Calcul Embarqué

Alain Le Gall - alain.legall@ece.fr

2022-2023

Table des matières

I-Calcul numérique

Introduction

0.1 Quelques mots avant de commencer

L'Homme a toujours eu besoin d'algorithmes pour l'aider à résoudre des problèmes concrets. Il n'est pas étonnant qu'un Newton ou qu'un Halley, tous deux astronomes et mathématiciens, aient inventé des algorithmes, l'astronomie étant un domaine où le calcul est roi. Poursuivant sur leur voie, et celle de nombreux autres, les scientifiques ont inventé beaucoup d'algorithmes, qui permettent à l'homme du XXIème siècle de résoudre des équations, des problèmes de physique, mathématique, biologie, économie etc... Nous allons ici aborder des algorithmes utilisés dans des domaines très variés, mais en centrant notre étude pratique sur les Systèmes Embarqués, domaine de l'électronique. Nos algorithmes seront implémentés sur l'Arduino DUE, que l'on codera avec le langage C++.

Les cours CI seront le lieu de découverte des algorithmes essentiels, les TD donneront l'occasion d'effectuer des études théoriques et informatiques, les TP quant à eux permettront d'appliquer les connaissances précédemment acquises à la résolution de questions d'ordre expérimental, à l'aide de l'Arduino DUE et du kit d'électronique.

IMPORTANT : En TD il est indispensable de disposer d'une calculatrice. Il est aussi indispensable en TP de disposer d'un ordinateur et de l'Arduino DUE pour exécuter les programmes que vous aurez à coder (on emploiera alors ARDUINO IDE 2 comme logiciel) ; et de disposer en plus de votre kit d'électronique avec ses composants.

Chapitre 1

Recherche de racines d'équations non linéaires

1.1 Contexte de notre étude

Il est très fréquent d'avoir à rechercher une ou des racines d'une fonction réelle f, c'est-à-dire d'avoir à résoudre l'équation $f(x) = 0$. Une quantité astronomique de problèmes d'étude technique, industrielle, R & D etc... se ramènent à la recherche de racines de fonctions. La plupart du temps, cette équation n'a pas de solution qui s'exprime simplement. Essayez par exemple de m'exprimer une solution non nulle de l'équation :

$$tan(x) = x$$

c'est-à-dire de l'équation

$$f(x) = tan(x) - x = 0$$

(on dit que l'on recherche une racine de f). Vous pouvez chercher longtemps, cette équation n'a pas de solution non nulle qui s'exprime analytiquement, seule une approximation peut être trouvée, grâce à un calcul numérique... Nous allons vous présenter ici différents de ces fameux algorithmes de recherche de racines. Comme on veut les comparer aussi, selon leur précision, leur vitesse, leurs avantages et inconvénients..., nous serons amenés à introduire des notions nouvelles mais pertinentes telle que par exemple l'efficacité d'un algorithme de recherche de racine.

Nous prendrons, pour appuyer notre étude théorique, l'exemple de la recherche du nombre d'or Φ à partir de nos différents algorithmes de recherche de racines, afin de pouvoir les comparer facilement ! Ce nombre est connu depuis l'Antiquité , il a de très nombreuses propriétés, et peut se retrouver dans la nature, l'architecture, la musique etc... Il est, entre autres, la solution positive de l'équation polynômiale du second degré très simple suivante :

$$x^2 - x - 1 = 0$$

Sa valeur, notée Φ, se trouve donc facilement :

$$\Phi = \frac{1 + \sqrt{5}}{2} \simeq 1,61803398874989...$$

Essayons de retrouver une approximation de cette valeur, à partir de l'équation dont il est solution. On note

$$f(x) = x^2 - x - 1$$

Il nous reste à déterminer par des méthodes numériques, issues des algorithmes que l'on va étudier dans ce cours, la racine positive de f. Nous allons donc prendre connaissance de différents algorithmes de recherche de racine, et on les comparera sur cet exemple précis, pour en évaluer l'efficacité, d'autant plus facilement avec notre présent exemple où la racine est connue de façon exacte (ce qui permettra facilement d'évaluer aussi la précision par exemple).

1.2 Méthode de dichotomie

1.2.1 Justification théorique de la méthode

On cherche à calculer α qui est une racine de la fonction réelle de la variable réelle $f(x)$. On suppose donc :

$$f(\alpha) = 0$$

Supposons que l'on connaisse un intervalle $I =]a, b[$ dans lequel la racine soit incluse :

$$\alpha \in]a, b[$$

La méthode de dichotomie consiste à diviser l'intervalle de recherche par deux à chaque itération en gardant la racine dans cet intervalle. La racine α est trouvée lorsque la largeur de l'intervalle (qui diminue forcément au cours des itérations) est inférieure à une certaine tolérance, notée ϵ.

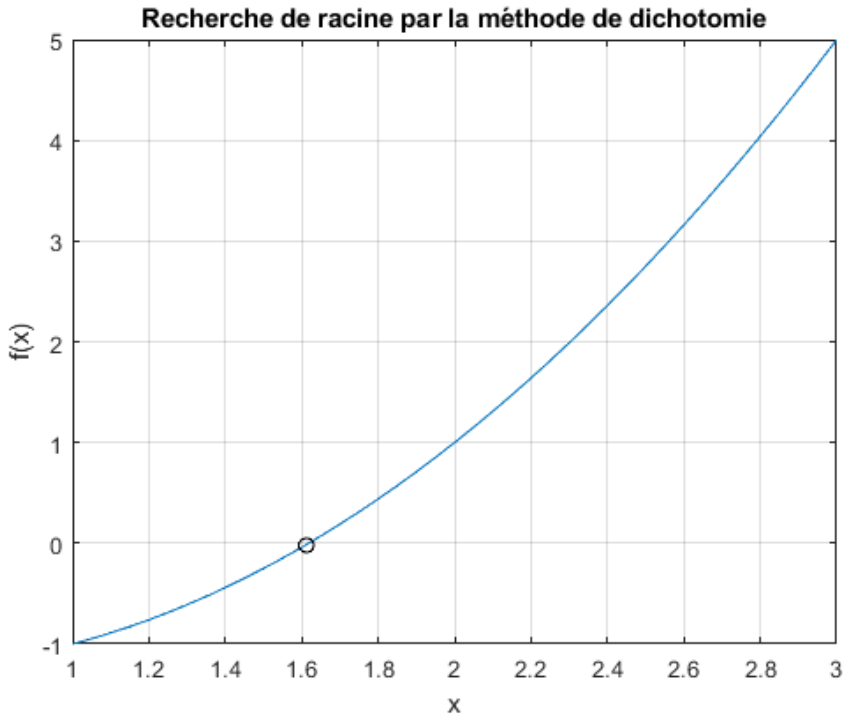
1.2.2 Algorithme

Algorithm 1: Algorithme de dichotomie

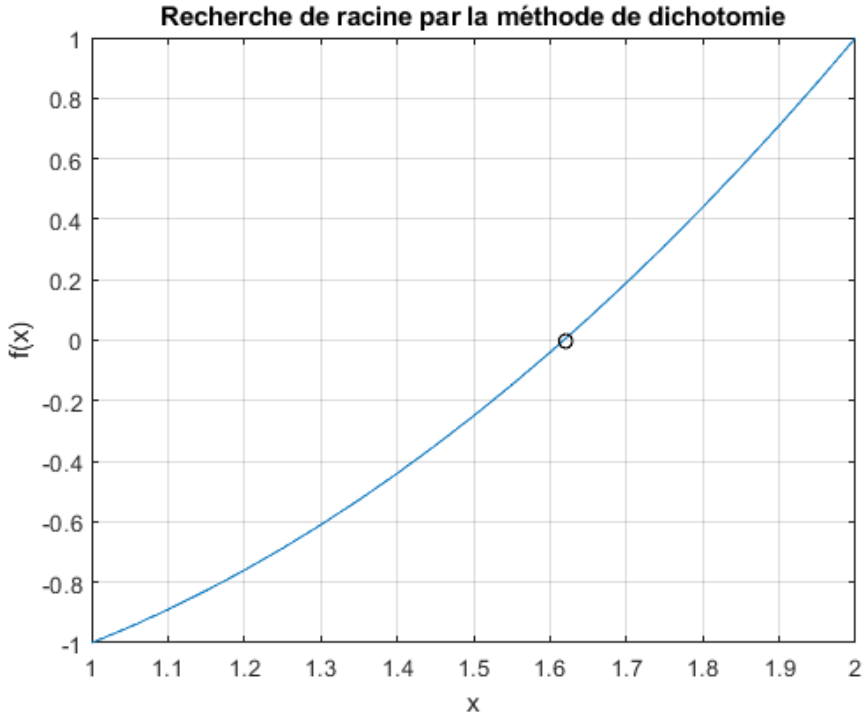
Data: a et b (intervalle de recherche initial), ϵ (précision)
$fa = f(a)$
$fb = f(b)$
if $fa.fb < 0$ **then**
  while $|b - a| > \epsilon$ **do**
    $x = \frac{a+b}{2}$ et $y = f(x)$
    if $fa.y > 0$ **then**
      $a = x$
      $fa = y$
    else
      $b = x$
      $fb = y$
    end
  end
end
Result: x : approximation de la racine cherchée α

1.2.3 Exemple

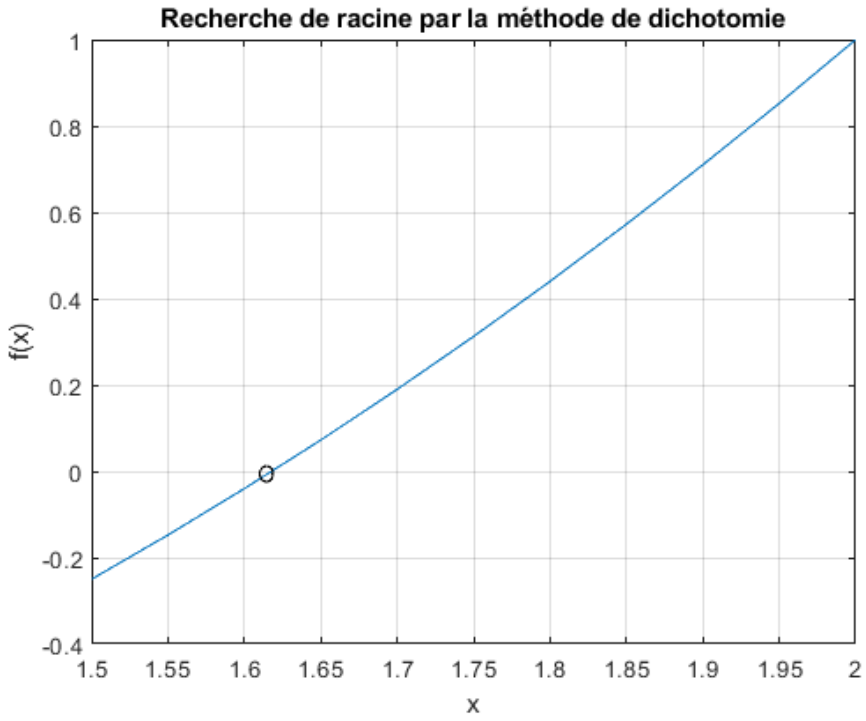
On étudie notre exemple : $f(x) = x^2 - x - 1$ On part de l'intervalle $[1; 3]$ pour lequel on sait sensiblement que la raciner s'y trouve (on peut aussi choisir de beaucoup plus grandes valeurs moins "confortables").



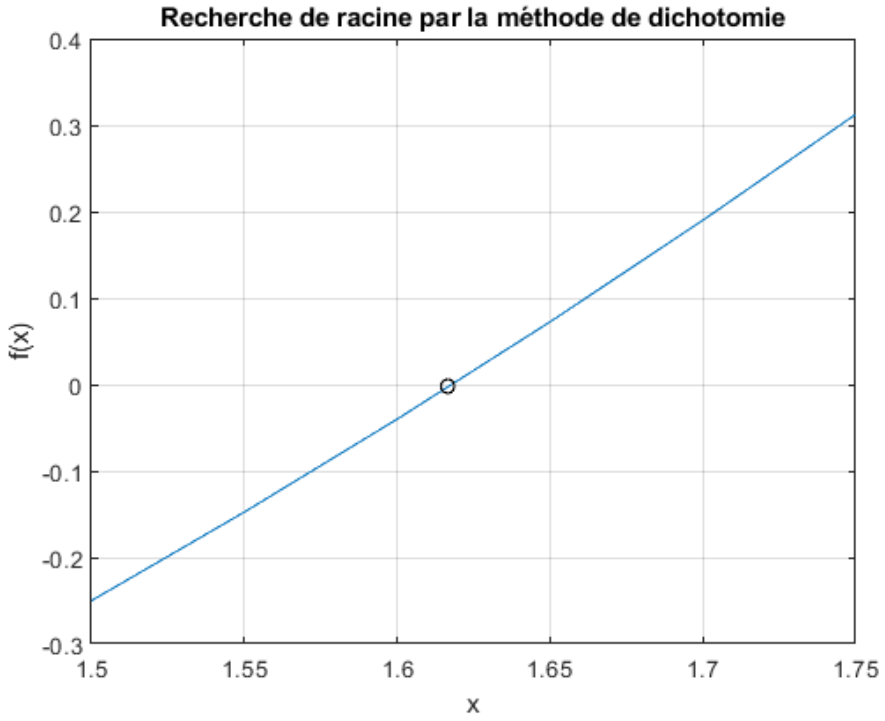
Comme $f(x)$ change de signe entre $a = 1$ et $\frac{a+b}{2} = 2$, on sélectionne comme nouvel intervalle : $[1; 2]$



Comme $f(x)$ change de signe entre $\frac{a+b}{2} = 1,5$ et $b = 2$, on sélectionne comme nouvel intervalle : $[1,5; 2]$



Comme $f(x)$ change de signe entre $a = 1,5$ et $\frac{a+b}{2} = 1,75$, on sélectionne comme nouvel intervalle : $[1,5; 1,75]$



etc...etc... A chaque étape, la largeur de l'intervalle est divisée par 2 seulement ; ce processus converge, certes, mais pas rapidement !

1.2.4 Programme en langage C++

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

double f(double x);
double dichotomie();

const double phi(0.5+0.5*sqrt(5.0));
char attends;

int main()
{
    double x(0.0);
    cout << "Dichotomie" << endl;
    x=dichotomie();

    cout<< "racine x = "<<setprecision(15)<<x<<endl;
    cout<< "f(x)="<<setprecision(15)<<f(x)<<endl;
    cout<<"erreur commise : "<<setprecision(15)<<fabs(phi-x)<<endl;

    return 0;
}

double f(double x){
    return x*x-x-1.0;
}

double dichotomie()
{
    double eps(1.0e-10),a(0.0),fa(0.0),b(0.0),fb(0.0),x(0.0),y(0.0);
    int n_max(200),n(0);
    do{
        cout<<"choix-intervalle"<<endl;
        cout<<"a=";
        cin>>a;
        fa=f(a);
        cout<<"b=";
        cin>>b;
        fb=f(b);
    }while((b<a)||(f(a)*f(b)>=0.0));

    do{
        n++;
        x=(a+b)/2.0;
        y=f(x);
        if(fa*y>0.0)
```

```
        {
            a=x;
            fa=y;
        }
        else{
            b=x;
            fb=y;
        }
    }while((fabs(b-a)>eps)&&(n<n_max));
    cout<<n<<" iterations"<<endl;
    return x;
}
```

1.2.5 Résultats numériques

Sur cet exemple où $\epsilon = 10^{-10}$: 10 décimales justes !

```
Dichotomie
choix-intervalle
a=1.0
b=3.0
35 iterations
racine x = 1.61803398869233
f(x)=-1.28714372493732e-10
erreur commise : 5.75628433807651e-11
```

Sur cet exemple où $\epsilon = 10^{-15}$: 15 décimales justes !

```
Dichotomie
choix-intervalle
a=1.0
b=3.0
51 iterations
racine x = 1.61803398874989
f(x)=-1.77635683940025e-15
erreur commise : 8.88178419700125e-16
```

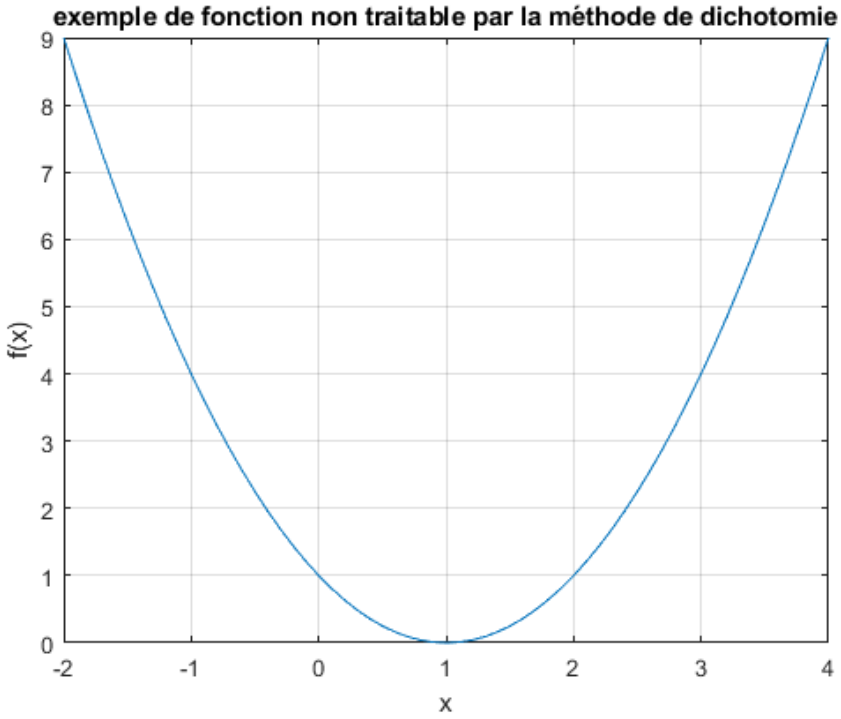
On observe que le nombre d'itérations est assez élevé, mais permet d'atteindre la précision voulue. Sur cette fonction f simple, ce nombre élevé d'itérations ne ralentit pas trop le calculateur, mais si la fonction était bien plus compliqué, cela serait le cas. On n'emploie donc pas habituellement une méthode de dichotomie jusqu'à la précision finale : on débute par cette méthode puis on passe le relai à une méthode plus rapide telle que la méthode de Newton-Raphson.

1.2.6 Remarques

— Le test $|b - a| < \epsilon$ permet de cerner rapidement la racine mais on peut aussi employer le test $|f(x)| < \epsilon$, ce qui permet évidemment aussi d'obtenir une bonne précision sur cette même racine α. Étant donné que la méthode de dichotomie est plus utilisée pour trouver une approximation de la racine, on utilisera de préférence le test $|b - a| < \epsilon$ puis on appliquera, comme dit précédemment, ensuite un autre algorithme de recherche de racine pour poursuivre la recherche, comme

par exemple l'algorithme de Newton-Raphson (voir plus bas dans ce cours), afin d'obtenir ensuite une précision optimale sur la racine α.

— La méthode de dichotomie ne fonctionne bien entendu pas dans des cas de figure comme celui présenté ci-dessous car la fonction f ne change pas de signe (condition nécessaire pour faire fonctionner l'algorithme de dichotomie).



1.2.7 Conclusion

L'algorithme de dichotomie est très efficace pour "cerner" une racine, mais il est plutôt lent ; on l'emploie en général pour débuter la recherche d'une racine, ensuite une méthode plus rapide est employée (cf. méthodes décrites plus loin, par exemple la méthode de Newton-Raphson).

1.3 Méthode de Newton-Raphson

1.3.1 Justification théorique de la méthode

On cherche à calculer α une racine de la fonction réelle de la variable réelle $f(x)$. On suppose donc :

$$f(\alpha) = 0$$

Posons $x = \alpha + h$, avec $h << 1$ (on se place près de la racine) et écrivons un développement de Taylor de la fonction f :

$$f(x) = f(\alpha + h) = f(\alpha) + hf'(\alpha) + ... + \frac{h^p}{p!} f^{(p)}(\alpha) + ...$$

Si α est une racine d'ordre p de f, alors par définition on a :

$$\begin{cases} f(\alpha) = f'(\alpha) = ... = f^{(p-1)}(\alpha) = 0 \\ f^{(p)}(\alpha) \neq 0 \end{cases}$$

Ainsi, $f(x) = \frac{h^p}{p!} f^{(p)}(\alpha)$+termes négligeables
Soit encore, avec une bonne approximation : $f(x) = C^{te}.h^p = C^{te}.(x-\alpha)^p$ donc $f'(x) = p.C^{te}.(x-\alpha)^{p-1}$ et ainsi :

$$s_1 = \frac{f'(x)}{f(x)} = \frac{p}{x-\alpha}$$

On en déduit : $x - \alpha = \frac{p}{s_1}$ soit $\alpha = x - \frac{p}{s_1}$.
Donc en résumé :

$$\alpha = x - p\frac{f(x)}{f'(x)}$$

Interprétation : Si $x = \alpha + h$ ($h << 1$) est proche de la racine α cherchée, alors une bonne approximation de α est donnée par :

$$\boxed{\alpha \simeq x - p\frac{f(x)}{f'(x)}}$$

Généralisation : On peut réitérer le processus et alors la suite définie par récurrence selon :

$$\boxed{x_{n+1} = x_n - p\frac{f(x_n)}{f'(x_n)}} \tag{1.1}$$

peut converger, en général cela dépend de la valeur initiale x_0 choisie qui doit être assez "proche" de α , vers la racine recherchée α :

$$\lim_{n \to +\infty} x_n = \alpha$$

1.3.2 Relation de récurrence explicite

Il est aussi possible de trouver une relation de récurrence directe afin de définir la suite x_k qui converge vers une racine α de la fonction f. Prenons notre exemple du nombre d'or. Ici, $f(x) = x^2 - x - 1$ et donc l'algorithme de Newton-Raphson s'écrit (on a aussi $p = 1$ ici) :

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

avec aussi $f'(x) = 2x - 1$, donc :

$$x_{k+1} = x_k - \frac{x_k^2 - x_k - 1}{2x_k - 1}$$

soit, après calculs :

$$x_{k+1} = \frac{x_k^2 + 1}{2x_k - 1}$$

Il est alors facile de calculer les différents termes de cette suite, qui converge vers le nombre d'or Φ. Même avec une simple calculette, le calcul est facile à mener :

1. Taper une valeur positive au hasard sur la calculette et valider
2. En utilisant la touche "ANS" ou "REP" selon le modèle de la calculette, saisir l'équation suivante :

$$(ANS^2 + 1)/(2 * ANS - 1)$$

3. Valider un certains nombre de fois jusqu'à ce que la valeur se stabilise à l'écran
4. Lorsque le processus a convergé, la valeur qui s'affiche est une approximation, dans les limites de la calculette, de la valeur de la racine Φ recherchée

On peut aussi écrire un programme informatique qui utilise la définition de cette suite x_k définie ci-dessus par récurrence, et calcule la limite recherchée :

Algorithme Écrivons l'algorithme permettant d'obtenir la racine à l'aide d'une suite définie par récurrence :

Algorithm 2: Algorithme de calcul de Φ à l'aide d'une suite définie par récurrence

Data: r(valeur initiale), ϵ (précision)
while $|r - s| > \epsilon$ **do**
  $s = r$
  $r = \frac{r^2+1}{2*r-1}$
end
Result: r approximation de la racine cherchée α

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

double suite_recurrence();

const double phi(0.5+0.5*sqrt(5.0));
char attends;

int main()
{
    double x(0.0);
    cout << "Recherche du nombre d or avec une suite" << endl;
    x=suite_recurrence();

    cout<< "racine x = "<<setprecision(15)<<x<<endl;
    cout<<"erreur commise : "<<setprecision(15)<<fabs(phi-x)<<endl;

    return 0;
}

double suite_recurrence()
{
    double eps(1.0e-15),r(1.0),s(0.0);
    int n_max(200),n(0);
```

```
    do{
    cout<<"valeur initiale :";
    cin>>r;
    }while(r<=0.0);

    do{
        n++;
        s=r;
        r=(r*r+1.0)/(2.0*r-1.0);
    }while((fabs(r-s)>eps)&&(n<n_max));
    cout<<n<<" iterations"<<endl;
    return r;
}
```

Résultats numériques

```
Recherche du nombre d or a l aide d une suite definie par recurrence
valeur initiale :1000.0
15 iterations
racine x = 1.61803398874989
erreur commise : 0
```

On n'observe aucune erreur... (précision choisie de $\epsilon = 10^{-15}$, dans les limites du calculateur bien entendu !)

```
Recherche du nombre d or a l aide d une suite definie par recurrence
valeur initiale :10.0
9 iterations
racine x = 1.61803398874989
erreur commise : 0
```

En partant d'une valeur plus proche de la racine, le nombre d'itérations est bien entendu plus faible ; et de même on n'observe aucune erreur... (précision choisie de $\epsilon = 10^{-15}$, dans les limites du calculateur bien entendu !)

```
Recherche du nombre d or a l aide d une suite definie par recurrence
valeur initiale :10.0
9 iterations
racine x = 1.6180339887498949
erreur commise : 0
```

Sur cet exemple affiché ci-dessus, on a choisi une meilleure précision encore : $\epsilon = 10^{-18}$!

Conclusion : La méthode de Newton-Raphson permet, soit de construire explicitement une suite définie par récurrence qui converge vers la racine voulue ; soit de coder en langage informatique un programme avec les définitions des fonctions f et f' et de faire fonctionner l'algorithme. Il est évident que si les fonctions f et f' sont simples d'expression, la première méthode est à privilégier. Cependant, la méthode informatique (cf ci-dessous) fonctionne à priori toujours.

1.3.3 Algorithme

Algorithm 3: Algorithme de Newton-Raphson

Data: x_0(valeur initiale), p (ordre de la racine), ϵ (précision)
while $|x_{k+1} - x_k| > \epsilon$ **do**
  $x_{k+1} = x_k - p\frac{f(x_k)}{f'(x_k)}$
end
Result: x_n approximation de la racine cherchée α

1.3.4 Programme en langage C++

cf TP

1.3.5 Exemple

cf TP

1.3.6 Amélioration : estimation en cours de calcul de l'ordre de la racine

cf TP

1.4 Méthode de Laguerre

1.4.1 Justification théorique de la méthode

Cet algorithme est fourni ici directement, la démonstration étant nettement plus compliquée ; il suffira de se reporter à un ouvrage spécialisé pour l'obtenir. L'intérêt de présenter cette méthode est que c'est un algorithme spécialement conçu pour rechercher les racines des polynômes, et il est vraiment très efficace.
La méthode est encore basée sur une formule de récurrence. Notons n le degré du polynôme étudié, et p l'ordre de la racine recherchée. On admet alors que la relation de récurrence suivante converge vers la racine :

$$x_{k+1} = x_k - \frac{nf(x_k)}{f'(x_k) + signe(f'(x_k))\sqrt{(\frac{n-p}{p})((n-1)f'^2(x_k) - nf(x_k)f''(x_k))}}$$

1.4.2 Algorithme

Algorithm 4: Algorithme de Laguerre

Data: x_0(valeur initiale), p (ordre de la racine), n (degré du polynôme), ϵ (précision)
while $|x_{k+1} - x_k| > \epsilon$ **do**
  $x_{k+1} = x_k - \frac{nf(x_k)}{f'(x_k)+signe(f'(x_k))\sqrt{(\frac{n-p}{p})((n-1)f'^2(x_k)-nf(x_k)f''(x_k))}}$
end
Result: x_n approximation de la racine cherchée α

1.4.3 Programme en langage C++

cf TP

1.4.4 Exemple

cf TP

1.4.5 Remarques

— Cet algorithme fonctionne à merveille avec des polynômes, et il n'est même pas obligatoire de connaitre l'ordre p de la racine (on peut poser $p = 1$ si p est inconnu).
— On peut bien entendu l'utiliser si f est une fonction différente d'un polynôme.
— Il faut connaitre f, f' et f'' comme dans l'algorithme de Halley (cf TD), ce qui peut être contraignant si f n'est pas un polynôme.

1.4.6 Conclusion

L'algorithme de Laguerre est extrêmement rapide, surtout en ce qui concerne les polynômes, et très précis. Il permet d'obtenir en très peu d'étapes la valeur précise d'une racine. Le seul inconvénient est qu'il faut connaitre les dérivées première et seconde de f (mais cela ne pose pas de soucis avec les polynômes en général !).

1.5 Conclusion

Nous voilà arrivés au terme de notre voyage informatique au pays des algorithmes de recherche de racines ! La plupart des racines des équations à coefficients réels de la variable réelle peuvent être obtenues par l'une ou l'autre des méthodes vues dans ce livre. En général, on conseille la méthode de Halley (cf TD), très rapide à condition de connaitre jusqu'à la dérivée seconde de la fonction étudiée bien entendu ; dans le cas contraire se rabattre sur l'excellente méthode de Newton-Raphson, ou encore une méthode plus lente mais moins exigeante. Bien entendu, si vous travaillez sur des polynômes, la méthode conseillée est celle de Laguerre, alors extrêmement rapide. On peut aussi mélanger les méthodes, à savoir commencer par un algorithme peu précis pour approcher la racine, puis finir la recherche de la racine par un algorithme plus précis. Enfin, bien entendu, il existe d'autres méthodes de recherche de racines qui n'ont pas été abordées ici, que vous pourrez étudier ultérieurement.

1.6 ANNEXE : ordre de convergence et efficacité d'un processus itératif

1.6.1 Méthode générale

Soit un processus itératif qui converge vers une racine α.

$$x_{k+1} = \varphi(x_k)$$

On note : $\epsilon_k = \alpha - x_k$: l'erreur de x_k par rapport à la racine α à la k^{ime} itération.

$$\epsilon_{k+1} = \alpha - x_{k+1} = \alpha - \varphi(x_k) = \alpha - \varphi(\alpha - \epsilon_k)$$

$$\alpha - \epsilon_{k+1} = \varphi(\alpha - \epsilon_k)$$

A la limite, on aura (théorème du point fixe), comme $\lim_{k\to+\infty} \epsilon_k = 0$, :

$$\alpha = \varphi(\alpha)$$

Et, comme

$$\epsilon_{k+1} = \alpha - \varphi(\alpha - \epsilon_k)$$

il vient en faisant un développement limité de φ autour de α :

$$\epsilon_{k+1} = \alpha - \varphi(\alpha) + \epsilon_k\varphi'(\alpha) - \epsilon_k^2\frac{\varphi''(\alpha)}{2!} + \epsilon_k^3\frac{\varphi'''(\alpha)}{3!} + ...$$

Or, $\alpha - \varphi(\alpha) = 0$, donc :

$$\boxed{\epsilon_{k+1} = \varphi'(\alpha)\epsilon_k - \frac{\varphi''(\alpha)}{2!}\epsilon_k^2 + \frac{\varphi'''(\alpha)}{3!}\epsilon_k^3 + ...} \tag{1.2}$$

On dira qu'un processus itératif $x_{k+1} = \varphi(x_k)$ est d'ordre m si l'on a pour tout entier naturel k :

$$\epsilon_{k+1} = C\epsilon_k^m$$

(avec C : constante).

Si $\varphi'(\alpha) \neq 0$ **:**

$$\epsilon_{k+1} \simeq \varphi'(\alpha)\epsilon_k$$

Le processus itératif est d'ordre 1, on a donc une convergence linéaire.

Si $\varphi'(\alpha) = 0$ **et** $\varphi''(\alpha) \neq 0$ **:**

$$\epsilon_{k+1} \simeq -\frac{\varphi''(\alpha)}{2}\epsilon_k^2$$

Le processus itératif est d'ordre 2, on a donc une convergence quadratique.

Si $\varphi'(\alpha) = \varphi''(\alpha) = 0$ **et** $\varphi'''(\alpha) \neq 0$ **:**

$$\epsilon_{k+1} \simeq \frac{\varphi'''(\alpha)}{6}\epsilon_k^3$$

Le processus itératif est d'ordre 3, on a donc une convergence cubique.

Etc etc...Il peut aussi y avoir des ordre de convergence qui ne sont pas des entiers naturels (cf méthode de la sécante).

1.6.2 Nombre de décimales justes lors du calcul de la racine

On a l'erreur $\epsilon_k = \alpha - x_k$ que l'on va écrire sous la forme

$$\epsilon_k = w_k 10^{-n_k}$$

avec :

. $0 < w_k < 1$

. n_k : nombre de décimales justes dans la valeur x_k approchée de α

On définit ensuite la quantité

$$S_k = -log_{10}(\epsilon_k)$$

On calcule alors que :

$$S_k = -log_{10}(\epsilon_k) = -log_{10}(w_k) + n_k$$

Comme on a : $0 < w_k < 1$, comme par exemple pour typiquement $n_k = 6$ et typiquement $n_k \simeq 0,5$, on obtient alors $log_{10}(w_k) \simeq -0,3 << 6$ donc $log_{10}(w_k)$ est négligeable. De plus, plus k augmente, plus n_k augmente et donc très rapidement : $n_k >> -log_{10}(w_k)$. On en conclut que :

$$S_k \simeq n_k$$

Dans un processus d'ordre m :

$$\epsilon_{k+1} = C\epsilon_k^m$$

Ainsi,

$$S_{k+1} = -log_{10}(\epsilon_{k+1}) = -log_{10}(C) - mlog_{10(\epsilon_k)}$$

$$S_{k+1} = mS_k - log_{10}(C)$$

Pour k assez grands, on aura ϵ_k assez faible, et donc S_k assez grand, et ainsi $--log_{10}(C)$ sera négligeable devant mS_k. On en déduit :

$$S_{k+1} \simeq mS_k$$

Or, on sait que dans ces conditions, on a : $S_k \simeq n_k$, donc on en déduit que :

$$\boxed{n_{k+1} \simeq m.n_k} \tag{1.3}$$

soit encore :

$$n_k = m^k n_0$$

Conclusion : Dans un processus d'ordre m, le nombre de décimales justes dans la suite x_k est multiplié par m à chaque itération !

1.6.3 Comparaison de deux processus différents. Efficacité d'un processus

On sait que $S_{k+1} = mS_k$ donc $S_k = m^k S_0$. Soient deux processus :

* $S^{(1)}$: un processus d'ordre m_1
* $S^{(2)}$: un processus d'ordre m_2

Hypothèse : on va les comparer pour un même résultat obtenu à partir des mêmes conditions initiales.

On a : $S_k^{(1)} = m_1^k S_0^{(1)}$ et $S_k^{(2)} = m_2^k S_0^{(2)}$. Donc, d'après l'hypothèse ; il faut, pour $S_0^{(1)} = S_0^{(2)}$, que l'on obtienne : $S_{k_1}^{(1)} = S_{k_2}^{(2)}$. On en déduit que :

$$m_1^{k_1} = m_2^{k_2}$$

$$k_1 log_{10}(m_1) = k_2 log_{10}(m_2)$$

$$\frac{k_1}{k_2} = \frac{log_{10}(m_2)}{log_{10}(m_1)}$$

Soit :

. θ : le travail par itération (nombre d'opérations pendant une itération).

. T : le travail total (pour obtenir le résultat final).

On a évidemment $T_1 = k_1\theta_1$ et $T_2 = k_2\theta_2$ et :

$$\frac{T_1}{T_2} = \frac{k_1\theta_1}{k_2\theta_2} = \frac{log_{10}(m_2)}{log_{10}(m_1)}\frac{\theta_1}{\theta_2} = \frac{\frac{1}{\theta_2}log_{10}(m_2)}{\frac{1}{\theta_1}log_{10}(m_1)} = \frac{log_{10}(m_2^{\frac{1}{\theta_2}})}{log_{10}(m_1^{\frac{1}{\theta_1}})}$$

On définit l'efficacité E d'un processus par une fonction décroissante de T nécessairement. Par exemple, $E = 10^{\frac{1}{T}}$. On a alors $T = \frac{1}{log_{10}(E)}$. Ainsi :

$$\frac{T_1}{T_2} = \frac{log_{10}(E_2)}{log_{10}(E_1)}$$

Ce qui fournit immédiatement le résultat suivant pour l'efficacité d'un processus :

$$\boxed{E = m^{\frac{1}{\theta}}} \tag{1.4}$$

avec :

. m = ordre du processus ($\epsilon_{k+1} = C\epsilon_k^m$)

. θ = travail par itération du processus (nombre d'opérations pendant une itération)

1.6.4 Application : méthode de Newton-Raphson

On utilise le résultat très général indépendant de la méthode employée, déjà démontré :

$$\epsilon_{k+1} = \varphi'(\alpha)\epsilon_k - \frac{\varphi''(\alpha)}{2!}\epsilon_k^2 + \frac{\varphi'''(\alpha)}{3!}\epsilon_k^3 + ...$$

avec ici :

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

$$\varphi'(x) = 1 - \frac{f'^2(x) - f(x)f''(x)}{f'^2(x)}$$

donc, sachant que par définition, $f(\alpha) = 0$, il vient $\varphi'(\alpha) = 1 - 1 = 0$
Pour calculer ensuite $\varphi''(\alpha)$, on écrit :

$$\varphi'(x) = 1 - \frac{f'^2(x) - f(x)f''(x)}{f'^2(x)} = 1 - 1 + \frac{f(x)f''(x)}{f'^2(x)} = \frac{f(x)f''(x)}{f'^2(x)}$$

On pose

$$\Psi(x) = \frac{f''(x)}{f'^2(x)}$$

, donc

$$\varphi'(x) = f(x)\Psi(x)$$

Ainsi,

$$\varphi''(x) = f'(x)\Psi(x) + f(x)\Psi'(x)$$

$$\varphi''(\alpha) = f'(\alpha)\Psi(\alpha) + 0 = f'(\alpha)\frac{f''(\alpha)}{f'^2(\alpha)} = \frac{f''(\alpha)}{f'(\alpha)} \neq 0$$

Donc,

$$\boxed{\epsilon_{k+1} = -\frac{f''(\alpha)}{2f'(\alpha)}\epsilon_k^2} \tag{1.5}$$

L'algorithme de Newton-Raphson est donc un processus d'ordre $m = 2$ et :

$$n_{k+1} \simeq 2n_k$$

Conclusion : Le nombre de décimales justes est doublé à chaque itération.