Projet maths-info
Résolution d'équations non-linéaires

Benamara Abdelkader
Benichou Yaniv

08 Juin 2020

Contents

1 Introduction

Le but de ce projet est de décrire les algorithmes les plus fréquemment utilisés pour résoudre des équations non linéaires du type :
$f(x) = 0$

2 Méthode De Dichotomie

2.1 Théoreme des valeurs intermédiaires :

Pour toute application continue $f : [a, b] \to \mathbb{R}$ et tout reel u compris entre f(a) et f(b), il existe au moins un reel c compris entre a et b tel que f(c) = u.
En particulier on a la version de Bolzano qui nous interesse précisement.

2.2 Théoreme de Bolzano

Soit une fonction continue $f : [a, b] \to \mathbb{R}$
si $f(a)f(b) < 0$ alors $\exists \alpha \in]a, b[$ tel que f(α)=0.

On considère deux nombres réels a et b et une fonction réelle f continue sur l'intervalle [a, b] telle que f(a) et f(b) soient de signes opposés. Supposons que nous voulions résoudre l'équation f(x) = 0. D'après le théorème des valeurs intermédiaires, f a au moins un zéro dans l'intervalle [a, b]. La méthode de dichotomie consiste à diviser l'intervalle en deux en calculant m = (a+b) / 2. Il y a maintenant deux possibilités : ou f(a) et f(m) sont de signes contraires, ou f(m) et f(b) sont de signes contraires.

L'algorithme de dichotomie est alors appliqué au sous-intervalle dans lequel le changement de signe se produit, ce qui signifie que l'algorithme de dichotomie est récursif.

2.3 L'algorithme de dichotomie

Algorithm 1: Méthode de dichotomie

Result: millieu tq f(millieu)=0
initialization;
while *fin-debut ¿ err* **do**
    millieu = (debut + fin) / 2;
    if *f(millieu) ¿ 0* **then**
        fin = millieu;
    else
        debut=millieu;
    end
end

```python
class Dichotomie(Equa_Solver):

    def solve(self):
        # Donnees en parametres
        a , b = self.a , self.b
        err=self.err
        try:
            f = lambda x: eval(str(self.f))
        except (TypeError,SyntaxError):
            return
        x_list=[]

        print(f" Fonction    : {self.f}")
        print(f" Intervalle : [{a},{b}]")
        print(f" Erreur      : {err} \n")


        if( f(a)*f(b) > 0):
            raise SolverException(" f(a) et f(b) doivent etre de signe different !")

        else:
            debut = a
            fin = b
            n=1

            while (fin - debut > err):
                millieu = (debut + fin) / 2
                x_list.append(millieu)
                print(f"Found solution after {n} iterations : {millieu} ")
                n+=1
                if (f(debut) * f(millieu) < 0):
                    fin = millieu
                else:
                    debut = millieu

            print(f" Solution approchee de f(x) = 0 est : {millieu}\n")
            return x_list
```
Listing 1: Méthode de dichotomie en Python

3 Méthode de Newton

3.1 Présentation

Afin de mettre au point des algorithmes possédant de meilleures propriétés de convergence que la méthode de dichotomie, il est nécessaire de prendre en compte les informations données par les valeurs de f et, éventuellement, par sa dérivée f' (si f est différentiable) ou par une approximation convenable de celle-ci.
L'algorithme de la méthode de Newton peut être présenté brièvement comme suit: à chaque itération, la fonction dont on cherche un zéro est linéarisée en l'itéré (ou point) courant et l'itéré suivant est pris égal au zéro de la fonction linéarisée. Cette description sommaire indique qu'au moins deux conditions sont requises pour la bonne marche de l'algorithme : la fonction doit être différentiable aux points visités (pour pouvoir y linéariser la fonction) et les dérivées ne doivent pas s'y annuler (pour que la fonction linéarisée ait un zéro) ; s'ajoute à ces conditions la contrainte forte de devoir prendre le premier itéré assez proche d'un zéro régulier de la fonction (i.e., en lequel la dérivée de la fonction ne s'annule pas), pour que la convergence du processus soit assurée.

Ecrivons pour cela le développement de Taylor de f en α au premier ordre. On obtient alors la version linéarisée du problème $f(\alpha) = 0 = f(x) + (\alpha x)f(\eta)$, où η est entre α et x. L'équation conduit à la méthode itérative suivante : $\forall k \geq 0$, étant donné x^k , déterminer $x^(k+1)$ en résolvant l'équation $f(x^k) + (x^{k+1} - x^k)q_k = 0$, où q_k est une approximation de $f(x^k)$.

Considérons maintenant quatre choix particuliers de q_k .
Ici on pose : $\quad q_k = f'(x_k) \quad \forall k \geq 0$
et en se donnant la valeur initiale x^0 , on obtient la méthode de Newton :
$x^{k+1} = x^k - \frac{f(x^k)}{f'(x^k)} \quad \forall k \geq 0$

A la k-ème itération, la méthode de Newton nécessite l'évaluation des deux fonctions f et f' au point x^k. Cet effort de calcul supplémentaire est plus que compensé par une accélération de la convergence, la méthode de Newton étant d'ordre 2.

Or, il est possible que la dérivée de la fonction f soit relativement pénible à calculer et c'est pour ça que nous avons présenté une deuxieme version d'implémentation de cette méthode,sans dérivée donnée en paramètre,puisqu'elle est automatiquement calculée. Cela rend la méthode de Newton très agréable à utiliser,puisqu'elle converge très rapidement et ne nécessite donc, uniquement un point approximatif x_0 comme argument supplementaire à la fonction.

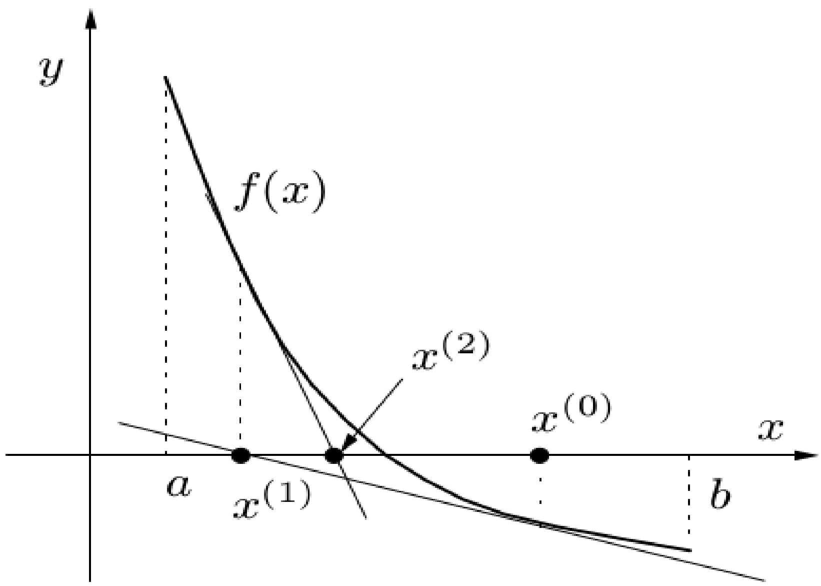
3.2 Interpretation Géometrique

Autrement dit, on veut approcher la fonction au premier ordre, c'est à dire, on la considère asymptotiquement égale à sa tangente en ce point :
$f(x) \simeq f(x_0) + f'(x_0)(x - x_0)$.

Partant de là, pour trouver un zéro de cette fonction d'approximation, on calcule l'intersection de la droite tangente avec l'axe des abscisses, c'est-à-dire résoudre l'équation affine :

$0 = f(x_0) + f'(x_0)(x - x_0)$.

On obtient alors un point x^1 qui en général a de bonnes chances d'être plus proche du vrai zéro de f que le point x^0 précédent. Par cette opération, on peut donc espérer améliorer l'approximation par itérations successives (voir illustration) : on approche à nouveau la fonction par sa tangente en $x_1 pourobtenirunnouveaupoint$x^2, etc.



Les deux premieres étapes de la méthode de Newton.

3.3 Implementation du code

```

class Newton(Equa_Solver):

    def solve_with_df(self):
        f=self.f
        Df=self.df
        max_iter=self.max_iter
        x0=self.x0
        epsilon=self.err
        x_list=[]

        fx = lambda x: eval(str(f))
        dfx = lambda x: eval(str(Df))
        print("\n\nfunction f : ", f, "\n", "Derivative f' : ", Df,
 "\n", "--------------------------------")
        xn = x0
        for n in range(0, max_iter):
```

```
            fxn = fx(xn)
            x_list.append(xn)
            self.affiche_info(n,xn,fxn)
            if abs(fxn) < epsilon:
                print('Found solution after', n, 'iterations.')
                print("the approximate solution is : ",x_list[-1])
                return x_list

            Dfxn = dfx(xn)

            if Dfxn == 0:
                print('Zero derivative. No solution found.')
                return None
            xn = xn - fxn / Dfxn

        print('Exceeded maximum iterations. No solution found.')
        return None

    def solve_without_df(self):
        f=self.f
        max_iter=self.max_iter
        x0=self.x0
        epsilon=self.err
        x_list=[]


        x = Symbol('x')
        fx = lambda x: eval(str(f))
        dfx = lambda x: eval(str(diff(f)))
        print("\n\nfunction f : ", f, "\n", "Derivative f' : ",
    diff(f), "\n", "--------------------------------")

        xn = x0

        for n in range(0, max_iter):
            fxn = fx(xn)
            x_list.append(xn)
            self.affiche_info(n, xn, fxn)
            if abs(fxn) < epsilon:
                print('Found solution after', n, 'iterations.')
                print("the approximate solution is : ",x_list[-1])
                return x_list

            Dfxn = dfx(xn)
            if Dfxn == 0:
                print('Zero derivative. No solution found.')
                return None
            xn = xn - fxn / Dfxn
        print('Exceeded maximum iterations. No solution found.')
        return None
```
Listing 2: Méthode de Newton en Python

3.4 Convergence

A ce jour, la méthode de newton est la méthode qui converge le plus rapidement,s'il y a convergence, parmis celles présentées aujourdh'ui. En effet, celle-ci est rapide (souvent quadratique), et de plus elle nécessite uniquement un seul point de départ (généralement grossièrement proche de la solution). Mais, malheureusement f doit être suffisamment régulière, la convergence n'est pas assurée dans tous les cas, s'il y a plusieurs racines elle ne converge par forcément vers la plus proche du point de départ.

4 Méthode des Cordes

4.1 Présentation

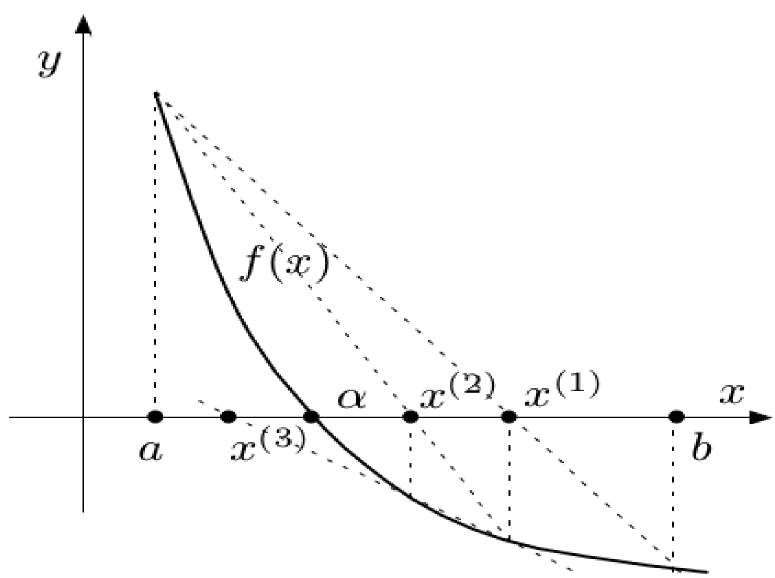
La méthode des cordes est une méthode comparable à celle de Newton, où l'on remplace $f'(x_n)$, *par* $\dfrac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}$

L'initialisation nécessite deux points x_0 et x_1, proches, si possible, de la solution recherchée. Il n'est pas nécessaire que x_0 et x_1 encadrent une racine de f. La méthode des cordes peut aussi être vue comme une généralisation de la méthode de la fausse position, où les calculs sont itérés.
Ici on pose donc : $q_k = \frac{f(x^k)-f(x^{k-1})}{x^k-x^{k-1}}$ $\forall k \geq 0$ d'où on déduit, en se donnant deux valeurs initiales x^1 et x^0,la relation suivante :

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

4.2 Interpretation Géometrique

De manière très intuitive, cette méthode consiste à tracer une droite entre les f(a) et f(b), qui passera forcement par l'axe des abscisses en un point,x^1 qui sera la prochaine itération. On réitère le procédé jusqu'à approximation de la solution.



Sur ce graphe,on peut voir les deux premières étapes de la méthode pour la résolution d'une fonction f.

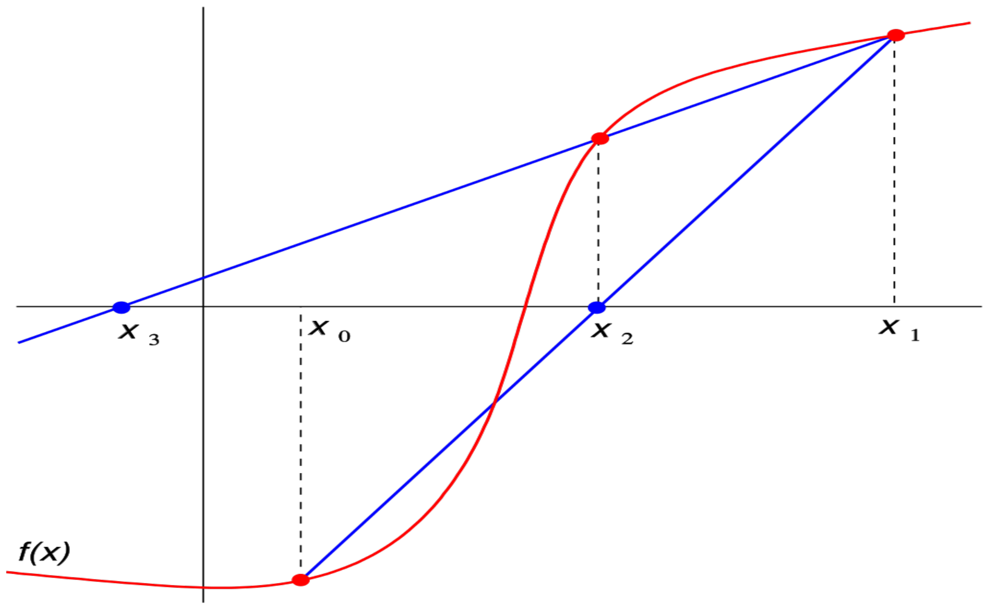


Illustration des deux premières itérations, pour une autre courbe (ici, la méthode va diverger car x^0 et x^1 sont choisis trop loin de la solution).

4.3 Implementation du code

```
class Cordes(Equa_Solver):

def solve(self):
    f=self.f
    a,b =self.a,self.b
    max_iter=self.max_iter
    epsilon=self.err
    x_list=[]

    fx = lambda x: eval(str(f))
    print("\n\nfunction f : ", f, " dans l'intervalle [", a, ", ", b, "] \n", "--------------------------------")


    for n in range(0, max_iter):
        self.affiche_info(n, b, fx(b))
        x_list.append(b)
        if (abs(a - b) < epsilon):
            print('Found solution after', n, 'iterations.')
            return x_list

        z = (a * fx(b) - b * fx(a)) / (fx(b) - fx(a))
        a, b = b, z

    print('Exceeded maximum iterations =', max_iter, '.No solution found.')
```

```python
        return None
```

Listing 3: Méthode des cordes en Python

4.4 Convergence

Si les valeurs initiales x^0 et x^1 sont suffisamment proches de la solution, la méthode aura un ordre de convergence de $\varphi = \dfrac{1+\sqrt{5}}{2} \simeq 1,618$ qui est le nombre d'or.

Sinon, nous avons vu que la méthode peu diverger.

5 Méthode de la Fausse-Position

5.1 Présentation

C'est une variante de la méthode des cordes dans laquelle, au lieu de prendre la droite passant par les points $(x^k, f(x^k))$ et $(x^{k-1}, f(x^{k-1}))$, on prend celle passant par $(x^k, f(x^k))$ et $(x^{k'}, f(x^{k'})))$.

Plus précisemment, une fois trouvées deux valeurs x^1 et x^0 telles que $f(x^1)\Delta f(x^0) <$ 0, on pose

$x^{k+1} = x^k$ - $\frac{x^k - x^{k'}}{f(x^k) - f(x^{k'}}\forall k \geq 0$

Ainsi, les itérations construites sont toutes contenus dans l'intervalle de départ, $[x^{-1}$,$x^0]$, à la différence de la méthode des cordes.

5.2 Interpretation Géometrique

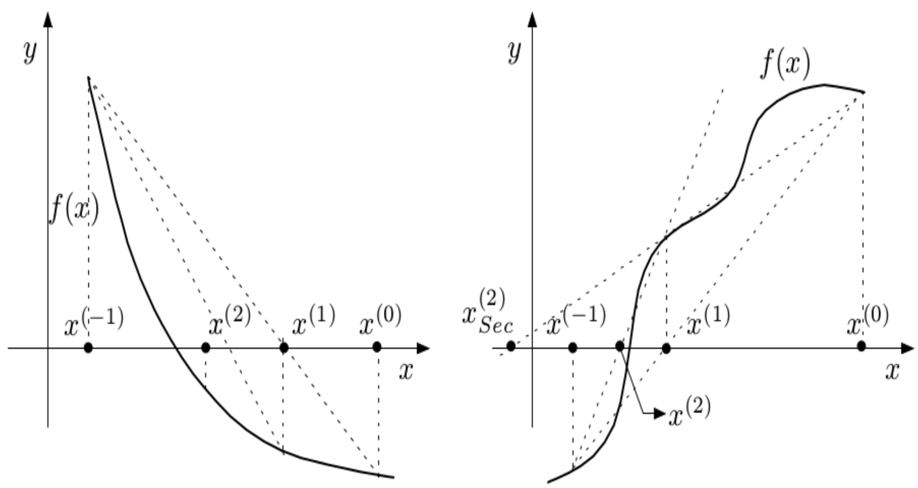


Fig. 6.3. Les deux premières étapes de la méthode de la fausse position pour deux fonctions différentes

5.3 Implementation du code

```
class FalsePosition(Equa_Solver):

    def solve(self):
        f=self.f
        a,b=self.a,self.b
        tol=self.err
        x_list=[]
        fx = lambda x: eval(str(f))
```

```
        print("\n\nfunction f : ", f, " dans l'intervalle [", a, ",
    ", b, "] \n", "--------------------------------")

        if fx(a) * fx(b) > 0:
            raise SolverException(" f(a) et f(b) doivent etre de
    signe different !")

        n = 0
        while abs(b - a) > 2 * tol:
            c = (a * fx(b) - b * fx(a)) / (fx(b) - fx(a))
            self.affiche_info(n, c, fx(c))
            n += 1

            x_list.append(c)

            if fx(c - tol) * fx(c + tol) <= 0:
                print('Found solution after', n, 'iterations.')
                return x_list
            if fx(a) * fx(c) > 0:
                a = c
            else:
                b = c

        print('Found solution after', n, 'iterations.')
        x_list.append((a+b)/2)
        return x_list
```
Listing 4: Méthode de la Fausse Position en Python

5.4 Convergence

La méthode de la fausse position, bien qu'ayant la même complexité que la méthode des cordes,a une convergence linéaire. La méthode de la fausse position peut être vue comme une méthode globalement convergente, tout comme celle de dichotomie.

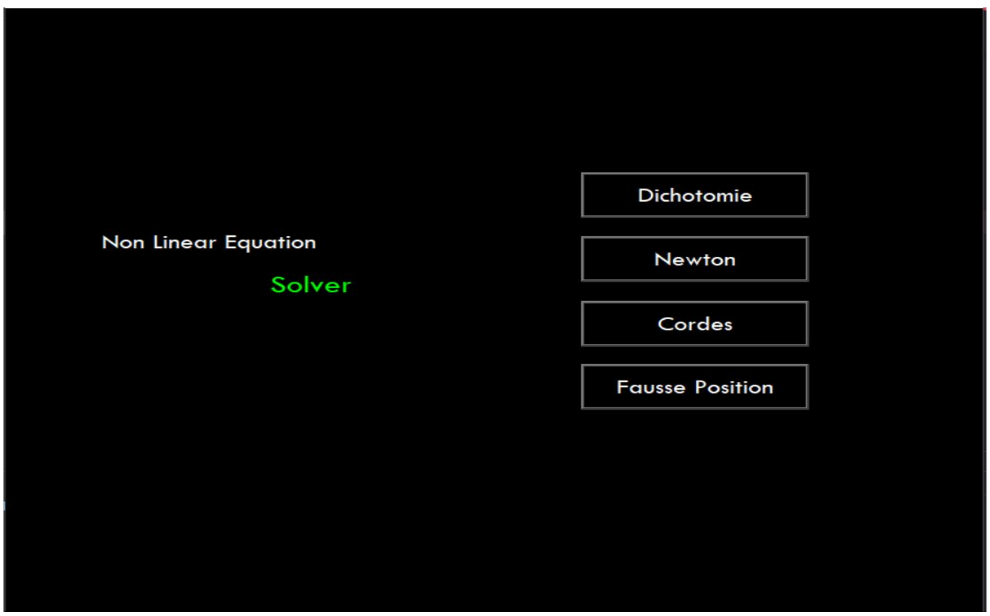
6 Interface Graphique

6.1 Outils utilisés

Afin de produire une version plus pratique à l'utilisateur plutot que faire un affichage sur console délicat de suivre les résultats avec on a préviligé de réaliser une interface graphique plutot simple mais efficace.
La réalisation de cette interface a été effectué à l'aide de la bibliotheque Tkinter des modules déjà existants sur Python et aussi la réalisation des graphes pour pouvoir afficher les courbes et les différents résultats était fait à l'aide de la bibliotheque Matplotlib qui du fait doit etre installé pour le bon fonctionnement du projet.

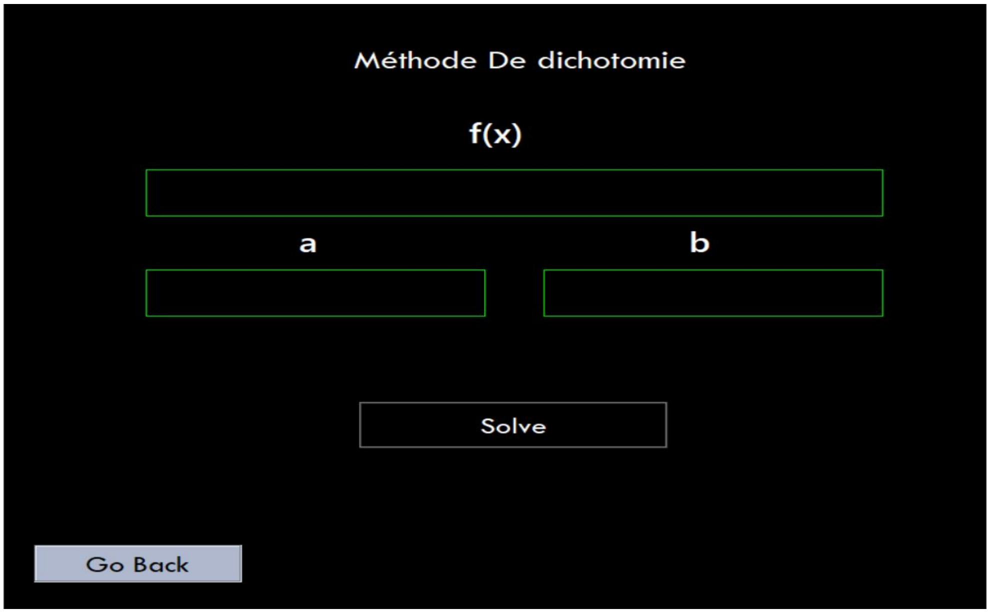
Notre application est consitué de 5 principales pages :

6.2 Accueil



L'interface d'accueil de notre application est constitué des quatres méthodes déjà citées afin de résoudre
des equations f(x)=0 et pour accéder à chaque méthode il suffit de suivre le boutton correspandant
à la méthode
Alors dans ce qui suit on va détailler chaque page dans notre interface graphique

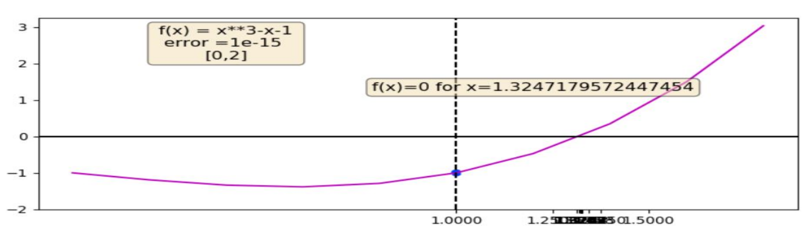
6.3 Dichotomie



Sur cette page on s'occupe de la méthode de dichotomie donc on prend en parametres :
-une fonction f
-un interval [a,b]

Alors si l'utilisateur de notre application rentre bien les infomations attendus et puis il clique sur le boutton solve il aura un affichage :

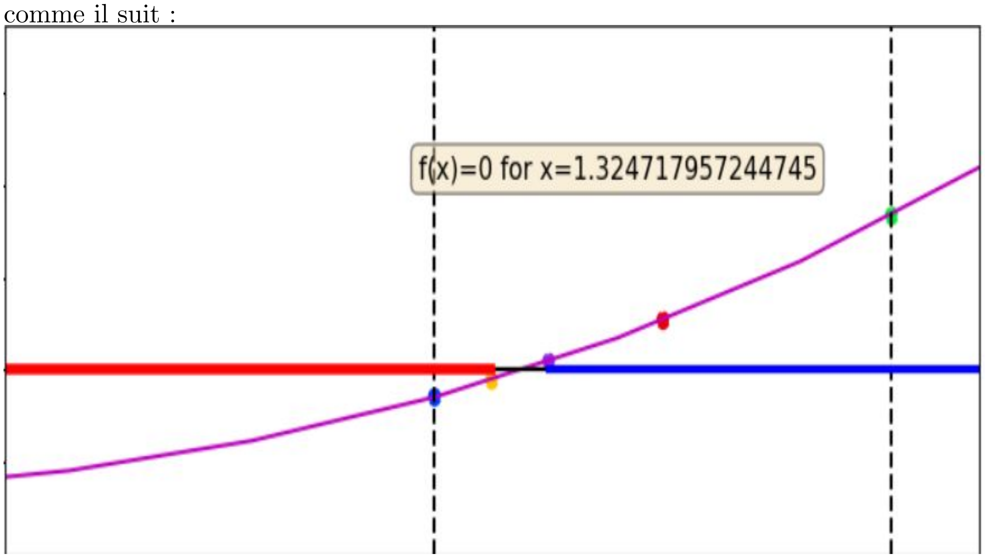
xn	f(xn)
1.3247179572454115	2.838174140151 75e-12
1.3247179572449568	8.988365607365267e-13
1.3247179572447294	-7.105427357601002e-14
1.324717957244843	4.1389114358025836e-13
1.3247179572447862	1.7141843500212417e-13
1.3247179572447578	5.0182080713057076e-14
1.3247179572447436	-1.021405182655144e-14
1.3247179572447507	1.9984014443252818e-14
1.3247179572447472	4.884981308350689e-15
1.3247179572447454	-2.6645352591003757e-15



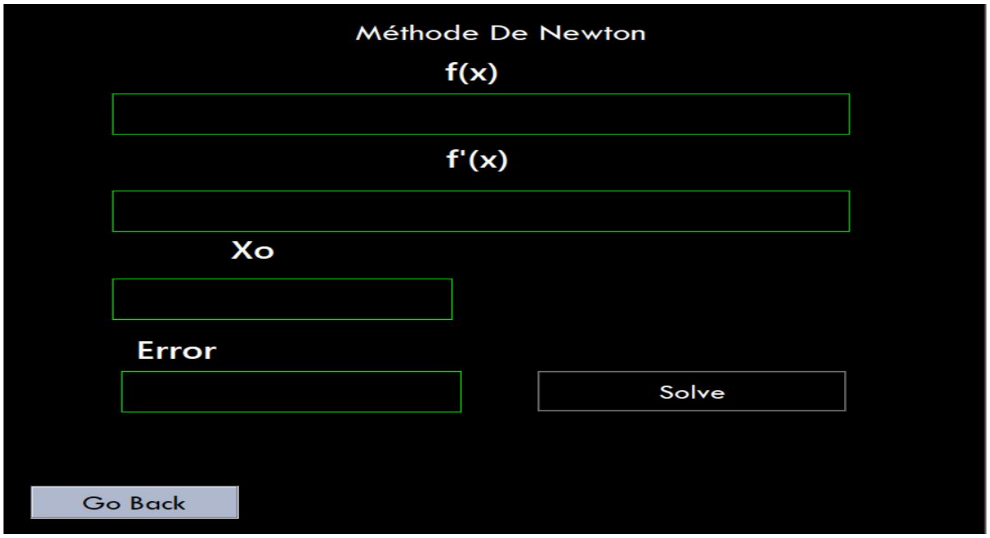
Dans la fenetre on remarque bien les informations relatives à la fonction rentrée mais aussi les résultats retournés apres éxecution de la méthode de Dichotomie
NB : le tableau affiche les résultats des 10 dernieres itérations de cette méthode

6.3.1 Convergence

Pour la méthode de dichotomie la convergence vers la solution est graphiqueement représentés par des lignes horizontales qui se rapprochent du point cherché comme il suit :

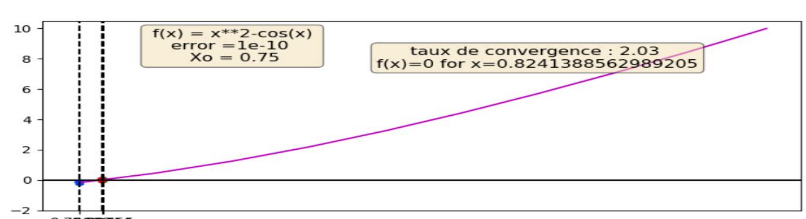


6.4 Newton



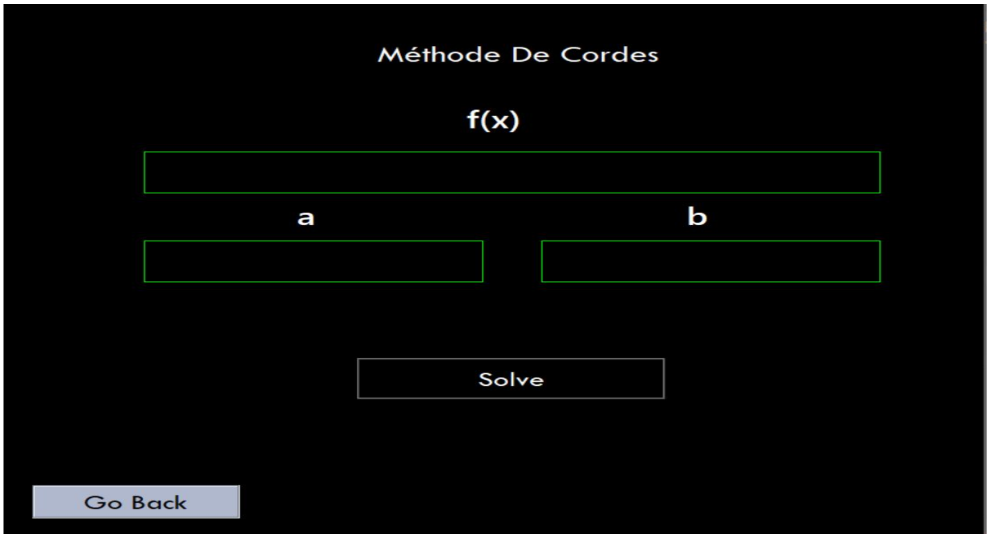
Dans cette page on résout les equations f(x)=0 à l'aide de la méthode de Newton donc on prends en arguments une fonction f et optionnellement sa dérivée et aussi un point de départ de notre récurrence x_0 et un champ error qui indique avec quelle précision les résultats devront etre pris en compte cette valeur est initialisée à 10^{-15} de base.
Apres rensignement des champs et cliquer sur le boutton solve une fentetre va s'afficher :

xn	f(xn)
0.75	-0.1691888688738209
0.8275512756621592	0.008160388437742694
0.8241388562989205	1.558931842038369e-05

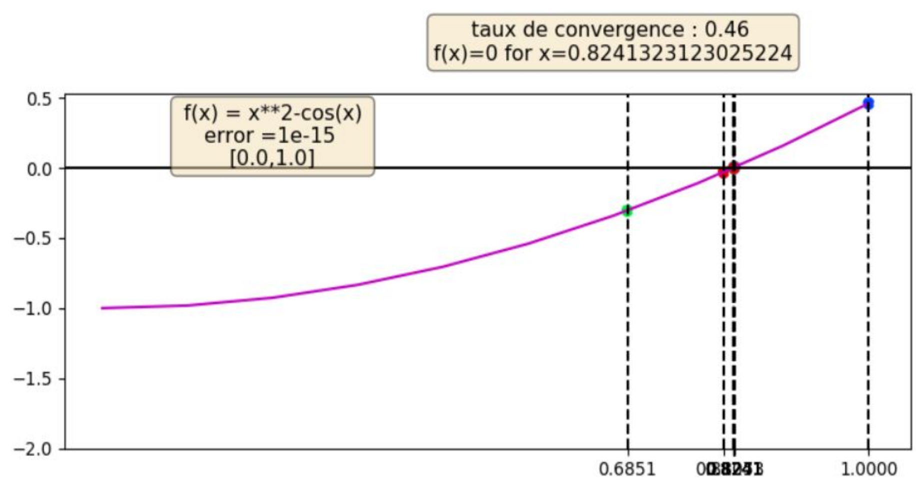


avec toutes les information comme déja expliquer dans le paragraphe précédent mais cette fois çi le taux de convergence est explicitement calculé et donné avec les résultats

6.5 Cordes

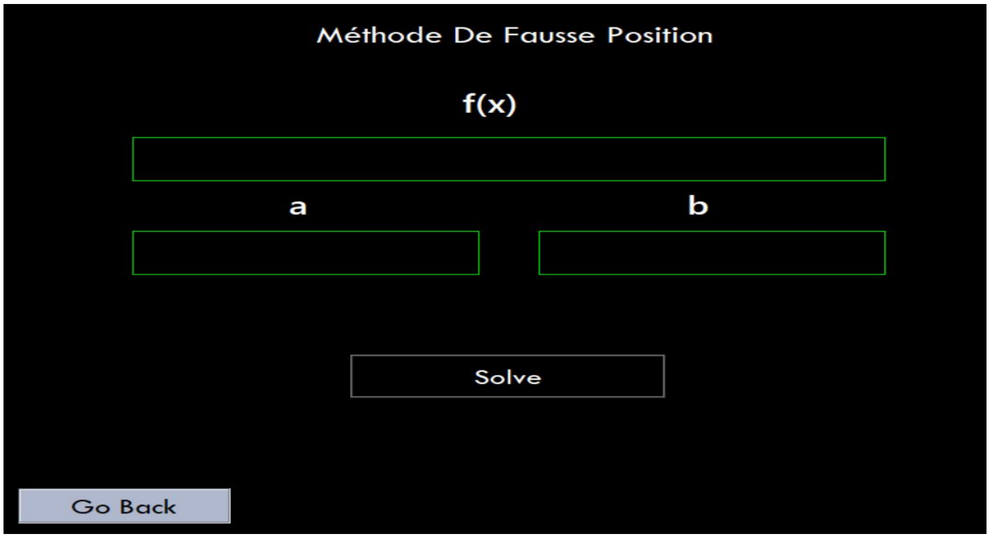


Cette page s'occupe comme son nom l'indique de résolution des equation f(x)=0 à l'aide de la méthode des cordes elle prend en arguments une fonction f et un interval [a,b] et un clique sur le boutton solve nous donne cette fenetre :

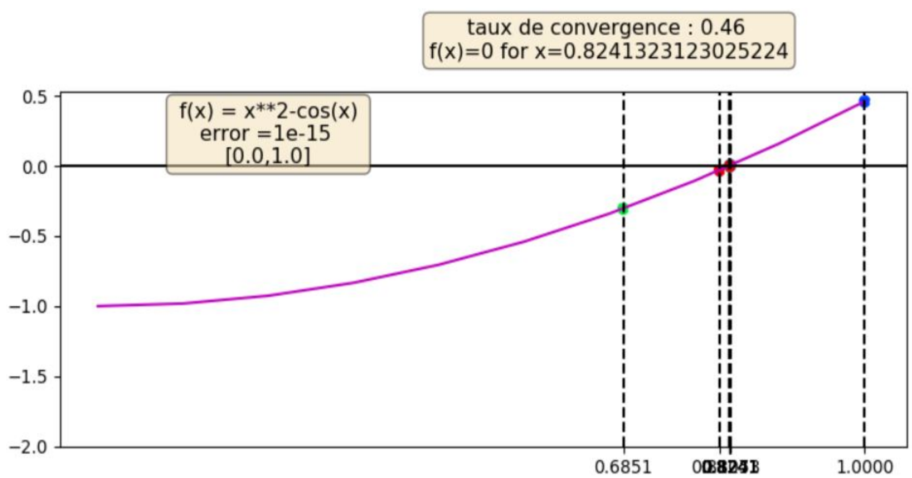


Comme bien illustré sur le graphique on nous permet de suivre la convergence de la solution volu tout en affichant le taux de convergence correspandant avec les informations nécaissaires de la fonction et l'interval.

6.6 Fausse Position



Cette page s'occupe comme son nom l'indique de résolution des equation f(x)=0 à l'aide de la méthode de fausse position elle prend en arguments une fonction f et un interval [a,b] et un clique sur le boutton solve nous donne cette fenetre :

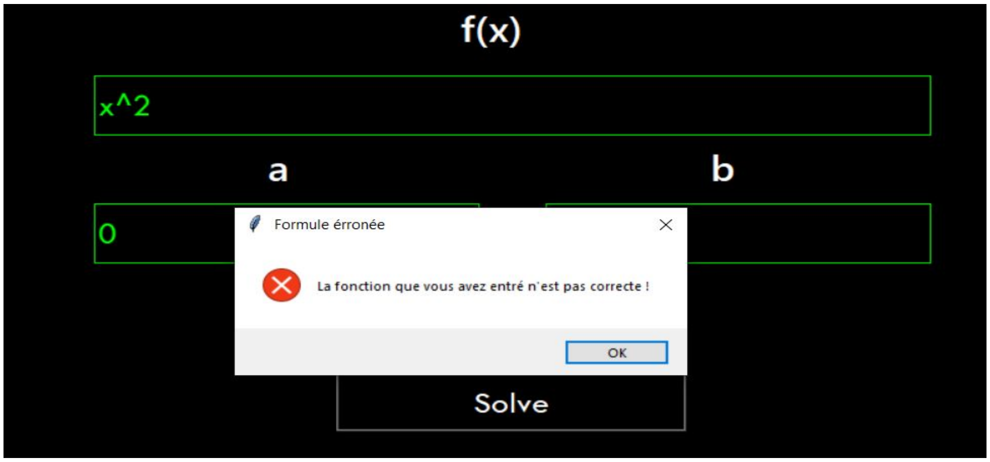


Comme bien illustré sur le graphique on nous permet de suivre la convergence de la solution volu tout en affichant le taux de convergence correspandant avec les informations nécaissaires de la fonction et l'interval.

6.7 Gestion des erreurs

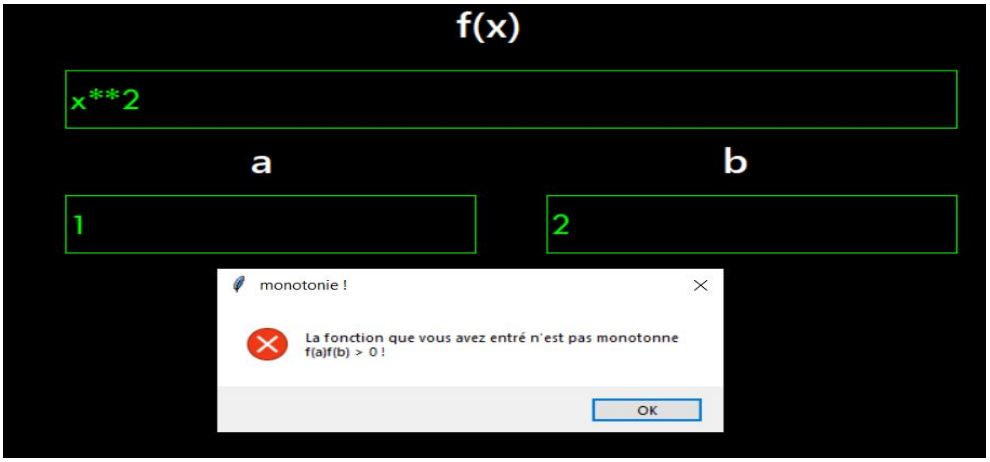
Comme déja cité dans les différentes fentetres de notre application une vérification de données tapées est nécaissaire pour le bon fonctionnement du projet alors les vérifications faites sont :

6.7.1 Formule Erronée



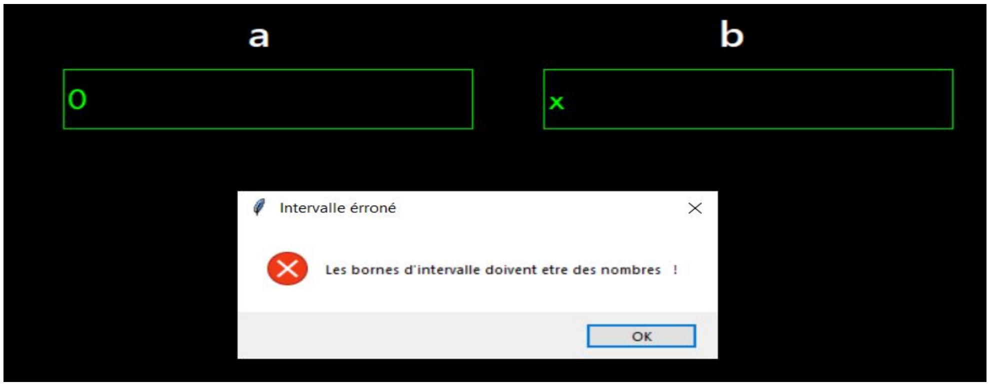
Une formule entrée en champ dédiée est dite erronée si elle est mal exprimée au sens de Python donc toutes les formules sont supposées écrites en syntaxe de Python

6.7.2 Monotonie



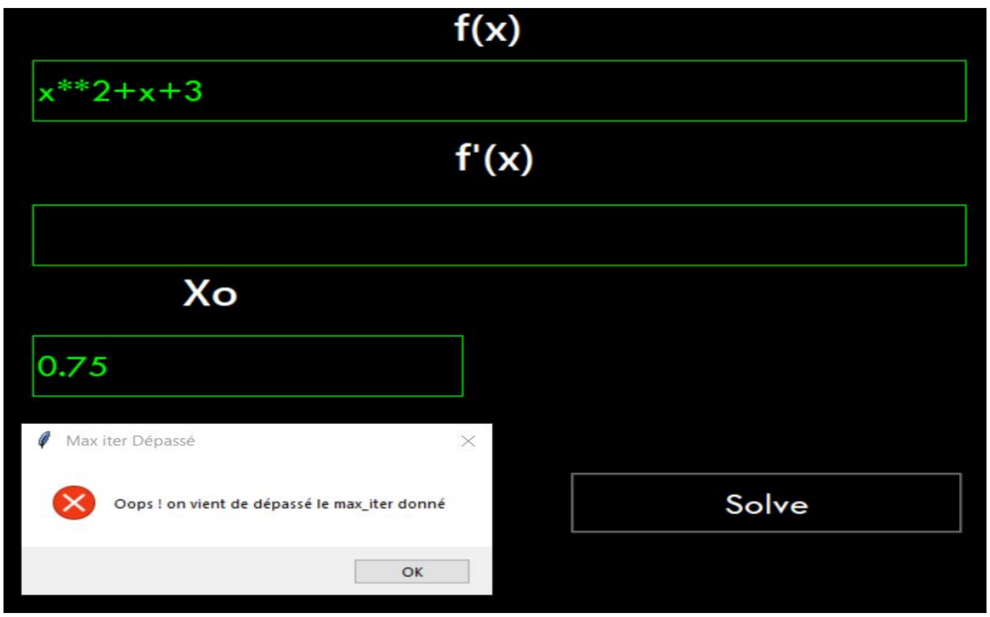
Si on rentre une fonction qui ne vérifie pas la condition des hypotheses de théoreme de valeurs intermédiaires une érreur s'affichera indiquant ce fait .

6.7.3 Interval Erroné



Si on rentre un interval qui ne correspand un de ces bornes à des nombres (entiers ou floats) une érreur de type interval erroné s'affichera.

6.7.4 Max d'itération atteint sans trouver de solution



Si notre fonction n'admet pas de valeurs qui le rendent nulle dans l'interval donné et donc on atteint le maximum d'itération sans trouver de solution pour f(x)=0 alors on afficher la fentetre d'erreur ci dessous

7 Applications

8 Conclusion