Analyse par intervalle pour l'étude de fonction

ISTIA - Université Angers - Année 2017

COUPEAU Patty
JUVIN Carla

Table des matières

1. Introduction

L'objectif de ce projet était de réaliser un programme python capable de donner les changements de variation d'une fonction de R à valeurs dans R.

Entrée du programme : une fonction quelconque de f: R->R

Sortie du programme: les box indiquant les valeurs pour lesquelles il y a changement de variation et les images de ces valeurs par la fonction f.

Définition 1.1 : Les intervalles sont des sous-ensembles fermés connexes de IR

Définition 1.2 : Soient I_1 et I_2 deux intervalles. Le résultat d'une opération entre deux intervalles : $I_1 \star I_2$ est le plus petit intervalle contenant $\{x \star y \mid x \in I_1, y \in I_2\}$, c'est-à-dire le plus petit intervalle contenant tous les résultats possibles de l'opération $\star$ appliquée à tous les éléments x de I_1 et tous les éléments y de I_2.

On obtient les formules suivantes :

- [a, b] +[c, d] = [a + c, b + d]
- [a, b] - [c, d] = [a - d, b - c]
- [a, b] ×[c, d] = [min(a × c, a × d, c × b, b × d), max(a × c, a × d, c × b, b × d)]
- 1 /[a, b] = [min(1 /a, 1 /b), max(1 /a, 1 /b)] si 0 ∉ [a, b]
- 1 /[a, b] = [-∞;+∞] si 0 ∈[a, b]
- [a, b] /[c, d] = [a, b] × (1 /[c, d]) si 0 ∉ [c, d]

Définition 1.3 : Une boîte est un vecteur d'intervalles.

On note IE l'ensemble des boîtes dans E.

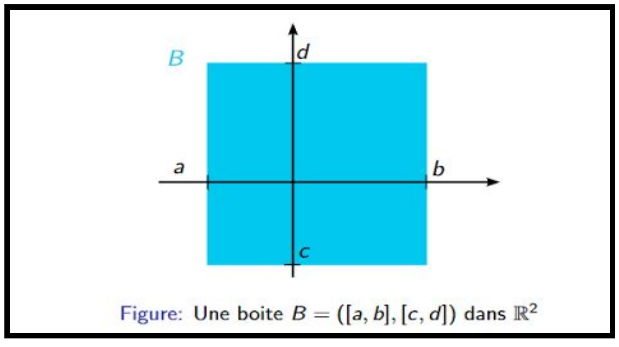


Figure: Une boite $B = ([a, b], [c, d])$ dans $\mathbb{R}^2$

Définition 1.4 : [f] : IE⟶IF est une fonction d'inclusion pour f : E⟶F si :

∀B∈IE , f(B) ⊆ [f](B)

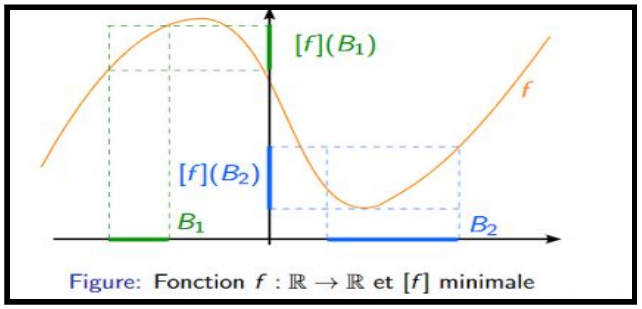


Figure: Fonction $f : \mathbb{R} \to \mathbb{R}$ et $[f]$ minimale

2. Algorithme d'inversion d'ensemble

L'objectif était de réaliser un programme capable de déterminer les intervalles dont l'image par une fonction quelconque f: R -> R est comprise dans un intervalle donné.

Entrée du programme : une fonction, le domaine d'étude de la fonction et un intervalle.

Sortie du programme : Une pile d'intervalles dont l'image par la fonction est comprise dans l'intervalle donné.

Algorithme 2.2 :

```
Fonction InversionEnsemble(f,D,I)

        pile = D
                Pour i dans pile
                        Si i est jaune Alors
                                [i1,i2] = Diviser l'intervalle i en deux
                                p = Color(f,[i1,i2],I)
                                InversionEnsemble(f,i1,I)
                                InversionEnsemble(f,i2,I)
                                Ajouter (p, newpile)
                        Sinon
                                Ajouter (i, newpile)
                        Fin sinon
                Fin Pour
        Retourner(newpile)
 Fin Fonction
```

Remarque 2.2: Cette fonction fait appel à une autre fonction qui permet de déterminer si l'image d'un intervalle i par la fonction f est comprise dans l'intervalle I donné. On a choisi le code couleur suivant, si l'image est comprise dans l'intervalle on dessine l'intervalle de départ en rouge. Si elle n'est pas dans cet intervalle, on le dessine en bleu. Sinon, si on ne peut pas conclure i.e. si l'intersection est non vide, on le colore en jaune.

Algorithme 2.3 :

```
Fonction Color(f,pile,I) //on utilise une pile d'intervalle

        Pour i dans pile Faire
                Si f(i)⋂I = ϕ alors
                        Dessiner l'intervalle i en bleu
                Fin Si
                Sinon Si f(i)∈I alors
                        Dessiner l'intervalle i en rouge
                Fin Si
                Sinon
                        Dessiner l'intervalle i en jaune
                Fin Sinon
        Fin Pour
       Retourner (pile)
 Fin Fonction
```

3. Algorithme de Newton par intervalles

Théorème 3.1 : Si $\exists$ x $\in$ [a, b] tel que f(x)=0 alors x $\in$ N_f ([a, b])

avec N_f ([a, b]) = (a+b) / 2 + f ((a+b)/2) / f ([a, b])

Remarque 3.2 : Afin de déterminer sur quels intervalles une fonction change de variation, ils faut connaître les zéros de sa dérivée. Nous avons conçu un algorithme permettant, à partir d'une fonction donnée et d'un intervalle [a, b] de construire l'intervalle $N_{f'}$ ([a, b]) basé sur le Théorème 3.1 appliqué à la dérivée de la fonction f.

Algorithme 3.3

```
Fonction Newton(f,i)
      Milieu = (i.inf + i.sup)/2  //le milieu de l'intervalle
      df= dérivée première de f
      ddf= dérivée seconde de f
      Retourner(Milieu + df(Milieu)/ddf(i))
Fin Fonction
```

Remarque 3.4 : Grâce au principe de Newton, nous avons conçu un algorithme permettant, à partir d'une fonction donnée et de son domaine d'étude, d'obtenir les intervalles sur lesquels sa dérivée s'annule .

Entrée du programme : une fonction et le domaine d'étude de la fonction.

Sortie du programme : Une pile d'intervalles contenant un zéro de la dérivée de f et donc un changement de variation de f.

Algorithme 3.5 :

```
Fonction ChangementVariation(f,D)
       pile = D
              Pour i dans pile
                     Si i est jaune Alors
                            [i1,i2] = Diviser l'intervalle i en deux
                            p = Color(Newton(f,i),[i1,i2],i)
                            Ajouter(ChangementVariation(f,i1),pileR)
                            Ajouter(ChangementVariation(f,i2),pileR)
                     Si i est rouge Alors
                            Ajouter (i, pileR)
                     Fin sinon
              Fin Pour
       Retourner(pileR)
 Fin Fonction
```

Remarque 3.6: Cette fonction fait appel à la fonction Newton (Algorithme 3.3) et à la fonction Color (Algorithme 2.3) .

4. Algorithme boites aux changements de variations

Remarque 4.1 : Afin d'étudier l'ensemble des changements de variation d'une fonction nous avons conçu un algorithme permettant à partir d'une fonction donnée et de son domaine d'étude, d'obtenir les intervalles sur lesquels sa dérivée s'annule. Le but était de réduire ces intervalles de manière à ce qu'ils ne se superposent pas.

Entrée du programme : une fonction et le domaine d'étude de la fonction.

Sortie du programme : des boites représentant l'ensemble des changements de variation de la fonction (intervalle de l'antécédent, intervalle de l'image) ne se superposant pas.

Algorithme 4.2 :

```
Fonction BoxChangementVariation(f,D)
        pile=Changement de variation (f,D)//pile est une pile d'intevalles

        Tant que ∃ i, j dans pile | f(i)∩f(j) ≠ ϕ ou i∩j ≠ ϕ Faire
                Enlever (i et j ,pile )
                [i1,i2] = Diviser l'intervalle i en deux
                [j1,j2] = Diviser l'intervalle j en deux
                Color(Newton(f,i),[i1,i2],i)
                Color(Newton(f,j),[j1,j2],j)
                Pour i dans [i1,i2] ∪ [j1,j2]
                        Si i est rouge Alors
                                Ajouter([i], pile)
                        Fin si
                Fin pour
        Fin Tant que

        Pour i dans pile
                        Ajouter([i,f(i)], pileB)//pileB est une pile de boîtes
        Fin pour
       Retourner(pileB)
 Fin Fonction
```

Remarque 4.3: Cette fonction fait appel à la fonction, Newton (Algorithme 3.3) et à la fonction Color (Algorithme 2.3).

Remarque 4.4: L'ensemble de ces boîtes nous permet d'étudier les domaines sur lesquels le nombre d'images est semblable. Nous ne sommes pas parvenues à programmer ce dernier algorithme.

Annexe

2. Inversion d'ensemble

Le but du programme créé est de trouver les intervalles sur lesquelles une fonction f: D⟶E est négative.

Nous avons travaillé avec la fonction f: x⟶0.5*x^2+3*x

On débute par définir un tableau de x et y pour tracer la fonction: on travaille sur l'intervalle [-10,10] avec un pas de 0.1

-on initialise une pile stack vide dans laquelle on place l'intervalle d'étude ([-10,10])

```
def f(x) :
    return(0.5*x*x+3*x)

class Bac(object):

    def __init__(self, can, num):  # on definit les valeurs initiales


        self.can = can  # ref du canevas
        self.x_can = int(can.cget("width"))
        self.y_can = int(can.cget("height"))

        # conditions intiales
        self.stack = []
        self.stack.append(Interval(-10,10))


        self.zoom = 8
        self.can.create_line(0, -10 , 0, 10)
        self.can.create_line(-10, 0, 10, 0)

        tabX = linspace(-10, 10, 20*10)
        tabY=f(tabX)

        for i in tabX :
            self.drawline(i,f(i) , i+0.1 , f(i+0.1) , "red")
```

Le programme tourne ensuite et commence son analyse de fonction :

```
    def simuler(self):
        stack3=[]
        while(len(self.stack)!=0):
            x=self.stack.pop()
            stack3.append(x)
        while(len(stack3)!=0):
            x=stack3.pop()
            self.Test(x)
            self.stack.append(x)
```

On place dans une nouvelle pile stack3, l'ensemble des éléments de stack pour obtenir les intervalles dans l'ordre croissant des x. Ensuite, on teste chaque intervalle de stack3 avant de les replacer dans stack.

Voici le déroulement de la fonction Test:

```
def Test(self,intervalle) :

    #self.can.delete(ALL)
    self.iTest=Interval(0,-float(inf))
    if (inter(self.iTest,f(intervalle)).isempty()):
        self.drawbox( self.dessin(intervalle), 'blue')
    elif (((inter(self.iTest,f(intervalle)))._inf==f(intervalle)._inf)and(inter(self.iTest,f(intervalle)))._sup==f(intervalle)._sup):
        self.drawbox( self.dessin(intervalle), 'red')
    else:
        self.drawbox( self.dessin(intervalle), 'yellow')
```

La fonction Test prend en argument l'intervalle x de simuler. Elle la compare avec l'intervalle [-∞,0]. Si :

-l'intersection de f(x) avec l'intervalle test est vide: on dessine une boîte bleue

-f(x) est inclus dans l'intervalle test: on dessine une boîte rouge

-si il y a une intersection non vide: on dessine une boîte jaune

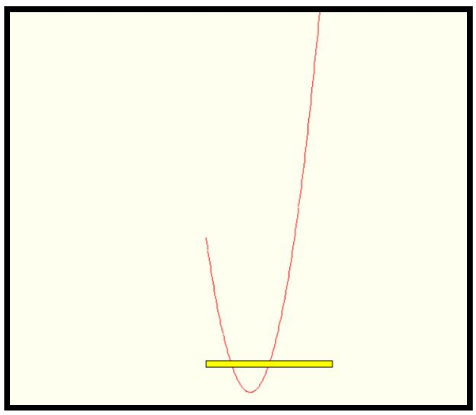
Une fois, l'étude faite, il faut diminuer les intervalles correspondant aux boîtes jaunes pour avoir plus de précision et recommencer les tests. On ajoute alors un bouton "Découpe" associé à la fonction Découpe:

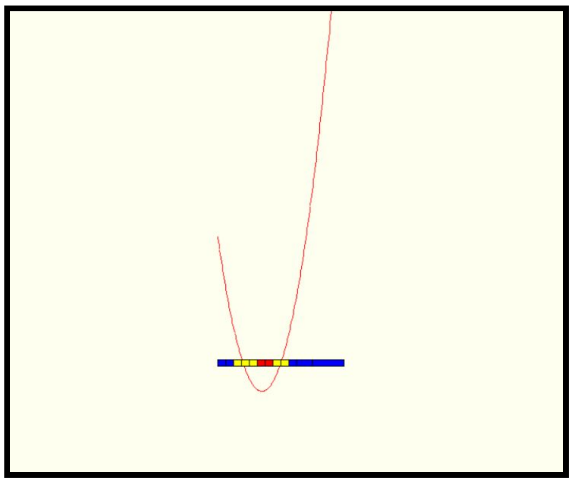
```
def Decoupe(self):
    stack2=[]

    while(len(self.b.stack)!=0):
        x=self.b.stack.pop()
        stack2.append(x)
    while (len(stack2) != 0):
        x = stack2.pop()
        if (inter(self.b.iTest,f(x)).isempty() or ((inter(self.b.iTest,f(x)))._inf==f(x)._inf and (inter(self.b.iTest,f(x))._sup==f(x)._sup))):
            self.b.stack.append(x)
        else:
            mil = (x._sup + x._inf) / 2
            a = Interval(x._inf, mil)
            b = Interval(mil, x._sup)
            self.b.stack.append(a)
            self.b.stack.append(b)
```

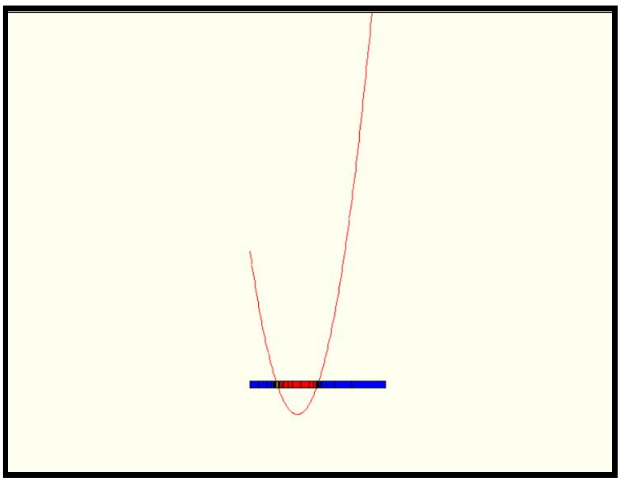
On place dans une nouvelle pile stack2, l'ensemble des éléments de stack pour obtenir les intervalles dans l'ordre croissant des x. On teste ensuite chacun des intervalles pour savoir si on doit les découper en deux et on les ajoute dans la pile stack d'origine.

Si l'intervalle présente une intersection non vide avec l'intervalle [-∞,0], on crée deux nouveaux intervalles a et b contenant chaque moitié de l'intervalle x jaune que l'on ajoute à la pile stack à la place de l'intervalle x.

Voici une évolution de l'algorithme appliqué à la fonction f: x⟶0.5*x^2+3*x :







3. Newton par intervalles

On a travaillé sur la fonction f: x -> 0.5*x^3+2*x^2+x+2

On peut ainsi appliquer la méthode de Newton pour déterminer les annulations de la dérivée:

```
def f(x):
    return (0.5*x*x*x+2*x*x+x+2)


def df(x):
    return (1.5*x*x+4*x+1)


def ddf(x):
    return (3*x+4)


def Newton(I):
    mil = (I._sup + I._inf ) / 2
    return ((mil + df(mil) / ddf(I )) )
```

On place dans une nouvelle pile stack3, l'ensemble des éléments de stack pour obtenir les intervalles dans l'ordre croissant des x. Ensuite, on teste chaque intervalle de stack3 avant de les replacer dans stack.

```
def simuler(self):
    stack3 = []
    while (len(self.stack) != 0):
        x = self.stack.pop()
        stack3.append(x)
    while (len(stack3) != 0):
        x = stack3.pop()
        self.Test(x, Newton(x))
        self.stack.append(x)
```

Voici le déroulement de la fonction Test:

```
def Test(self, interv, newton):

    # self.can.delete(ALL)
    self.iTest = newton
    if (inter(self.iTest, interv).isempty()):
        self.drawbox(self.dessin(interv), 'blue')

    elif (((inter(self.iTest, interv))._inf == self.iTest._inf) and ((inter(self.iTest, interv))._sup == self.iTest._sup)):
        print(interv)
        self.drawbox(self.dessin(interv), 'red')

    else:
        self.drawbox(self.dessin(interv), 'yellow')
```

La fonction Test prend en argument l'intervalle x de simuler ainsi que l'intervalle obtenu par la méthode de Newton. Si :

-l'intersection de x avec l'intervalle de Newton est vide: on dessine une boîte bleue

-l'intervalle de Newton est inclus dans x: on dessine une boîte rouge(unique x tq f'(x)=0)

-si il y a une intersection non vide: on dessine une boîte jaune

Une fois, l'étude faite, il faut diminuer les intervalles correspondant aux boîtes jaunes pour avoir plus de précision et recommencer les tests.

Voici un aperçu de l'évolution obtenue: