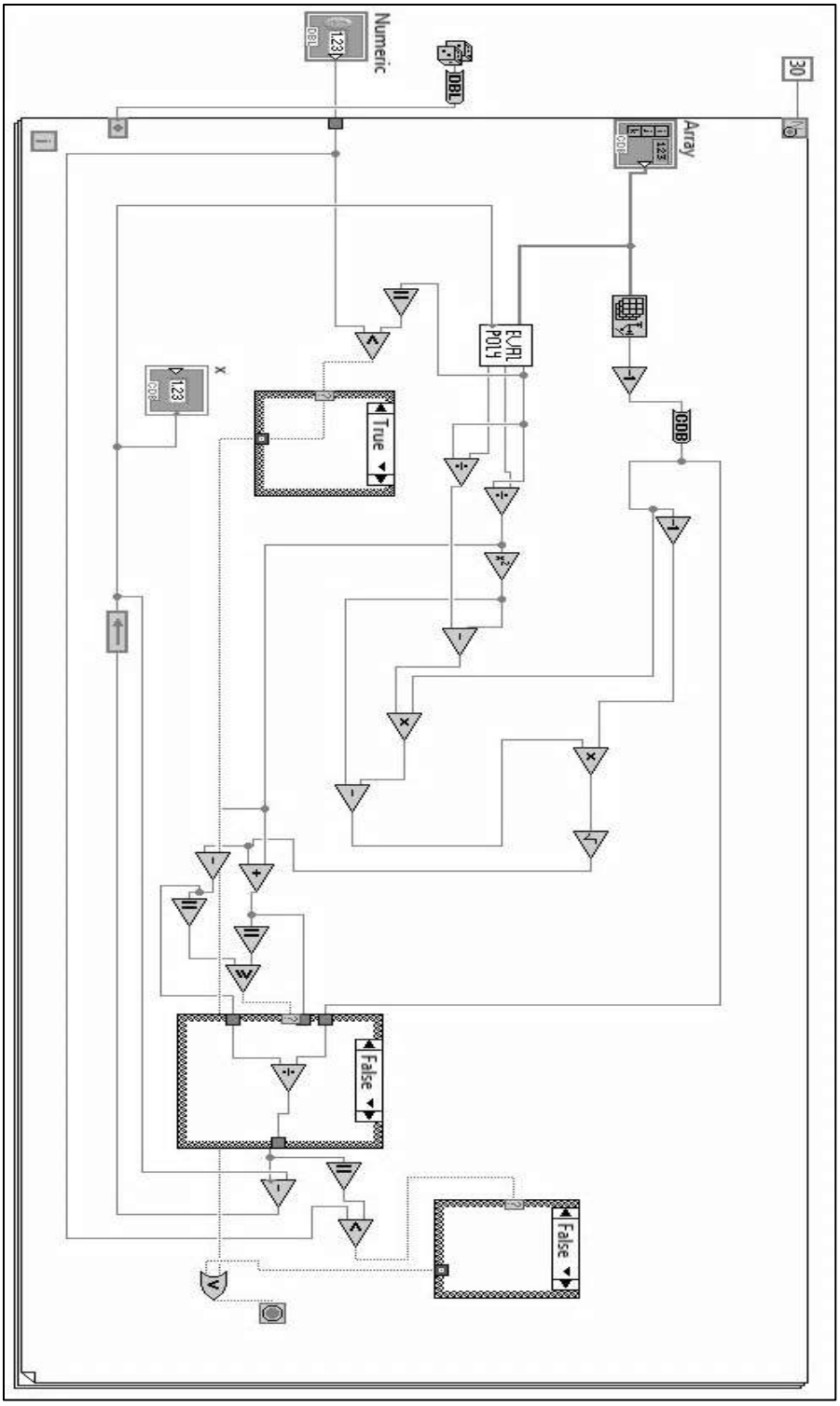
a. Diagramme d'évaluation (`evalpoly.vi`)



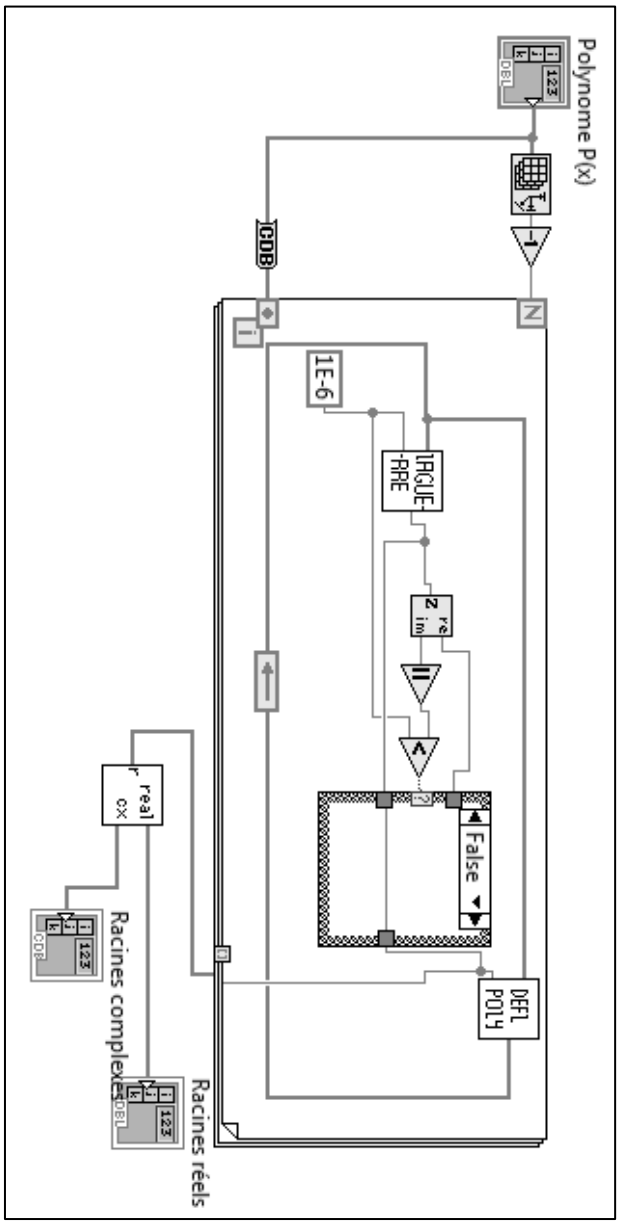
b. Diagramme de déflation (`deflpoly.vi`)



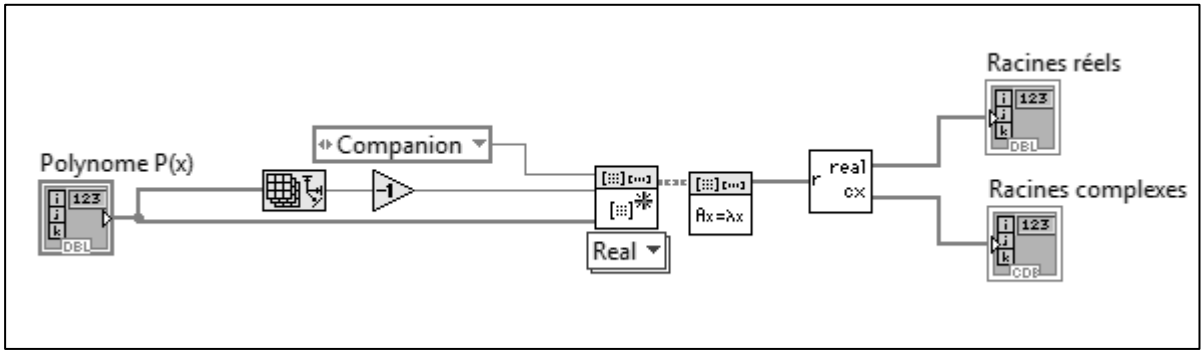
c. Diagramme de Laguerre (`laguerre.vi`)



d. Diagramme de Polyroots (`Polyroots.vi`)



- **Bloc-diagramme méthode Matricielle (`M_Matricielle.vi`)**



- **Code source C (Pour DSP)**

f) Méthode de Laguerre

a. Fonction d'évaluation

```
void evalpoly (){
     int i,de;
     de=2;
     dp = Complex(0.0,0.0);
     ddp = Complex(0.0,0.0);
     p=a[0];
     for (i = 0; i <nnn; i++){
       ddp= Cadd(Cmul(ddp , x) ,RCmul(de , dp));
          dp= Cadd(Cmul(dp , x) ,p);
       p = Cadd(Cmul(p , x) ,a[i+1]);
       }

 }
```

b. Fonction de Déflation

```
void deflpoly (){
  int i;
  b[0] =a[0];
  for (i = 1; i <nnn; i++){
        b[i] = Cadd(a[i],Cmul(x,b[i-1]));
        a[i]=b[i];}
  }
```

c. Fonction de Laguerre

```
void laguerre (){
  int i;
  float nn;
  fcomplex  g, f,h , dx,c1 ,c2 ,x1;
   x = Complex(0.0,0.0);
   nn=(float) nnn;
   x1 = Complex(nn,0.0);
   for (i = 1; i <= 30; i++){
     evalpoly ();
     if (Cabs(p) < tol) return;
     g = Cdiv( dp, p);
     h= Csub(Cmul(g , g) , Cdiv( ddp, p));
     f = Csqrt(RCmul( (nnn-1) , Csub(RCmul( nnn , h) , Cmul(g , g))));
     c1 = Cadd(g,f);
     c2 = Csub(g,f);
     if (Cabs(c1) >= Cabs(c2))
      dx = Cdiv(x1,c1);
     else
      dx =Cdiv(x1,c2);
     x = Csub(x,dx);
     if (Cabs(dx) < tol) return;
        }
  }
```

d. Fonction Polyroots :

```
void polyroots ()
{ int j;
  nnn=n;
  for (j = 0; j < n; j++){
    laguerre();
    root[j]=x;
    deflpoly ();
     nnn=nnn-1;
    }
  }
```

g) La méthode Matricielle

a. Polyroots

```
void polyroots()
{
int j,k;
float xr,xi;
    if (n > MAXM || a[n] == 0) printf("faux argument polyroots");
    for (k=1;k<=n;k++) {
       a[1][k] = -aa[n-k]/aa[n];
       for (j=2;j<n;j++) a[j][k]=0.0;
       if (k < n) a[k+1][k]=1.0;
    }
    eig();
    for (j=2;j<=n;j++) {
    xr=wr[j];
        xi=wi[j];
        for (k=j-1;k>=1;k--) {
            if (wr[k] <= xr) break;
            wr[k+1]=wr[k];
            wi[k+1]=wi[k];
        }
        wr[k+1]=xr;
        wi[k+1]=xi;
    }
}
```

b. Les valeurs propres

```
void eig()
{
int nn,m,l,k,j,its,i,mmin;
float z,y,x,w,v,u,t,s,r,q,p,anorm;
   anorm=0.0;
   for(i=1;i<=n;i++)
      for (j=IMAX(i-1,1);j<=n;j++)
          anorm += fabs(a[i][j]);
   nn=n;
   t=0.0;
   while (nn >= 1) {
```

```
            its=0;
        do {
           for (l=nn;l>=2;l--) {
               fabs(a[l-1][l-1])+fabs(a[l][l]);
               if (s == 0.0) s=anorm;
               if ((float)(fabs(a[l][l-1]) + s) == s) {
                  a[l][l-1]=0.0;
                 break;
               }
           }
           x=a[nn][nn];
           if (l == nn) {
              wr[nn]=x+t;
              wi[nn--]=0.0;
           } else {
              y=a[nn-1][nn-1];
              w=a[nn][nn-1]*a[nn-1][nn];
              if (l == (nn-1)) {
                p=0.5*(y-x);
                q=p*p+w;
                z=sqrt(fabs(q));
                x += t;
                if (q >= 0.0) {
                   z=p+SIGN(z,p);
                   wr[nn-1]=wr[nn]=x+z;
                   if (z) wr[nn]=x-w/z;
                   wi[nn-1]=wi[nn]=0.0;
                } else {
                   wr[nn-1]=wr[nn]=x+p;
                   wi[nn-1]= -(wi[nn]=z);
                }
                nn -= 2;
               } else {
                   if (its == 90) printf("Too many iterations in eig");
                   if (its == 30 || its == 60) {
                      t += x;
                      for (i=1;i<=nn;i++) a[i][i] -= x;
                      s=fabs(a[nn][nn-1])+fabs(a[nn-1][nn-2]);
                      y=x=0.75*s;
                      w = -0.4375*s*s;
                    }
                    ++its;
                    for (m=(nn-2);m>=l;m--) {
                        z=a[m][m];
                        r=x-z;
                        s=y-z;
                        p=(r*s-w)/a[m+1][m]+a[m][m+1];
                        q=a[m+1][m+1]-z-r-s;
                        r=a[m+2][m+1];
                        s=fabs(p)+fabs(q)+fabs(r);
                        p /= s;
                        q /= s;
                        r /= s;
                        if (m == l) break;
                        u=fabs(a[m][m-1])*(fabs(q)+fabs(r));
                        v=fabs(p)*(fabs(a[m-1][m-
1])+fabs(z)+fabs(a[m+1][m+1]));
                        if ((float)(u+v) == v) break;
```

```
                    }
                    for (i=m+2;i<=nn;i++) {
                        a[i][i-2]=0.0;
                        if (i != (m+2)) a[i][i-3]=0.0;
                    }
                    for (k=m;k<=nn-1;k++) {
                        if (k != m) {
                           p=a[k][k-1];
                           q=a[k+1][k-1];
                           r=0.0;
                           if (k != (nn-1)) r=a[k+2][k-1];
                           if ((x=fabs(p)+fabs(q)+fabs(r)) != 0.0) {
                              p /= x;
                              q /= x;
                              r /= x;
                           }
                        }
                        if ((s=SIGN(sqrt(p*p+q*q+r*r),p)) != 0.0) {
                           if (k == m) {
                              if (l != m)
                                 a[k][k-1] = -a[k][k-1];
                              } else
                                 a[k][k-1] = -s*x;
                              p += s;
                              x=p/s;
                              y=q/s;
                              z=r/s;
                              q /= p;
                              r /= p;
                              for (j=k;j<=nn;j++) {
                                  p=a[k][j]+q*a[k+1][j];
                                  if (k != (nn-1)) {
                                     p += r*a[k+2][j];
                                     a[k+2][j] -= p*z;
                                  }
                                  a[k+1][j] -= p*y;
                                  a[k][j] -= p*x;
                               }
                               mmin = nn<k+3 ? nn : k+3;
                               for (i=l;i<=mmin;i++) {
                                   p=x*a[i][k]+y*a[i][k+1];
                                   if (k != (nn-1)) {
                                      p += z*a[i][k+2];
                                      a[i][k+2] -= p*r;
                                   }
                                   a[i][k+1] -= p*q;
                                   a[i][k] -= p;
                               }
                           }
                       }
                   }
               }
           } while (l < nn-1);
       }
   }
```

- **La bibliothèque**

a. Les nombres complexes

1. Complex.h :

```
#include <math.h>
typedef struct FCOMPLEX {float r,i;} fcomplex;
   fcomplex Cadd(fcomplex a, fcomplex b);

   fcomplex Csub(fcomplex a, fcomplex b);

   fcomplex Cmul(fcomplex a, fcomplex b);

   fcomplex Complex(float re, float im);

   fcomplex Cdiv(fcomplex a, fcomplex b);

   float Cabs(fcomplex z);

   fcomplex Csqrt(fcomplex z);

   fcomplex RCmul(int x, fcomplex a);
```

2. Complex.c :

```
#include <math.h>
#include<stdio.h>
typedef struct FCOMPLEX {float r,i;} fcomplex;
   fcomplex Cadd(fcomplex a, fcomplex b)
      {
       fcomplex c;
        c.r=a.r+b.r;
        c.i=a.i+b.i;
        return c;
      }
   fcomplex Csub(fcomplex a, fcomplex b)
      {
      fcomplex c;
      c.r=a.r-b.r;
      c.i=a.i-b.i;
      return c;
      }
    fcomplex Cmul(fcomplex a, fcomplex b)
      {
      fcomplex c;
      c.r=a.r*b.r-a.i*b.i;
      c.i=a.i*b.r+a.r*b.i;
      return c;
      }
    fcomplex Complex(float re, float im)
      {
        fcomplex c;
        c.r=re;
        c.i=im;
        return c;
```

```
   }
 fcomplex Cdiv(fcomplex a, fcomplex b)
    {
       fcomplex c;
       float r,den;
       if (fabs(b.r) >= fabs(b.i)) {
       r=b.i/b.r;
       den=b.r+r*b.i;
       c.r=(a.r+r*a.i)/den;
       c.i=(a.i-r*a.r)/den;
       } else {
         r=b.r/b.i;
         den=b.i+r*b.r;
         c.r=(a.r*r+a.i)/den;
         c.i=(a.i*r-a.r)/den;
      }
       return c;
     }
float Cabs(fcomplex z)
   {
    float x,y,ans,temp;
    x=fabs(z.r);
    y=fabs(z.i);
   if (x == 0.0)
      ans=y;
   else if (y == 0.0)
      ans=x;
     else if (x > y) {
      temp=y/x;
      ans=x*sqrt(1.0+temp*temp);
     } else {
       temp=x/y;
     ans=y*sqrt(1.0+temp*temp);
       }
    return ans;
    }
  fcomplex Csqrt(fcomplex z)
     {
       fcomplex c;
       float x,y,w,r;
       if ((z.r == 0.0) && (z.i == 0.0)) {
        c.r=0.0;
        c.i=0.0;
       return c;
       } else {
    x=fabs(z.r);
    y=fabs(z.i);
   if (x >= y) {
     r=y/x;
     w=sqrt(x)*sqrt(0.5*(1.0+sqrt(1.0+r*r)));
     } else {
       r=x/y;
       w=sqrt(y)*sqrt(0.5*(r+sqrt(1.0+r*r)));
    }
   if (z.r >= 0.0) {
       c.r=w;
       c.i=z.i/(2.0*w);
   } else {
```

```
        c.i=(z.i >= 0) ? w : -w;
        c.r=z.i/(2.0*c.i);
       }
 return c;
  }
}
  fcomplex RCmul(int x, fcomplex a)
    {
     fcomplex c;
     c.r=x*a.r;
     c.i=x*a.i;
     return c;
    }
```

BIBLIOGRAPHIE

[1] Franck Jedrzejewski," Introduction aux méthodes numériques Deuxième édition", Springer-Verlag France, Paris 2005.

[2] Wankere R. Mekwi, "Iterative Methods for Roots of Polynomials". University of Oxford, Trinity 2001.

[3] Massoud Malek, "Root-Finding Methods", California State University, East Bay.

[4] http://www.sciences.univ-nantes.fr/sites/fethi_aloui/matlab/11intro/11intro.htm

[5] National instrument, Manuel, "Getting Started with LabVIEW", November 2001.

[6] Khaled Zaabi, "Implémentation d'une méthode de reconnaissance de la parole sur le processeur de traitement numérique du signal tms320c6711", montréal, 10 juin 2004.

[7] d. Lacroix, "Tome 1 Généralités sur les DSP", Stage DSP – IUFM.

[8] Sen M Kuo, Bob H lee, "Real-Time Digital Signal Processing", USA 2001.

[9] Pascal Viot, "Méthodes d'analyse numérique Cours du DEA « Modélisation Dynamique et Statistique des Systèmes Complexes »", Laboratoire de Physique Théorique des Liquides, Boite 121, 4, Place Jussieu, 75252 Paris Cedex 05, 17 janvier 2003.