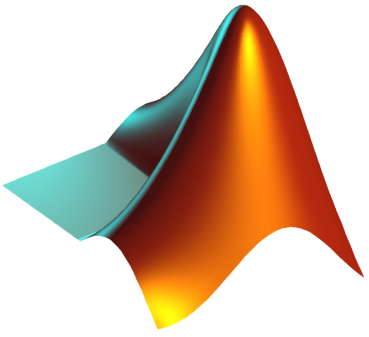
ECOLE SUPERIEURE D'INGENIEURS DE RENNES

UNIVERSITÉ DE RENNES 1

CUPGE 1 - ESIR
TP
MATHS S2

RÉSOLUTION D'ÉQUATIONS NON LINÉAIRES

COMPTE-RENDU



Groupe D6 - VERDIER Éric, THIÉBAUD Malo

Mention sur le plagiat

Article L-122-4 du Code de la propriété intellectuelle[1] :

"Toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite. Il en est de même pour la traduction, l'adaptation ou la transformation, l'arrangement ou la reproduction par un art ou un procédé quelconque."

Version en vigueur depuis le 03 juillet 1992

Nous attestons que le contenu de ce document est original et est issu de nos réflexions personnelles

Vous trouverez en fin de document une bibliographie de toutes les sources utilisées.

Sommaire

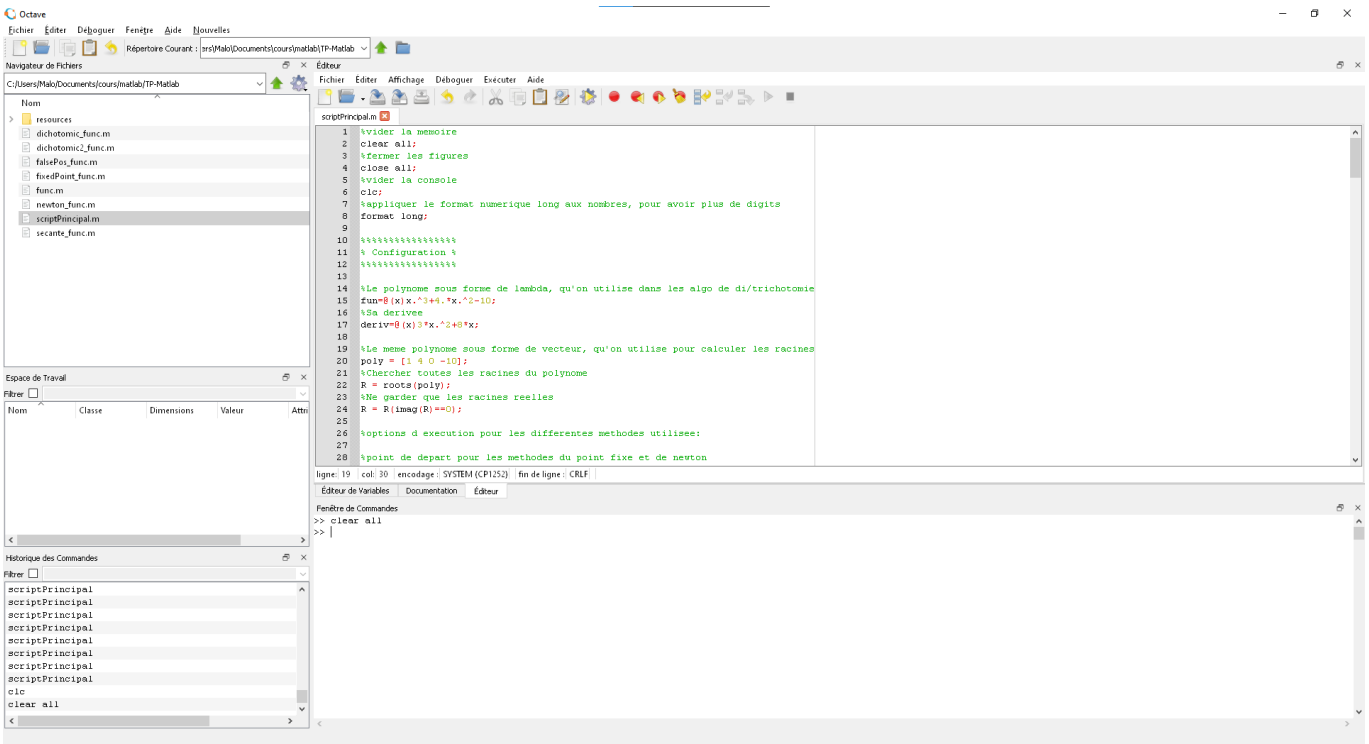
Introduction

En cours de Mathématiques, nous avons jusqu'ici eu l'habitude de pouvoir résoudre analytiquement l'intégralité de nos problèmes. De part diverses démonstrations et théorèmes, nous étions capable de résoudre de façon exacte des équations et d'autres types de problème. Cela dit, il peut arriver que nous arrivions face à des problèmes que nous ne savons pas résoudre analytiquement. Entre alors en jeu une nouvelle partie des mathématiques : l'algorithmique. L'algorithmique est une discipline visant à définir une suite d'instructions claires et précises dans le but de résoudre un problème[2]. Une analogie récurrente de l'algorithmique est la recette de cuisine[3] : vous possédez des ingrédients en entrée, les utilisez à travers différentes étapes et finissez par obtenir votre plat en résultat. Dans le cadre de ce TP, l'algorithmique va devoir nous permettre de rechercher les solutions d'équations non linéaires et de déterminer la manière la plus efficace de le faire. Nous nous focaliserons sur la recherche de zéros (x tel que pour une fonction f, f(x) = 0) de fonctions pour lesquelles nous ne saurions, à notre niveau, le faire analytiquement. Trouver le zéro de fonctions peut être utile dans le cas de recherche d'extremum de fonctions (chercher les points où la dérivée d'une fonction s'annule), ou encore pour approximer les valeurs exactes de nombres irrationnels (rechercher la valeur de $\sqrt{2}$ revient à chercher la valeur de x tel quel $x^2-2 = 0$). Il existe cependant une multitude de manières d'implémenter cet algorithme, chacune ayant ses avantages et inconvénients. Il n'est par exemple pas toujours simple de réussir à faire converger notre algorithme, on peut parfois se retrouver dans des cas de figures qui vont entraîner une erreur, il n'est pas toujours possible de transformer la fonction que l'on souhaite étudier comme il le faudrait car on ne connaît tout simplement pas son écriture analytique, … Durant ce TP, nous allons mettre en place plusieurs de ces algorithmes et les tester avec différentes valeurs de départ de façon à comparer le temps d'exécution, la vitesse de convergence et la stabilité des différentes méthodes étudiées. Cela nous permettra de dire dans quel cas de figure telle ou telle méthode est à privilégier.

Méthodes et matériels

MATLAB et GNU Octave

Pour exécuter nos méthodes, nous avions besoin d'un langage de programmation qui puisse nous permettre de générer des courbes afin de pouvoir étudier nos résultats.
C'est pourquoi nous avons utilisé le langage Matlab[4]. En effet, Matlab est un langage de programmation scientifique qui permet de réaliser différentes opérations de calcul numériques. Il permet par exemple de générer des courbes, d'étudier des fonctions, de créer des matrices etc…
Il a été créé en 1984 et est basé sur le calcul matriciel. Il est utilisable dans un environnement de développement (EDI) portant le même nom.
Néanmoins, le logiciel MATLAB est un logiciel payant, nous empêchant donc de pouvoir travailler sur nos algorithmes ailleurs qu'à l'université.
Nous avons donc téléchargé la version libre de MATLAB : GNU Octave[5]. Octave est un logiciel similaire à MATLAB, nous permettant donc de commencer nos algorithmes en salle de TP sur le logiciel MATLAB, puis de les améliorer chez nous sur Octave.



aperçu de l'interface de GNU Octave

Méthode de la dichotomie

La méthode de la dichotomie consiste à prendre un intervalle [a , b] où f(a) et f(b) ne sont pas du même signe et où la fonction f est continue sur [a , b]. De cette façon, on a bien d'après le théorème des valeurs intermédiaire le point où f(x)=0.
Pour qu'il n'y ait qu'une racine dans notre intervalle, il faut que la fonction soit strictement monotone sur notre intervalle.
Une fois l'intervalle choisi, on définit un troisième point, c, tel que

$$c = \frac{a+b}{2}.$$

Ensuite on compare le signe des images des différents points :
Si on a f(c) et f(a) de signes opposés alors on met b <- c
Si on a f(c) et f(b) de signes opposés alors on met a <- c

L'erreur est calculée par err = $|c_n - c|$ où c est la valeur exacte de la racine

Cela fait une itération, puis on recommence jusqu'à ce que notre erreur soit inférieure à la tolérance.
En sortie, la variable c est donc notre approximation de la racine.

Cette méthode présente comme principal défaut la lenteur. En effet, cette fonction peut, en fonction de la taille de l'intervalle ou de la tolérance recherchée, prendre beaucoup de d'itérations avant d'atteindre la racine.

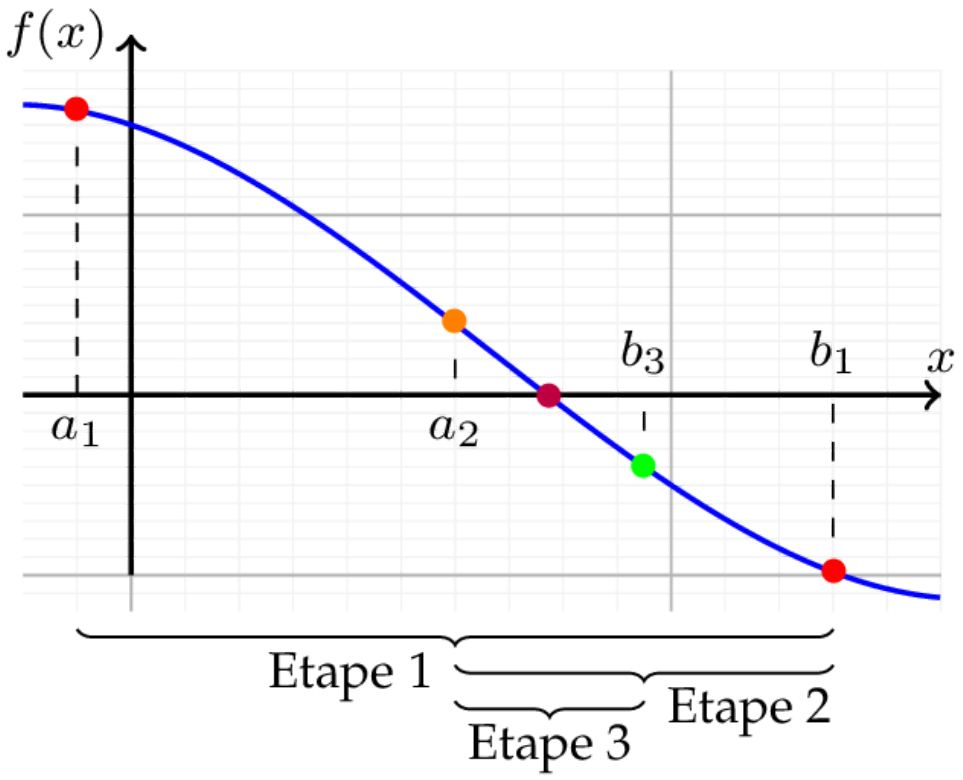


image illustrative de la dichotomie

Lien vers le code matlab de dichotomic_func.m
pseudo code

```
%Inputs :
%    -> a - Float - Borne inférieure de l intervalle
%    -> b - Float  - Borne supérieure de l intervalle
%    -> tol - Float - critère d arrêt
%    -> iterMax - Int - Maximum d itérations de notre algorithme
%    -> trueValue - Float - le zero calcule par matlab du polynôme
%    -> fun - handle - le polynôme
%Outputs :
%    -> xFinal - Float - L'approximation de notre racine
%    -> i - Int - Nombre d'itérations nécessaire pour trouver la bonne valeur approchée
%    -> err - [Float] - Valeur de l'erreur entre l élément calculé et la véritable valeur
%Variables:
%    -> c - Float - Approximation du zéro
%    -> FA - Float - Image de a
%    -> FC - Float - Image de c

set i = 1
set FA = fun(a)
set err = []
while ( i <= iterMax ) do
       set c = a + (b-a) / 2
       set FC = fun(c)
       set err = [ err, |c-trueValue| ]
       if ( |FC| < tol ) then
              set xFinal = c
              return
       endif
       if ( FA * FC > 0) then
              set a = c
              set FA = FC
       else
              set b = c
       endif
       set i = i + 1
endwhile
set xFinal = c
```

Méthode de la trichotomie

La méthode de la trichotomie est similaire dans son fonctionnement à la méthode de dichotomie. Elle consiste à prendre un intervalle [a , b] où f(a) et f(b) ne sont pas du même signe et où la fonction f est continue sur [a , b]. De cette façon, on a bien d'après le théorème des valeurs intermédiaire le point où f(x)=0.
Pour qu'il n'y ait qu'une racine dans notre intervalle, il faut que la fonction soit strictement monotone sur notre intervalle.
Une fois l'intervalle choisi, on définit un troisième et quatrième point, c_1 et c_2, tel que

$$c_1 = \frac{a+b}{3} \text{ et } c_2 = \frac{2(a+b)}{3}.$$

Ensuite on compare le signe des images des différents points :

Si on a f(c_1) et f(a) de signes opposés, alors on met b <- c_1
Si on a f(c_2) et f(b) de signes opposés alors on met b <- c_2
Si on a f(c_1) et f(c_2) de signes opposés alors on met a <- c_1 et b <- c_2

L'erreur est calculée par err = |c_1 - c| ou err = |c_2 - c| (avec c la valeure exacte de la racine) ou encore err=$\frac{(|c_1-c|+|c_2-c|)}{2}$, en fonction des différents cas de figures.

Cela fait une itération, puis on recommence jusqu'à ce que notre erreur soit inférieure à la tolérance.
En sortie, c_1 et c_2 sont donc deux approximations de la racine, on choisira la meilleure des deux.

Cette méthode est censée être plus rapide que la dichotomie, mais nous allons devoir nous en assurer.

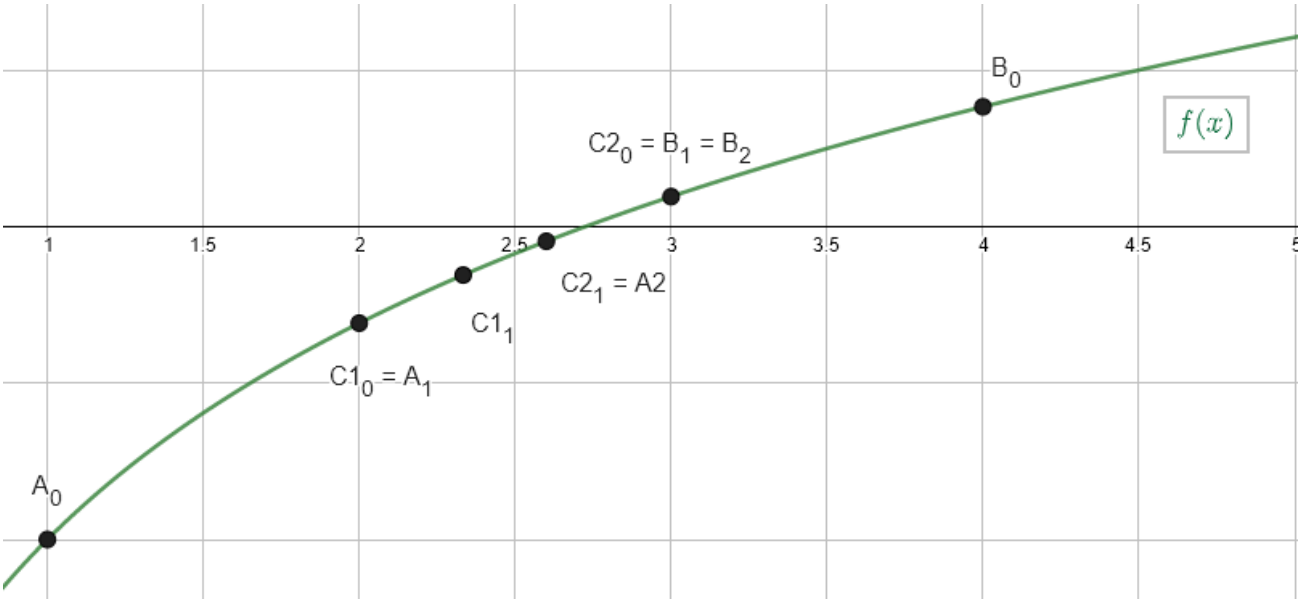
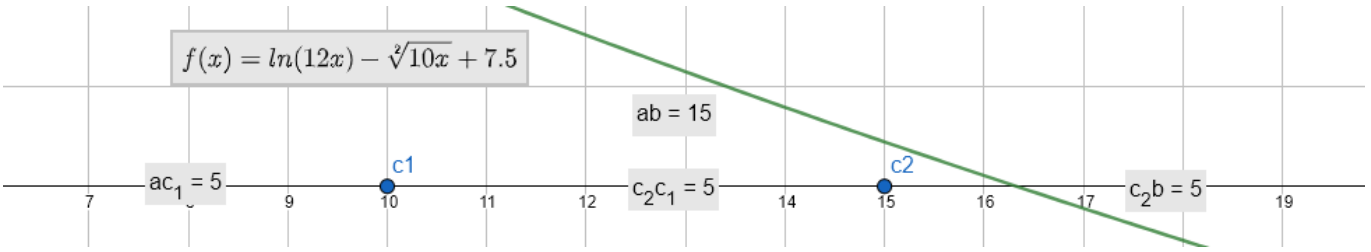


image illustrative de la trichotomie

Convergence de la trichotomie

Pour évaluer la convergence de la trichotomie, on va s'intéresser à l'évolution de son intervalle de recherche au fur et à mesure des itérations n.

à chaque itération, on divise l'intervalle de recherche par 3. [a,b] devient [a,c1], [c1,c2] ou [c2,b]



Ainsi, on peut majorer l'erreur |cn - c| de la sorte:

$\left|c_1 - c\right| \leq \frac{b-a}{3}$; $\left|c_2 - c\right| \leq \frac{\frac{b-a}{3}}{3}$; $\left|c_3 - c\right| \leq \frac{\frac{\frac{b-a}{3}}{3}}{3}$; et ainsi de suite…

Par récurrence on trouve l'inéquation $\left|c_n - c\right| \leq \frac{b-a}{3^n}$

Grâce au cours, nous savons que dans le cas de la dichotomie, l'erreur est majorable par $\frac{b-a}{2^n}$

Il est alors possible de comparer les vitesses de convergence de la dichotomie et de la trichotomie.

En effet, si pour une suite U_n et une suite V_n on a $\lim_{n \to \infty} \frac{U_n}{V_n} = 0$,

On peut alors dire que la suite U_n converge plus vite vers que (V_n).

Soient $U_n = \frac{b-a}{3^n}$ et $V_n = \frac{b-a}{2^n}$

On a $\frac{U_n}{V_n} = \frac{b-a}{3^n} \div \frac{b-a}{2^n} = \frac{b-a}{3^n} \times \frac{2^n}{b-a} = \frac{2^n}{3^n} = (\frac{2}{3})^n$

$0 < \frac{2}{3} < 1$ donc $\lim_{n \to \infty} (\frac{2}{3})^n = \lim_{n \to \infty} \frac{U_n}{V_n} = 0$

On peut donc conclure que la méthode de la trichotomie converge analytiquement plus vite vers 0 que la méthode de la dichotomie. Il sera possible de comparer ce résultat à la mise en pratique plus tard.

Lien vers le code matlab de [dichotomic2_func.m]
pseudo code

```
%Inputs :
%    -> a - Float - Borne inférieure de l'intervalle
%    -> b - Float - Borne supérieure de l'intervalle
%    -> tol - Float - critère d'arrêt
%    -> iterMax - Int - Maximum d'itérations de notre algorithme
%    -> trueValue - Float - le zero calcule par matlab du polynôme
%    -> fun - handle - le polynôme
% Outputs :
%    -> xFinal - Float - L'approximation de notre racine
%    -> i - Int - Nombre d'itérations nécessaire pour trouver la bonne valeur approchée
%    -> err - [Float] - Valeur de l erreur entre l'élément calculé et la véritable valeur
%Variables:
%    -> c1 - Float - Approximation du zéro au premier tier de [a,b]
%    -> c2 - Float - Approximation du zéro au deuxième tier de [a,b]
%    -> FA - Float - Image de a
%    -> FC1 - Float - Image de c1
%    -> FC2 - Float - Image de c2

set i = 1
set FA = fun(a)
set err = []
while ( i <= iterMax ) do
      set c1 = a + (b-a) / 3
      set c2 = a + 2*(b-a) / 3
      set FC1 = fun(c1)
      set FC2 = fun(c2)
      if ( FA * FC1 < 0 ) then
            set b = c1
            set err = [ err, |c1 - trueValue| ]
      else if ( FC1 * FC2 < 0 ) then
            set a = c1
            set FA = FC1
            set b = c2
            err = [ err, ( |c1 - trueValue| + |c2 - trueValue| / 2 ]
      else
            set a = c2
            set FA = FC2
            set err = [ err, |c2-trueValue| ]
      endif
      if ( |FC1| <= tol ) then
            set xFinal = c1
            return
      else if (|FC2| <= tol ) then
            set xFinal = c2
            return
      endif
      set i = i + 1
endwhile
set xFinal = (c1 + c2) / 2
```

Méthode du point fixe

La méthode du point fixe utilise des fonctions g(x) issues de f(x) telle que g(x)=x. On transforme une recherche de racine de type f(x)=0 en une recherche de point fixe de type g(x)=x.
Il suffit ensuite de fixer un point de départ c_0 et que, à chaque tour de boucle on ait : $c_{n+1}=f(c_n)$.
Pour que la méthode fonctionne et converge vers le point recherché, il faut que la fonction respecte le théorème d'existence du point fixe et le théorème d'unicité du point fixe.
On retourne le résultat quand $|f(c_{n+1})-f(c_n)|$ est inférieur à la tolérance.
On calcul l'erreur err = $|c_n-c|$ où c est la valeur exacte de la racine.

Théorème d'existence du point fixe :
Si g est une fonction continue sur [a , b] et si g(x) ∈ [a , b] pour tout x ∈ [a , b], alors g possède au moins un point fixe dans [a , b].
Théorème d'unicité du point fixe :
Si g est dérivable sur [a , b] et si il existe une constante k ∈ [0 , 1 [telle que $|g'(x)| \leq k$ pour tout x ∈ [a , b], alors ce point fixe de [a , b] est unique.[6]

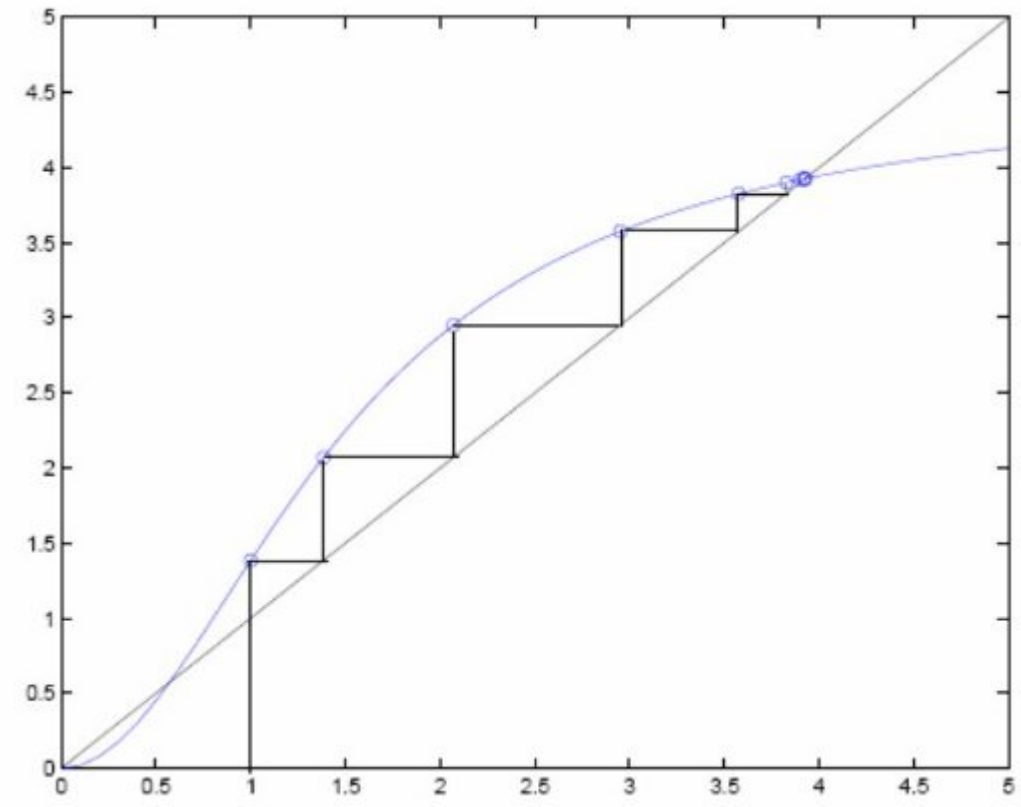


Image illustrative de la méthode du point fixe

Recherche de fonctions

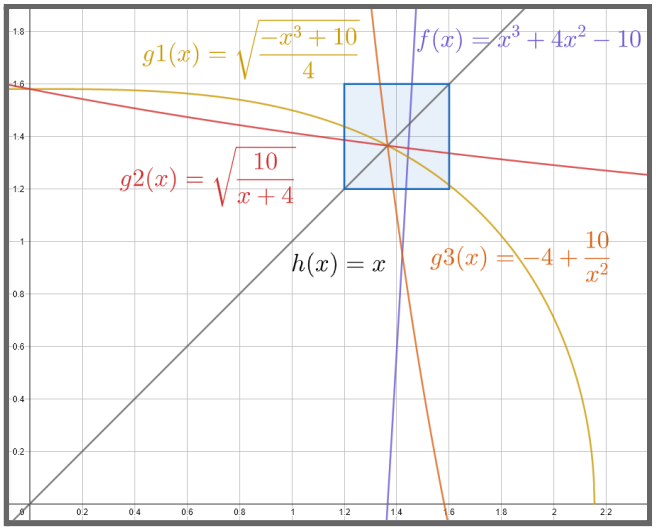
En partant du polynôme proposé dans le sujet $f(x) = x^3 + 4x^2 - 10$, on va donc chercher des fonctions g(x) telles que $g(x) = x$. Pour ça on va transformer l'équation $f(x) = 0$.

$x^3 + 4x^2 - 10 = 0 \Rightarrow 4x^2 = -x^3 + 10 \Rightarrow x = \sqrt{(-x^3 + 10)/4} = g1(x)$ définie sur $]-\infty ; \sqrt[3]{10}[$

$x^3 + 4x^2 - 10 = 0 \Rightarrow x^2(x + 4) = 10 \Rightarrow x = \sqrt{10/(x + 4)} = g2(x)$ définie sur $]-4 ; +\infty[$

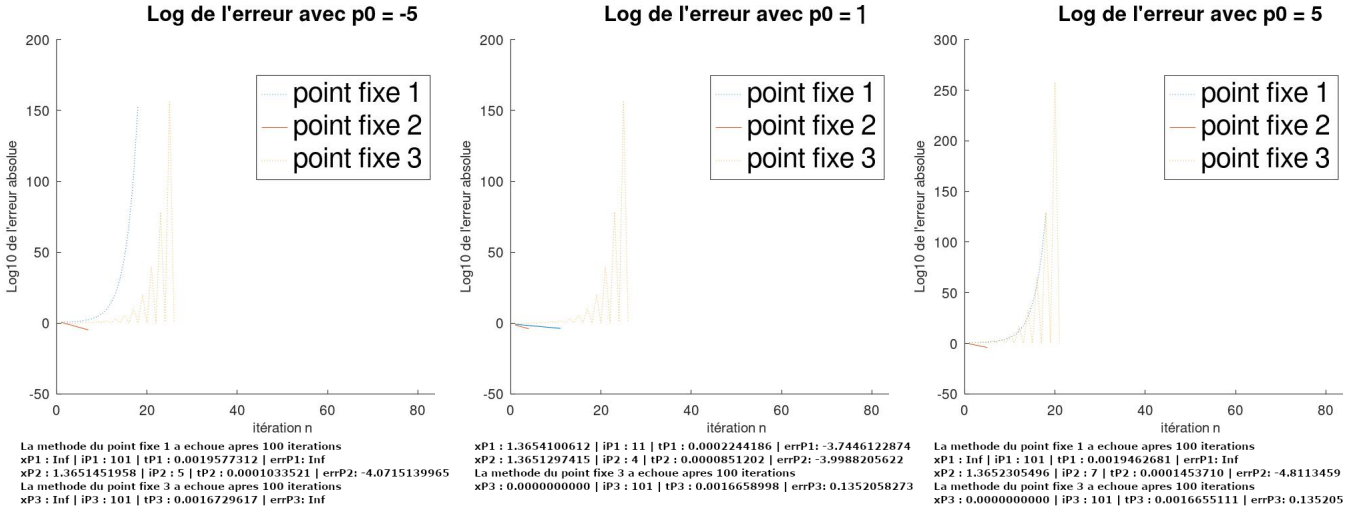
$x^3 + 4x^2 - 10 = 0 \Rightarrow x^3 = -4x^2 + 10 \Rightarrow x = -4 + 10/x^2 = g3(x)$ définie sur $\mathbb{R}^*$

On trace nos trois fonctions sur geogebra pour vérifier que la racine de f(x) coïncide bien avec les points fixes de ces fonctions.
le théorème d'existence du point fixe nous dit que pour un x appartenant a un intervalle [a,b], il faut que g(x) appartienne à cet intervalle (cela correspond au carré bleu sur l'image). Ici on voit que g1(x) est à la limite de cette règle, que g2(x) entre parfaitement dans l'intervalle et que g3(x) pas du tout.



Fonctions représentées sur Géogébra
Image originale

En testant nos fonctions grâce à l'algorithme du point fixe, nous avons pu observer que nos trois fonctions ne convergent pas toujours vers le zéro. En effet, dès lors que nous nous éloignons trop de 1.365 (valeur approchée de la racine), g1 à tendance à diverger. g2 quant à elle continue de converger à chaque fois mais g3 ne converge jamais. On peut très probablement relier ces observations avec le constat fait plus tôt vis-à-vis de la règle d'existence du point fixe. De part ces observations, nous garderons g2, qui est la plus stable des trois fonctions, pour comparer la méthode du point fixe avec les autres.



Erreurs à mesure de l'exécution de l'algorithme du point fixe pour différentes fonctions et différents p0.
Image originale

Lien vers le code matlab de [fixedPoint_func.m]
pseudo code

```
%Inputs:
%    -> fun - handle - pointeur de fonction à traiter
%    -> p0 - Float - approximation initiale
%    -> tol - Float - critère d'arrêt
%    -> iterMax - Int - Maximum d'itérations de notre algorithme
%    -> trueValue - Float - véritable valeur de la racine
%Outputs:
%    -> xFinal - Float - L'approximation de notre racine
%    -> i - Int - Nombre d'itérations nécessaire pour trouver la bonne valeur approchée
%    -> err - [Float] - Valeur de l'erreur entre l'élément calculé et la véritable valeur
%Variables:
%    -> old - Float - Une variable temporaire utilisée pour mesure la convergence

set err = []
set i = 1
while ( i <= iterMax ) do
      set old = p0
      set p0 = fun(p0)
      err = [ err, |p0-trueValue| ]
      if ( |p0-old| <= tol ) then
            set xFinal = p0
            return
      endif
      set i  = i + 1
endwhile
set xFinal = p0
```

Méthode de Newton

La méthode de newton est identique à la méthode du point dans son fonctionnement.
La différence se fait au niveau de de la recherche de la fonction g(x).
Dans la méthode de Newton, la fonction $g(x) = x - \frac{f(x)}{f'(x)}$

On a donc $c_{n+1} = c_n + \frac{f(c_n)}{f'(c_n)}$

On retourne le résultat quand $|f(c_{n+1})-f(c_n)|$ est inférieur à la tolérance.
On calcul l'erreur err = $|c_n-c|$ où c est la valeur exacte de la racine.

Cette méthode a l'avantage d'être rapide et fiable. Néanmoins, elle ne peut pas toujours être mise en place car certaines fonctions n'ont pas de dérivées connues.

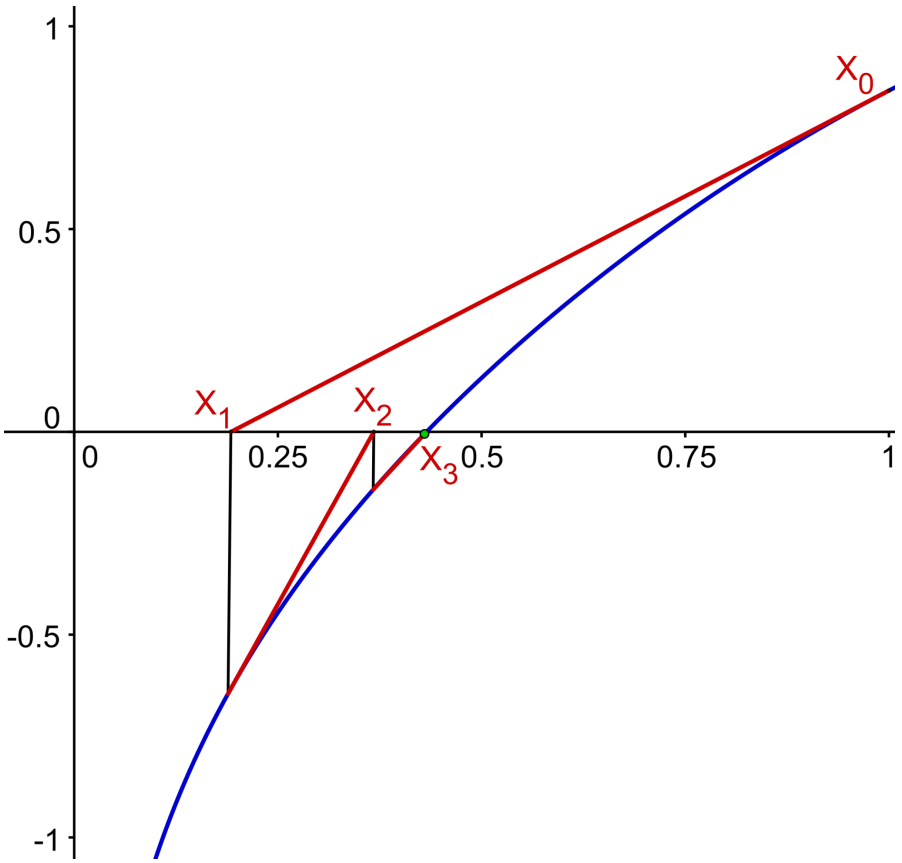


Image illustrative de la méthode de Newton

Lien vers le code matlab de [newton_func.m]
pseudo code

```
%Inputs:
%    -> fun - handle - Pointeur de fonction à traiter
%    -> deriv - handle - Pointeur sur la dérivée de fonction à traiter
%    -> p0 - Float - premiere approximation
%    -> tol - Float - critere d arret
%    -> iterMax - Int - Maximum d'itérations de notre algorithme
%    -> trueValue - Float - véritable valeur de la racine
%Outputs :
%    -> xFinal - Float - L'approximation de notre racine
%    -> i - Int - Nombre d'itérations nécessaire pour trouver la bonne valeur approchée
%    -> err - [Float] - Valeur de l'erreur entre l'élément calculé et la véritable valeur

set err = []
set i = 1
while ( i <= iterMax ) do
       set p0 = p0 - fun(p0)/deriv(p0)
       set err = [ err, |p0-trueValue| ]
       if ( |fun(p0) <= tol ) then
              set xFinal = p0
              return
       endif
       set i = i + 1
endwhile
set xFinal = p0
```

Méthode de la sécante

La méthode de la sécante est une méthode itérative basée sur la méthode du point fixe. Elle est très proche de la méthode de Newton dans son implémentation, à l'exception qu'au lieu d'utiliser $f'(n)$ dans sa suite convergente, elle utilise la formule du taux d'accroissement $\frac{f(b)-f(a)}{b-a}$ où b et a sont deux nombres différents mais suffisamment proches l'un de l'autre de façon à ce que le taux d'accroissement calculé approxime une tangente de la fonction. Contrairement à la méthode de Newton, cette méthode ne requiert pas la connaissance de la dérivée de la fonction, ce qui peut être utile quand on ne connaît pas l'écriture analytique de la fonction étudiée (en cas de mesure par capteurs par exemple).

La méthode de la sécante utilise donc le même algorithme que la méthode de Newton ou que la méthode du point fixe,

en remplaçant toutefois la suite de la méthode de Newton $c_{n+1} = c_n - \frac{f(c_n)}{f'(c_n)}$ par la suite

$$c_{n+1} = c_n - \frac{f(c_n)}{\frac{f(c_n)-f(c_{n-1})}{c_n-c_{n-1}}} = c_n - f(c_n) * \frac{c_n-c_{n-1}}{f(c_n)-f(c_{n-1})}$$

On itère ainsi jusqu'à ce qu'on ait convergé vers le zéro, c'est à dire quand $|f(c_n)|$ est inférieur à notre tolérance.
On calcule l'erreur à la n-ième itération avec $err_n = |c_n - c|$ où c est la valeur exacte de la racine

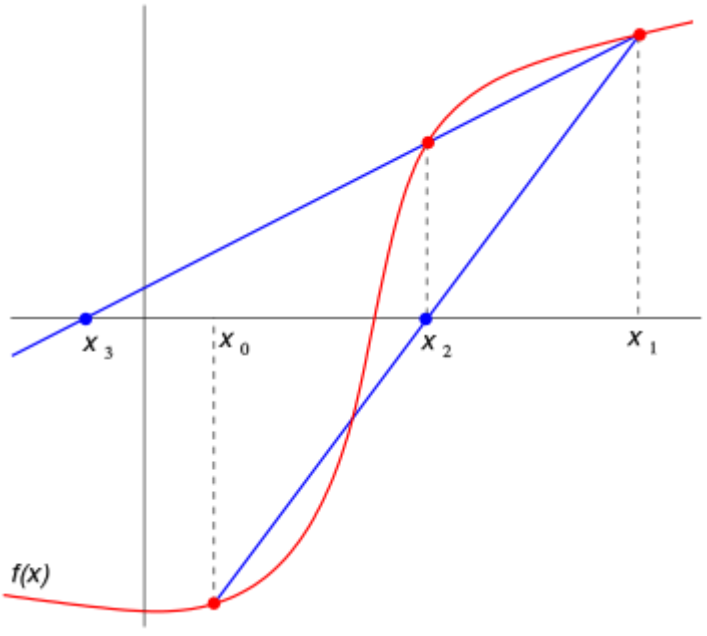


image Illustrative de la méthode de la sécante

Lien vers le code matlab de secante_func.m
pseudo code

```
%Inputs:
%   -> fun - handle - Pointeur de fonction à traiter
%   -> a - Float - Borne inf de l intervalle de recherche
%   -> b - Float - Borne sup de l intervalle de recherche
%   -> iterMax - Int - Maximum d'itérations de notre algorithme
%   -> tol - Float - critere d arret
%   -> trueValue - Float - véritable valeur de la racine
%Outputs:
%    -> xFinal - Float - L'approximation de notre racine
%    -> i - Int - Nombre d'itérations nécessaire pour trouver la bonne valeur
%    -> err - [Float] - Valeur de l'erreur entre l'élément calculé et la véritable valeur
%Variables:
%    -> tmp - Float - Variable temporaire pour stocker une ancienne valeur
%    -> FB - Float - Image de b
%    -> FpB - Float - taux d'accroissement entre a et b

set i = 1
set err = []
if ( a == b ) then
      error("les deux points de départ ne peuvent pas être égaux")
endif
while ( i <= iterMax ) do
      set tmp = b
      set FB = fun(b)
      set FpB = (FB-fun(a)) / (b-a)
      set b = b - FB/FpB
      set a = tmp
      set err = [ err, |b-trueValue| ]
      if ( |fun(b)| < tol ) then
            set xFinal = b
            return
      endif
      set i = i + 1
endwhile
set xFinal = b
```

Méthode de la fausse position

La méthode de la fausse position, ou méthode de Lagrange[7], est encore une fois une méthode basée sur la méthode du point fixe. Elle est, pour ainsi dire, un mélange entre la méthode de la sécante et la méthode de la dichotomie. En effet, elle consiste en l'utilisation d'une suite convergente vers un point fixe correspondant au zéro de la fonction étudier. Pour cela, elle utilise une variante de la suite de la méthode de la sécante : $c_n = b_n - f(b_n) * \frac{b_n - a_n}{f(a_n) - f(a_n)}$.

Là ou les deux méthodes se distinguent est le fait que la méthode de la fausse position n'utilise pas systématiquement les 2 derniers points calculés, mais va utiliser des points a et b (de signes opposés) et les mettre à jour de la façon suivante :
si $f(c_n) * f(a_n) > 0$ alors $a_{n+1} = c_n$, si non $b_{n+1} = c_n$ [8]
Ainsi, il n'est pas possible de dépasser le zéro recherché et de diverger (en cas de point d'inflection entre a et b par exemple).

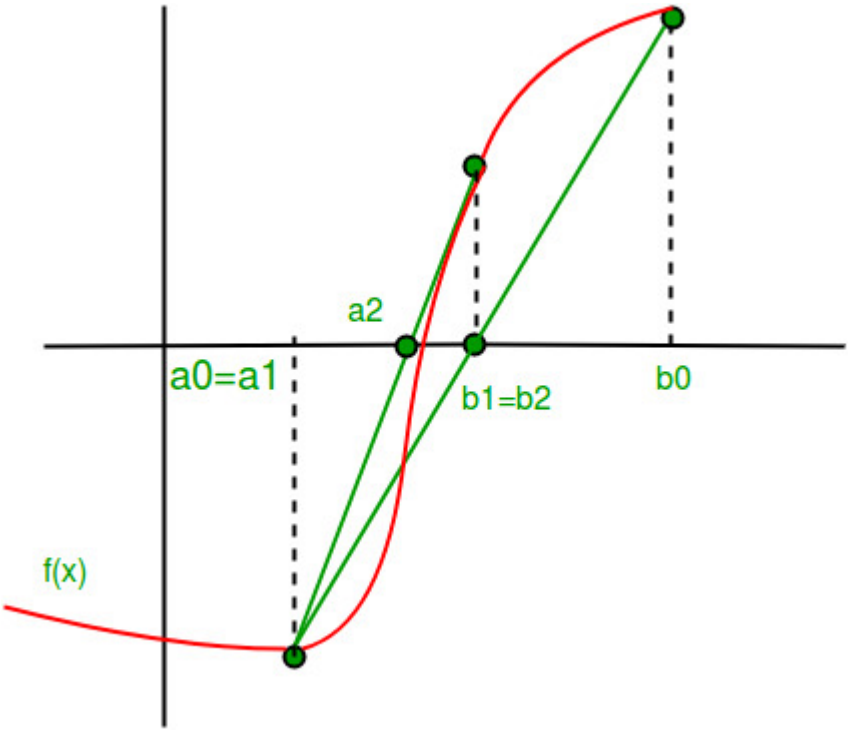


Image Illustrative de la méthode de la fausse position

Lien vers le code matlab de [falsePos_func.m]
pseudo code

```
%Inputs:
%    -> fun - handle - Pointeur de fonction à traiter
%    -> a - Float - Borne inf de l intervalle de recherche
%    -> b - Float - Borne sup de l intervalle de recherche
%    -> iterMax - Int - Maximum d'itérations de notre algorithme
%    -> tol - Float - critere d arret
%    -> trueValue - Float - véritable valeur de la racine
%Outputs:
%    -> xFinal - Float - L'approximation de notre racine
%    -> i - Int - Nombre d'itérations nécessaire pour trouver la bonne valeur approchée
%    -> err - [Float] - Valeur de l'erreur entre l'élément calculé et la véritable valeur
%Variables:
%    -> c - Float - Approximation du zéro
%    -> FB - Float - Image de b
%    -> Fp - Float - Taux d'accroissement entre a et b
%    -> FC - image de c

set i = 1
set err = []

if ( a == b ) then
      error("les deux points de depart de ne peuvent pas être égaux")
endif

while ( i <= iterMax) do
      set FB = fun(b)
      set Fp = (FB-fun(a)) / (b-a)
      set c = b - FB/Fp
      set err = [ err, |c-trueValue| ]
      if ( |FC| < tol ) then
            set xFinal = c
            return
      endif
      if ( FC * FB > 0 ) then
            set b = c
      else
            set a = c
      endif
      set i = i + 1
endwhile
set xFinal = c
```

Résultats

Tableau récapitulatif des résultats obtenus après exécution des méthodes

	Approximation du zéro	Nombre d'itérations	Temps moyenne (*ms*)	Erreur en log10
Dichotomie	1.3652038574	15	0,4074693	-4,5
Trichotomie	1.3652221037	10	0,3889990	-4,3
Point fixe : g1(x)	1.3654100612	11	0,2325392	-3,7
Point fixe : g2(x)	1.3651297415	4	0,0845408	-3,9
Newton	1.3652366002	3	0,0968289	-5,1
Sécante	1.3652300134	11	0,2994394	-6,7
Fausse position	1.3651725483	33	0,9606910	-4,2

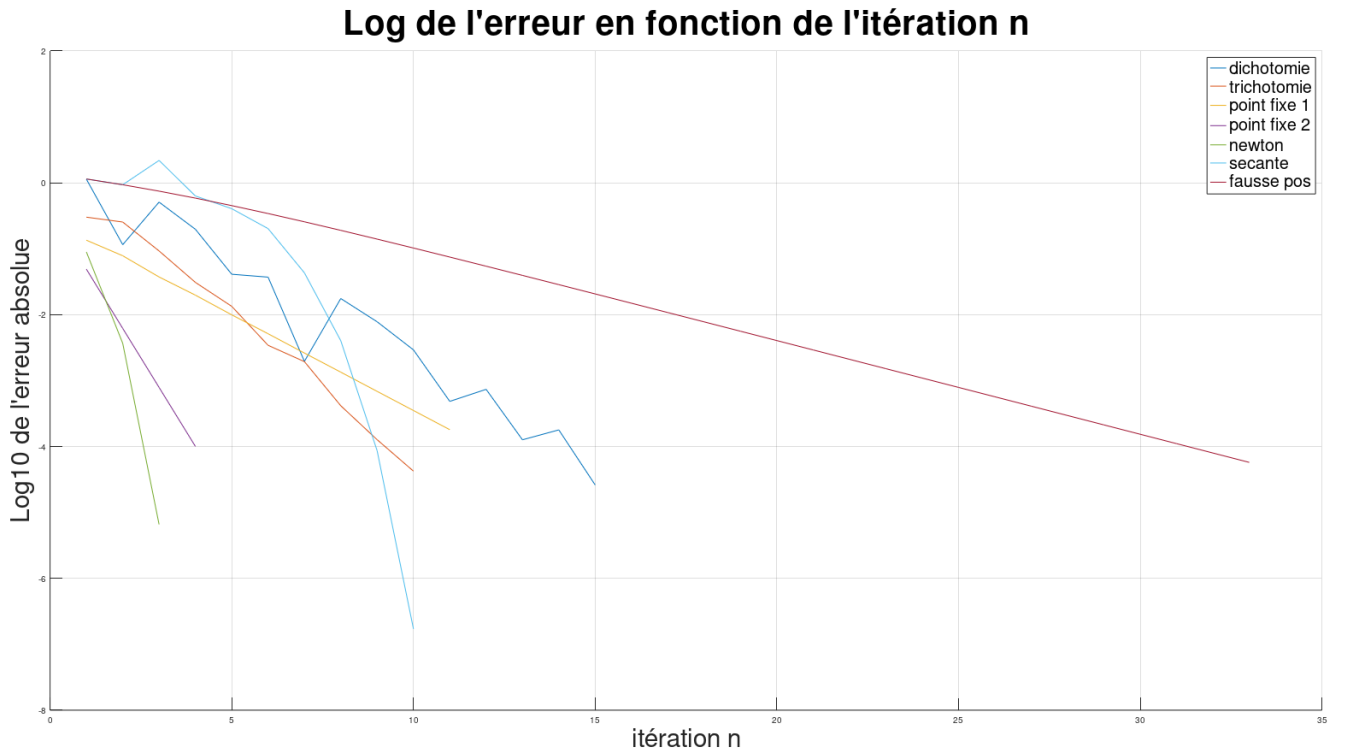


image originale

On observe dans le tableau que toutes les méthodes ont convergé vers le même point, qui est la racine, confirmant leur bon fonctionnement. On note que la fonction de Newton et du point fixe 2 ont été les plus rapides à converger, tant en nombre d'itération qu'en temps d'exécution. La dichotomie, trichotomie, point fixe 1 et sécante ont été plus longues et lentes à converger, mais leurs résultats restent assez proches des 2 premières méthodes. La fausse position a pris beaucoup de temps et d'itération à converger : 10 fois plus que Newton en termes de temps et d'itérations.

On observe également que la plupart des méthodes ont un graphe d'erreur assez régulier, que ce soit une courbe ou une droite, à l'exception de la dichotomie et de la trichotomie qui ont un graphe en ligne brisée.

Enfin, on observe que les erreurs finales des méthodes ne sont pas les mêmes dans toutes les méthodes, et que la sécante est particulièrement basse, à 10^{-6}.

Discussion

Que penser de ces résultats ? Sont-ils fiables ? Que peut-on en déduire ?

Tout d'abord, lorsque l'on regarde les résultats, on s'aperçoit que notre erreur est largement inférieure à la tolérance. Cela s'explique par le fait que notre erreur est calculée en comparant la valeur exacte de la racine avec la valeur calculée, alors que les conditions de sortie des algorithmes sont basées sur la l'image de cette approximation.

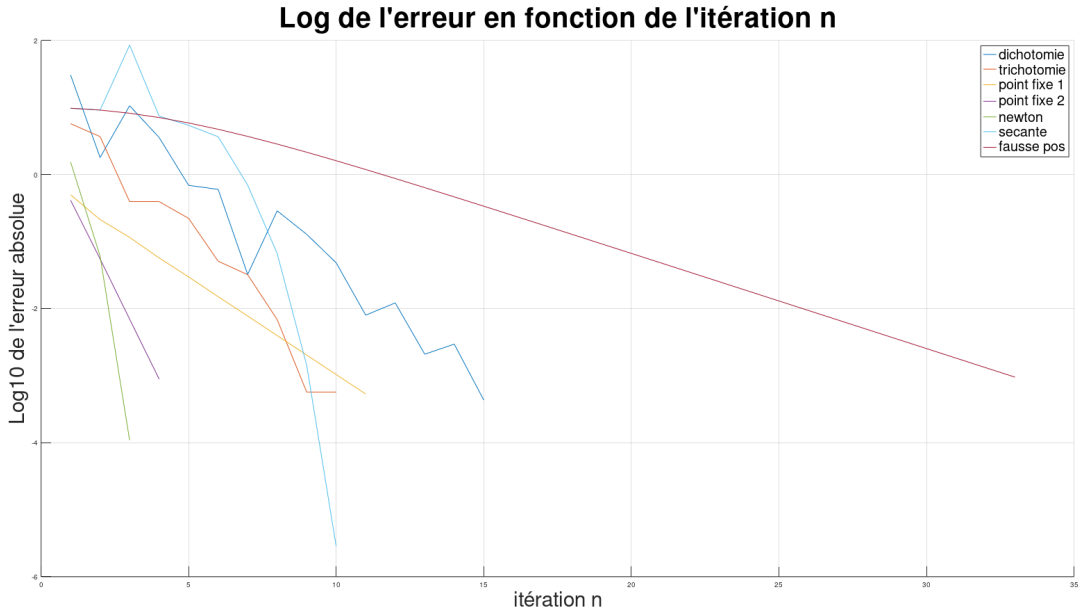


image originale

En calculant l'erreur sur l'image, on rectifie cela et on s'arrête (à une itération près) à 10^{-3}

On remarque également que certaines fonctions ont une erreur inférieure aux autres (Newton notamment). Cela ne veut cependant pas dire qu'une méthode n'est pas plus précise qu'une autre; cela dépend de la fonction que l'on veut étudier, mais aussi du ou des points d'initialisation, de la condition de sortie…

Traitons maintenant les méthodes au cas par cas :

Les méthodes de dichotomie et trichotomie sont peu efficaces, elles nécessitent du temps pour converger. Néanmoins, ces méthodes sont simples et faciles à mettre en place et à imager. De plus, elles ne nécessitent pas de connaître l'expression de la fonction et pourraient être utilisées dans un cas où les valeurs sont obtenues par des capteurs ou par une simulation. Leur dernier avantage est le fait que pour des points d'initialisation de sens opposés, la convergence vers la racine est assurée.

Ces deux méthodes sont également très irrégulières dans leur façon de converger. Cela est dû au fait que l'erreur représentée soit l'erreur effective à l'itération n : $|c_n - c|$
En affichant l'erreur maximale à l'itération n $\frac{b-a}{2^n}$, on trouve une évolution linéaire qui majore bien l'erreur effective à chaque itération.

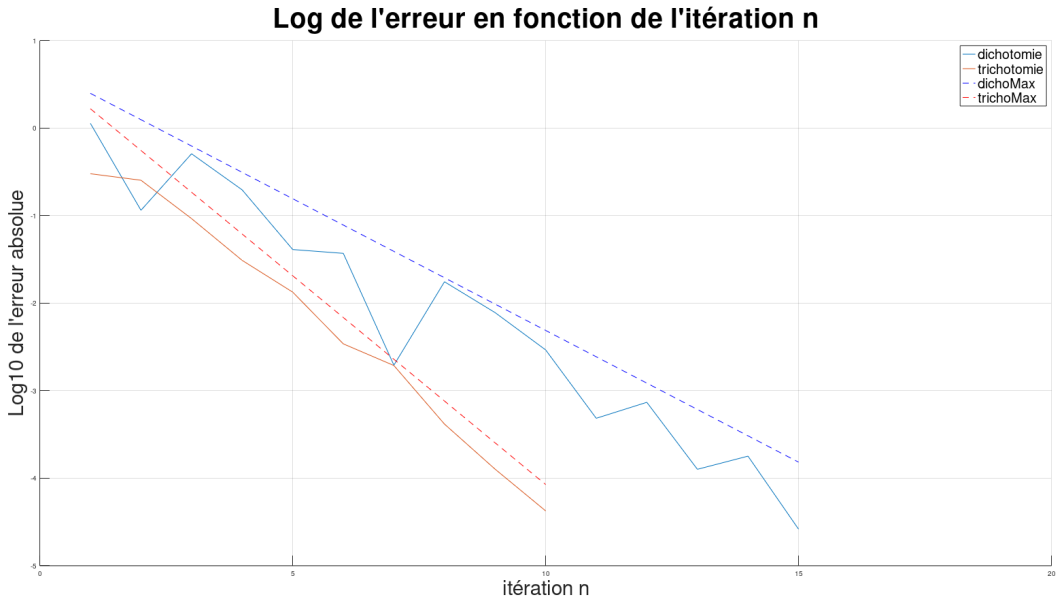


image originale

La méthode du point fixe peut être efficace mais nécessite une recherche poussée pour trouver une bonne fonction g(x). De plus, cela nécessite un point d'initialisation proche de la racine ce qui peut être contraignant. Dans le cas contraire, la suite risque de diverger au lieu de converger vers la racine.

La méthode qui converge le plus vite est la méthode de Newton. Cette méthode est plutôt fiable semble fonctionner avec des valeurs initiales éloignées de la racine, elle peut cependant diverger.
Néanmoins, cette méthode n'est pas parfaite, en effet, la méthode de Newton nécessite de connaître la fonction dérivée de la fonction dont on souhaite déterminer les racines, et cela n'est, hélas, pas toujours le possible.

L'avantage de la méthode de la sécante est qu'elle ne nécessite pas de connaître la fonction dérivée de la fonction que l'on étudie car elle se base sur le calcul du taux d'accroissement, qui est une approximation en un point de la fonction dérivée. De plus, comme dichotomie et trichotomie, cette méthode peut s'exécuter sans l'expression littéral de la fonction, ce qui peut être très utile lors de l'utilisation de capteurs. Elle peut cependant être mise en échec en cas de point d'inflection

La méthode de la fausse position est une méthode fiable. En effet, elle convergera toujours. Elle est cependant plus lente que la méthode de Newton ou de la sécante. Cette méthode utilise le même principe de fonctionnement que la méthode de la sécante, c'est pourquoi cette méthode peut aussi être utilisée sans l'expression littérale de la fonction.

Conclusion

En conclusion, malgré l'existence de beaucoup de méthodes itératives pour trouver des zéros, certaines se démarquent plus que d'autres selon le besoin et les possibilités. En effet, le fait de connaître l'expression de la fonction, une approximation de la racine ou encore le besoin de généralisation peuvent influer sur le choix de la méthode. Si vous avez besoin d'un algorithme simple à mettre en place, il faudra opter pour la méthode de dichotomie ou de trichotomie. Si il y a besoin d'une méthode particulièrement stable mais que la vitesse de convergence n'est pas un problème, la fausse position sera de mise. Si le fait de devoir retravailler la fonction avant de lancer la recherche de zéro n'est pas problématique, le point fixe fera l'affaire, bien que la méthode de Newton soit au moins aussi rapide. Mais si vous ne connaissez pas la dérivée de votre fonction voire la fonction elle même, la méthode de la sécante sera utilisable. Chaque problème est différent, chacun pourrait être favorisé par telle ou telle méthode.

Références

Images

- logo esir : esir.univ-rennes1.fr/
- logo rennes1 : fr.wikipedia.org/wiki/Universit%C3%A9_Rennes-I
- logo matlab : fr.wikipedia.org/wiki/MATLAB
- Illustration dichotomie : cpge.frama.io
- Illustration du point fixe : cours-info.iut-bm.univ-fcomte.fr
- Illustration de la méthode de Newton : fr.wikipedia.org/wiki/Fichier:Methode_Newton.svg
- Illustration méthode de la sécante : fr.wikipedia.org
- Illustration fausse position : acervolima.com.fr

Références

1. https://www.legifrance.gouv.fr/codes/article_lc/LEGIARTI000006278911/
2. fr.wikipedia.org/wiki/Algorithmique
3. infodocbib.net/2020/09/algorithmique-et-algorithmes/#cuisine
4. https://fr.wikipedia.org/wiki/MATLAB
5. https://fr.wikipedia.org/wiki/GNU_Octave
6. https://fr-academic.com/dic.nsf/frwiki/1633985
7. https://www.bibmath.net/dico/index.php?action=affiche&quoi=./l/lagrangemeth.html
8. https://fr.wikipedia.org/wiki/M%C3%A9thode_de_la_fausse_position