UYI 23 November 2021

INFO 4127

DESCENT GRADIANT ALGORITHMS

GROUP 1 MEMBER'S:

Name	Surname	Matricule	Email
DJIEMBOU TIENTCHEU	Victor Nico	17T2051	nico.djiembou@facsciences-uy1.cm / Viclegranddab@gmail.com
KENFACK TEMGOUA	Vanessa	17J2871	vanessa.kenfack@facsciences-uy1.cm / Vanstemgoua21@gmail.com
DONGMO NGUIMKENG	BIBICHE LAURE	21S2812	laurenguimkeng2@gmail.com
NYA NJIKE	ARMEL	21S2802	Armel.njike@yahoo.com

Double-click (or enter) to edit

Purpose of the assignment:

> - *Be able to implement Gradiant Descent Algorithms*
>
> > 1. with **Fixed Step**
> > 2. with **Optimal Step**
> > 3. with **Fixed Step using Armijo condition**
> > 4. with **Fixed Step using Wolfe condition**.
>
> - *Apply an experimental study of each approach*

1. with **Limits explanation** of each algorithms
2. with **Temporal evaluation** of each algorithms
3. with **Space evaluation** of each algorithms
4. with **Data visualization** of each algorthms outputs.

INTRODUCTION

The focus here is on the design of numerical methods for solving unconstrained differentiable optimization problems. In other words, the constraint domain X is an open of R^n. Thus, we seek to solve the problem:

$$(P) = min_{xinR^n} f(x)$$

where f is a real-valued function defined on R^n and assumed to be differentiable, or even twice differentiable.

General principle of descent methods

General principle of descent methods Starting from an arbitrarily chosen point x0, a descent algorithm will try to generate a sequence of iterates $(x_k), kinN$ defined by :

$x_{k+1} = x_k + s_k d_k$

and such that :

k in N, $f(x_{k+1})$ less than or equals to $f(x_k)$.

Gradiant Descent Algorithm environment preparing

Import libraries

```
import sympy as sp # for symbolic mathematic manipulation
import numpy as np # for mathematic utils usage
```

```
from sympy.parsing.sympy_parser import parse_expr # convert string to mathematic expression
# implicite 2x to 2*x convertion
from sympy.parsing.sympy_parser import standard_transformations,implicit_multiplication_application
transformations = standard_transformations + (implicit_multiplication_application,)

import time # for time evaluation

# for plotting
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
from matplotlib.colors import LogNorm
import matplotlib.pyplot as plt
```

Define useful methods

create symbolic expression function

```
def toExp(prefix_exp="(x**2)/2 + (7*y**2)/2"):
    """Summary or Description of the Function

    Parameters:
    prefix_exp (string): expression of a function

    Returns:
    toExp(prefix_exp):Returning  a mathematic expression of the function

   """
    return parse_expr(prefix_exp)
```

```
toExp()
```

$$\frac{x^2}{2} + \frac{7y^2}{2}$$

create symbolic args function

```
def toArgs(prefix_args="x,y"):
    """Summary or Description of the Function

    Parameters:
    prefix_args (string): suite of caracters separate by <<,>>

    Returns:
    toArgs(prefix_args):Returning  a tuple of mathematical symbole link to caracters passed as parameters

   """
    return sp.symbols(prefix_args)
```

```
toArgs()
```

```
(x, y)
```

Gradiant function

$$\textbf{Gradiant(f)} = \frac{d\mathbf{f}}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$$

```
def gradiant(f, args):
    """Summary or Description of the Function

    Parameters:
    f (expression): expression of function
    args (tuple): differents symbols in expression
```

```
    Returns:
    gradiant(f,args):Returning  a matrix (col vector) containing the result of the partial
    differential of the function following each symbol of the expression

   """
    Df = []
    for var in args:
        Df.append(f.diff(var))
    return sp.Matrix(Df)
```

```
gradiant(toExp(),toArgs())
```

$$\begin{bmatrix} x \\ 7y \end{bmatrix}$$

Hessian function

$$\textbf{Hessian(f)} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 x_n} \\ \frac{\partial^2 f}{\partial x_2 x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n x_1} & \frac{\partial^2 f}{\partial x_n x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

```
def hessian(f, args):
    """Summary or Description of the Function

    Parameters:
    f (expression): expression of function
    args (tuple): differents symbols in expression

    Returns:
```

```
    hessian(f,args):Returning  a matrix containing the result of the second partial
    differential of the function following each symbol of the expression

   """
    H = []
    for i in args:
        line = []
        for j in args:
            line.append(f.diff(j).diff(i))
        H.append(line)
    return sp.Matrix(H)
```

```
hessian(toExp(),toArgs())
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 7 \end{bmatrix}$$

Trace Logs function

```
def traceLogs(s,k,f, norm, X, args):
    """Summary or Description of the Function

    Parameters:
    s (float): step descent
    args (tuple): differents symbols in expression
    f (expression): mathematic expression of the function studied
    X (matrix): coordinate of current point
    norm (float): norm of a vector field or absolute value of a scalar field

    Returns:
    traceLogs(s,k,f, norm, X, args):Returning  a console information about current iteration

   """
    X_S = [] #X[k]
```

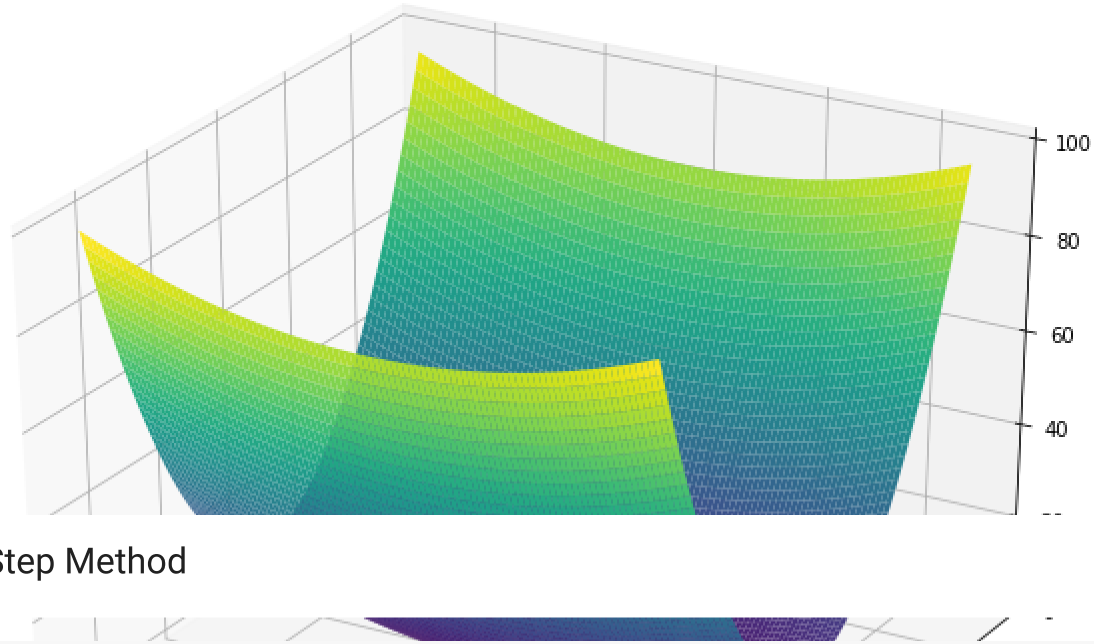
```
    info = ""
    for i in range(len(args)):
        info+= str(args[i])+":  "+str(X[i]) +" "
        X_S.append((args[i], X[i]))
    print("""k: {}   f(xk, yk): {}  ||∇f(xk,yk)||: {} sk: {}  {}   """.format(k, f.subs(X_S), norm, s,info))
```

Preview of our study function

```
def h(x, y):
    return (x**2)/2 + (7*y**2)/2

fig = plt.figure()
fig.set_size_inches(9, 7, forward=True)
#ax = Axes3D(fig,azim=-19, elev=19)
ax = Axes3D(fig)
a = np.arange(-5,5,0.1)
b = np.arange(-5,5,0.1)
A,B = np.meshgrid(a,b)
C = h(A,B)
#ax.plot_wireframe(A,B,C,rstride=1, cstride=1)
ax.plot_surface(A,B,C,rstride=1, cstride=1,cmap = plt.cm.viridis)
plt.xlabel("parameter 1 : x")
plt.ylabel("parameter 2 : y")
```

```
Text(0.5, 0, 'parameter 2 : y')
```



Define Fixed Step Method

```
def fixedStep(f, args, s, init_point, e = 10**(-5), iter=100 ):
    """Summary or Description of the Function

    Parameters:
    s (float): step descent
    args (tuple): differents symbols in expression
    f (expression): mathematic expression of the function studied
    init_point (list): coordinate of init point where we start to find the optimum of the function
    e (float): optimum search precision
    iter (int): limit of iteration to find optimum

    Returns:
    fixedStep(f, args, s, init_point, e, iter):Returning  an approximation of optimum of the function

   """
    Df = gradiant(f,args) #calcul differential of the function f
    X = sp.Matrix(init_point) #convert init_point in matrix n*m
```

```
    k = 0
    start_time = time.time()
    while (k < iter): # first stop condition
        X_S = [] #X[k]
        for i in range(len(args)):
            X_S.append((args[i], X[i])) #link args to init value to substitude in Gradiant matrix

        grad = [] #Lf(X[k])
        for expr in Df:
            grad.append(expr.subs(X_S)) #substitude values in gradiant matrix
        grad = sp.Matrix(grad) # convert the result in matrix

        if ((grad.norm()) < e): #break if ||lf(X[k])|| < precision is second stop condition
            break

        X = sp.Matrix(X - s*grad) #calcul the X[k+1] = X[k]-s*Lf(X[k])
        #traceLogs(s,k,f, grad.norm(), X, args)
        #ax.scatter(X[0],X[1],h(X[0],X[1]), marker="o", color="#00FF00")
        #plt.draw()
        #plt.pause(0.05)
        k += 1
    end_time = time. time()
    time_elapsed = (end_time - start_time)
    X_S = [] #X[k]
    for i in range(len(args)):
        X_S.append((args[i], X[i])) #link args to init value to substitude in Gradiant matrix


    print('FixedStep function has taken : {} second for {} iteration, image of function {} and the result is :'.format(tim
    return X #return the last X
```

```
fixedStep(toExp(),toArgs(),0.2,[7,1.5])
```

```
FixedStep function has taken : 0.10810422897338867 second for 61 iteration, image of function 3.68251476978936E-11 ar
```

```
fixedStep(toExp(),toArgs(),0.1,[7,1.5])
```

```
FixedStep function has taken : 0.15582895278930664 second for 100 iteration, image of function 1.72849438162055E-8 ar
```

$$\begin{bmatrix} 0.000185929792213112 \\ 7.73066281098007 \cdot 10^{-53} \end{bmatrix}$$

```
fixedStep(toExp(),toArgs(),0.15,[7,1.5])
```

```
FixedStep function has taken : 0.12709712982177734 second for 83 iteration, image of function 4.70660501189075E-11 ar
```

$$\begin{bmatrix} 9.70216987265298 \cdot 10^{-6} \\ -1.55096364853682 \cdot 10^{-108} \end{bmatrix}$$

```
fixedStep(toExp(),toArgs(),0.25,[7,1.5])
```

```
FixedStep function has taken : 0.08877015113830566 second for 49 iteration, image of function 1.84592303558624E-11 ar
```

$$\begin{bmatrix} 5.28566879331809 \cdot 10^{-6} \\ -1.13264331285388 \cdot 10^{-6} \end{bmatrix}$$

```
fixedStep(toExp(),toArgs(),0.3,[7,1.5])
```

```
FixedStep function has taken : 0.15798115730285645 second for 100 iteration, image of function 1495504052.12611 and t
```

$$\begin{bmatrix} 2.26413355673733 \cdot 10^{-15} \\ 20670.9185097332 \end{bmatrix}$$

Name	Values				
s	0.2	0.1	0.15	0.25	0.3
Elapsed times	0.10810422897338867	0.15582895278930664	0.12709712982177734	0.08877015113830566	0.15798115730285645
Nb iterations	61	100	83	49	100
Nb of exact significant numbers.	-11	-8	-11	-11	5

Define Optimal Step Method

function to solve: optimal setp s_k solution of :

$$min f(x_k + sd_k)$$

```
def s_k(xk,dk,args,f):
    """Summary or Description of the Function

    Parameters:
    xk (matrix): current point
    dk (matrix): direction of greatest gradient
    f (expression): mathematic expression of the function studied
    args (tuple): symbols inside function expression

    Returns:
    s_k(xk,dk,args,f):Returning  optimal step

   """
    s = sp.symbols("s") # define our symbol arg
    X = sp.Matrix(xk - s*dk) # calcul Xk+sdk to get our Xk+1
    X_K = [] #X[k]
    for i in range(len(args)):
        X_K.append((args[i], X[i]))
    phi = f.subs(X_K)
    grad = gradiant(phi,('s'))
    return sp.solve(grad,s)[s]
```

```
def optimalStep(f, args, init_point, e = 10**(-5), iter=1000 ):
    """Summary or Description of the Function

    Parameters:
    args (tuple): differents symbols in expression
    f (expression): mathematic expression of the function studied
    init_point (list): coordinate of init point where we start to find the optimum of the function
    e (float): optimum search precision
    iter (int): limit of iteration to find optimum
```

```
    Returns:
    optimalStep(f, args, init_point, e, iter):Returning  an approximation of optimum of the function

    """
    Df = gradiant(f,args) #calcul gradiant of the function
    X = sp.Matrix(init_point) #convert init_point in matrix n*m

    k = 0
    start_time = time.time()
    while (k < iter):
        X_S = [] #X[k]
        for i in range(len(args)):
            X_S.append((args[i], X[i])) #link coord to init value to substitude in Gradiant matrix

        grad = [] #Lf(X[k])
        for expr in Df:
            grad.append(expr.subs(X_S)) #substitude values in gradiant matrix
        grad = sp.Matrix(grad) # convert the result in matrix

        if ((grad.norm()) < e): #break if ||lf(X[k])|| < precision
            break

        # determine sk
        s = s_k(X,grad,args,f)
        #print("OKK")
        #break
        X = sp.Matrix(X - s*grad) #calcul the X[k+1] = X[k]-s*Lf(X[k])
        #ax.scatter(X[0],X[1],h(X[0],X[1]), marker="o", color="#00FF00")
        #plt.draw()
        #plt.pause(0.05)
        #traceLogs(s,k,f, grad.norm(), X, args)
        k += 1
    end_time = time. time()
    time_elapsed = (end_time - start_time)
    X_S = [] #X[k]
    for i in range(len(args)):
```

```
        X_S.append((args[i], X[i])) #link args to init value to substitude in Gradiant matrix


    print('FixedStep function has taken : {} second for {} iteration, image of function {} and the result is :'.format(tim
    return X #return the last X
```

```
optimalStep(toExp(),toArgs(),[7,1.5])
```

FixedStep function has taken : 2.7532622814178467 second for 43 iteration, image of function 2.50227013205719E-11 anc

$$\begin{bmatrix} 6.85985549474891 \cdot 10^{-6} \\ -6.53319570928464 \cdot 10^{-7} \end{bmatrix}$$

```
optimalStep(toExp(),toArgs(),[7,1.5],10**(-4))
```

FixedStep function has taken : 2.02842116355896 second for 37 iteration, image of function 1.22564117767989E-9 and th

$$\begin{bmatrix} 4.80097608837455 \cdot 10^{-5} \\ -4.57235817940434 \cdot 10^{-6} \end{bmatrix}$$

```
optimalStep(toExp(),toArgs(),[7,1.5],10**(-6))
```

FixedStep function has taken : 2.3732500076293945 second for 51 iteration, image of function 1.39624615510908E-13 anc

$$\begin{bmatrix} 5.1242329683565 \cdot 10^{-7} \\ -4.88022187462517 \cdot 10^{-8} \end{bmatrix}$$

```
optimalStep(toExp(),toArgs(),[7,1.5],10**(-7))
```

FixedStep function has taken : 2.386981725692749 second for 57 iteration, image of function 2.85057740760852E-15 and

$$\begin{bmatrix} 7.32173979567855 \cdot 10^{-8} \\ -6.97308551969381 \cdot 10^{-9} \end{bmatrix}$$

```
optimalStep(toExp(),toArgs(),[7,1.5],10**(-8))
```

```
FixedStep function has taken : 2.657686233520508 second for 65 iteration, image of function 1.59059875040037E-17 and
```
$\begin{bmatrix} 5.46925521615783 \cdot 10^{-9} \end{bmatrix}$

Name	Values				
e	10**(-5)	10**(-4)	10**(-6)	10**(-7)	10**(-8)
Elapsed times	2.7532622814178467	2.02842116355896	2.3732500076293945	2.386981725692749	2.657686233520508
Nb iterations	43	37	51	57	65
Nb of exact significant numbers.	-11	-9	-13	-15	-17

Define Fixed Step Method with Armijo condition

Armijo Condition:

$$f(x + sd) <= f(x) + \gamma s(\frac{\partial \mathbf{f}}{\partial X} d), 0 < \gamma < 1$$

```
def MeriteFunction(dk,args,f):
    """Summary or Description of the Function

    Parameters:
    dk (matrix): direction of greatest gradient
    f (expression): mathematic expression of the function studied
    args (tuple): symbols inside function expression

    Returns:
    MeriteFunction(dk,args,f):Returning  expression of merite function

   """
    s = sp.symbols("s") # define our symbol arg
    X = sp.Matrix(args) + s*dk # calcul Xk+sdk to get our Xk+1
    #print(function)
    #print(X)
    X_S = [] #X[k]
    for i in range(len(args)):
        X_S.append((args[i], X[i]))
```

```
    M = f.subs(X_S)
    return M
```

```
def·ArmijoSuffCond(gk,args,f,e=10**(-4)):
····"""Summary·or·Description·of·the·Function

····Parameters:
····gk·(matrix):·function·gradient
····f·(expression):·mathematic·expression·of·the·function·studied
····args·(tuple):·symbols·inside·function·expression
····e·(float):·precision·of·optimum·armijo·condition·search

····Returns:
····ArmijoSuffCond(gk,args,f,e):Returning·expression·of·second·member·of·Armijo·condition

···"""
····s·=·sp.symbols("s")·#·define·our·symbol·arg
····X·=·f·+·e*s*((sp.transpose(gk)*(-gk))[0])·#·calcul·F(X)+esDF(X)Tdk·
····M·=·MeriteFunction(-gk,args,f)
····solu·=·sp.solve(M-X,·s)
····return·solu
```

```
def ArmijoFixedStep(f, args, init_point, e = 10**(-5), iter=100 ):
    """Summary or Description of the Function

    Parameters:
    gk (matrix): function gradient
    f (expression): mathematic expression of the function studied
    args (tuple): symbols inside function expression
    e (float): precision of optimum search

    Returns:
    ArmijoOptimalStep(gk,args,f,e):Returning expression of second member of Armijo condition

   """
```

```
    Df = gradiant(f,args) #calcul gradiant of the function
    X = sp.Matrix(init_point) #convert init_point in matrix n*m

    p = sp.Matrix(ArmijoFixedStep(Df,args,f))
    k = 0
    start_time = time.time()
    while (k < iter):
        X_S = [] #X[k]
        for i in range(len(args)):
            X_S.append((args[i], X[i])) #link coord to init value to substitude in Gradiant matrix

        grad = [] #Lf(X[k])
        for expr in Df:
            grad.append(expr.subs(X_S)) #substitude values in gradiant matrix
        grad = sp.Matrix(grad) # convert the result in matrix

        # determine sk
        s = (p[1].subs(X_S)-p[0].subs(X_S))/2
        #print("OKK")
        #break

        N = sp.Matrix(X - s*grad) #calcul the X[k+1] = X[k]-s*Lf(X[k])
        X_S1 = [] #X[k]
        for i in range(len(args)):
            X_S1.append((args[i], N[i]))
        #ax.scatter(X[0],X[1],h(X[0],X[1]), marker="o", color="#00FF00")
        #plt.draw()
        #plt.pause(0.05)
        #traceLogs(s,k,f, grad.norm(), X, args)


        if (abs(f.subs(X_S1)-f.subs(X_S)) < e*(1+abs(f.subs(X_S)))): #break if ||f(xk+1)-f(x)|| < e*(1+|f(xk))
            break
        X = N
        k += 1
    end_time = time. time()
    time_elapsed = (end_time - start_time)
```

```
    X_S = [] #X[k]
    for i in range(len(args)):
        X_S.append((args[i], X[i])) #link args to init value to substitude in Gradiant matrix


    print('FixedStep function has taken : {} second for {} iteration, image of function {} and the result is :'.format(tim
    return X #return the last X
```

```
ArmijoFixedStep(toExp(),toArgs(),[7,1.5])
```

```
FixedStep function has taken : 0.05548286437988281 second for 22 iteration, image of function 0.0000203961523448521 a
```

$$\begin{bmatrix} 0.00555336856323215 \\ 0.0011923807092541 \end{bmatrix}$$

```
def WolfeSuffCond(gk,args,f,e1 = 10**(-4),e2=0.99):
    """Summary or Description of the Function

    Parameters:
    gk (matrix): function gradient
    f (expression): mathematic expression of the function studied
    args (tuple): symbols inside function expression
    e (float): precision of optimum armijo condition search

    Returns:
    ArmijoSuffCond(gk,args,f,e):Returning expression of second member of Armijo condition

   """
    s = sp.symbols("s") # define our symbol arg
    X = f + e1*s*((sp.transpose(gk)*(-gk))[0]) # calcul F(X)+esDF(X)Tdk
    M = MeriteFunction(-gk,args,f)
    AM = M-X
    DM = (-gk)*gradiant(M,('s'))
    DM -= e2*(sp.transpose(gk)*(-gk))

    solu = sp.solve(sp.Matrix([AM,DM]), s)
```

```
    print(solu)
    return solu

def WolfeFixedStep(f, args, init_point, e = 10**(-5), iter=100 ):
    """Summary or Description of the Function

    Parameters:
    gk (matrix): function gradient
    f (expression): mathematic expression of the function studied
    args (tuple): symbols inside function expression
    e (float): precision of optimum search

    Returns:
    ArmijoOptimalStep(gk,args,f,e):Returning expression of second member of Armijo condition

   """
    Df = gradiant(f,args) #calcul gradiant of the function
    X = sp.Matrix(init_point) #convert init_point in matrix n*m

    p = sp.Matrix(WolfeSuffCond(Df,args,f))
    k = 0
    start_time = time.time()
    while (k < iter):
        X_S = [] #X[k]
        for i in range(len(args)):
            X_S.append((args[i], X[i])) #link coord to init value to substitude in Gradiant matrix

        grad = [] #Lf(X[k])
        for expr in Df:
            grad.append(expr.subs(X_S)) #substitude values in gradiant matrix
        grad = sp.Matrix(grad) # convert the result in matrix

        # determine sk
        s = (p[1].subs(X_S)-p[0].subs(X_S))/2
        #print("OKK")
        #break

        N = sp.Matrix(X - s*grad) #calcul the X[k+1] = X[k]-s*Lf(X[k])
```

```
        X_S1 = [] #X[k]
        for i in range(len(args)):
            X_S1.append((args[i], N[i]))
        #ax.scatter(X[0],X[1],h(X[0],X[1]), marker="o", color="#00FF00")
        #plt.draw()
        #plt.pause(0.05)
        #traceLogs(s,k,f, grad.norm(), X, args)


        if (abs(f.subs(X_S1)-f.subs(X_S)) < e*(1+abs(f.subs(X_S)))): #break if ||f(xk+1)-f(x)|| < e*(1+|f(xk))
            break
        X = N
        k += 1
    end_time = time. time()
    time_elapsed = (end_time - start_time)
    X_S = [] #X[k]
    for i in range(len(args)):
        X_S.append((args[i], X[i])) #link args to init value to substitude in Gradiant matrix


    print('FixedStep function has taken : {} second for {} iteration, image of function {} and the result is :'.format(tim
    return X #return the last X
```

```
WolfeFixedStep(toExp(),toArgs(),[7,1.5])
```

```
FixedStep function has taken : 0.05428647994995117 second for 22 iteration, image of function 0.000020396152344852 1 a
```

$$\begin{bmatrix} 0.00555336856323215 \\ 0.0011923807092541 \end{bmatrix}$$

Conclusion

At the end of our experiment, we can notice :

the fixed step algorithm has a better execution time than the optimal step algorithm. However, it offers a better accuracy of the approximate value of the optimum of the function to be optimized. To overcome this, we add two approaches to the fixed step algorithm: Armijo's condition and Wolfe's condition. Armijo avoids the selection of too large steps for the function except that this correction can imply to have very small steps. It is with the aim of having a step that is neither very big nor very small that I the condition of wolfe is set up.

as recapitulative, we have:

Algorithm	elapsed time	Nb of iteration	exact signification numbers
Fixed Step Algorithm	***	*	*
Optimal Step Algorithm	**	***	***
Armijo Fixed Step Algorithm	***	**	***

✓ 0s completed at 1:25 PM