NAME: DHRUV KAMALESH KUMAR
NUID: 002767429

HOMEWORK-5

Problem 1 Interval Bisection Search and Golden Section Search - 1-D Optimization

We have learned from lecture that the Interval Bisection Search (IBS) is a derivative-based Search Method for a global minimum x_{opt} in a box constraint. Additionally, both R and Python implement 1-D direct searches (no gradient) using Golden Section Search (GSS). GSS chooses an interior point x_{GSS} such that the larger subinterval is the same fraction of $U - L$ as the smaller subinterval is to the larger subinterval.

Wiki summarizes the further details of GSS in the following figure.

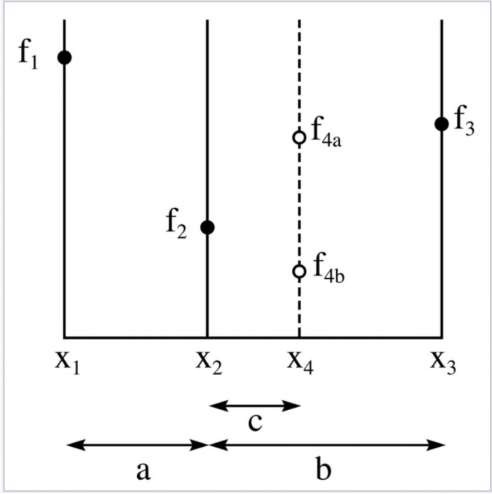


Diagram of a golden-section search. The initial triplet of x values is {x1,x2,x3}. If $f(x_4)=f_{4a}$, the triplet $\{x_1,x_2,x_4\}$ is chosen for the next iteration. If $f(x_4)=f_{4b}$, the triplet $\{x_2,x_4,x_3\}$ is chosen.

In this exercise, you will compare the number of iterations for IBS and GSS to reach a certain tolerance condition. Does GSS reach the tolerance condition with fewer iterations, on average?

We will consider minimizing the convex function $f(x) = x^2$ in this problem. Of course, the minimum occurs at $x = 0$, but we focus here on the smallest number of iterations T until $ U_T - L_T < $ `tol`, where `tol` is a constant. The initial search interval $[L_0, U_0]$ will be randomly produced as follows: $L_0 \sim U[-10, 0]$ and $U_0 \sim U[0, 10]$. (U stands for uniform distribution.) Let's terminate each algorithm using the above tolerance condition, for the same value of `tol`.

Plot the log10 of the average value of T for IBS (blue lines and points) and GSS (red lines and points) as a function of $log_10 $ `tol`. Consider 50 trials of each algorithm to compute this average. To prevent long run times, set the maximum number of iterations to 1000. Let the value of `tol` vary in the interval $[10^{-8}, 10^{0}]$. Which algorithm performs best, and why?

Additional credit: Can you improve the GSS algorithm in this case? Consider modifying the GSS to reduce the average number of iterations. Change the golden ratio by +10% and -10%, and compare the average number of iterations to that of GSS as above. Plot the results for GSS in red, +10% in blue, and -10% in magenta. Which of these three methods performs best in this case, and why?

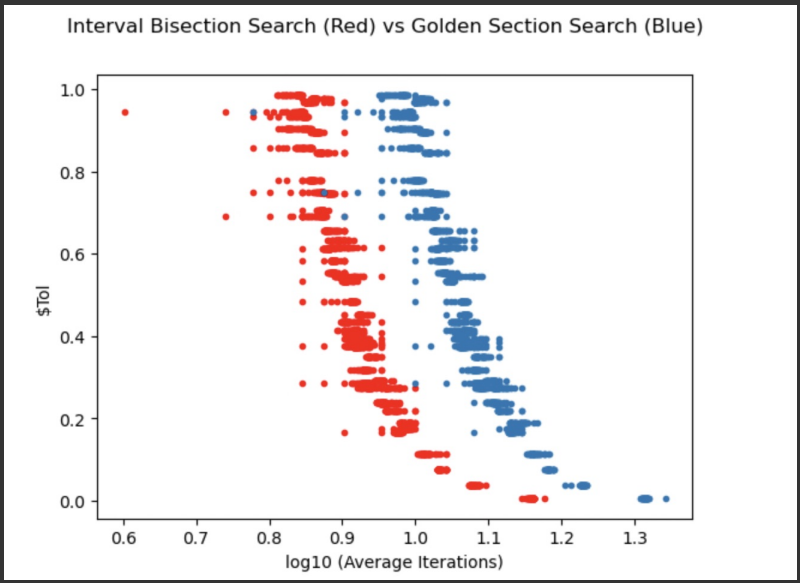
⇒ CODE AND OUTPUT:

```
import matplotlib.pyplot as plt
import math
import random

gr = (math.sqrt(5) + 1) / 2

def IBS(L,U,tol):
    f=lambda x: 2*x
    ibs=[]
    iterations=0
    a=L
    b=U
    while (b-a) > tol and iterations<1000:
        if f((a+b)/2) > 0:
            a=a
            b=(a+b)/2
        else:
            a=(a+b)/2
            b=b
        iterations+=1
    ibs.append(iterations)
    ibs.append((a+b)/2)
    return ibs

def GSS(L,U,tol):
    a=L
    b=U
    f=lambda x: x*x
    iterations=0
    gss=[]
    c=b-(b-a)/gr
    d=a+(b-a)/gr
    while (b - a) > tol and iterations<1000:
        if f(c) < f(d):
            b=d
        else:
            a=c
        iterations+=1
        c = b - (b - a) / gr
        d = a + (b - a) / gr
    gss.append(iterations)
    gss.append((a+b)/2)
    return gss

print("Part one of the problem 1 \nGenerating ")
L=random.randint(-10, 0)
U=random.randint(0, 10)
print("Part one of the problem 1 \nRandomly generated values of L[0] = ",L ," U[0] = ",U)
print("Number of iterations for Interval Bisection Search to reach tolerance 0.01 is ",IBS(L,U,0.01)[0])
print("Number of iterations for Golden Section Search to reach tolerance 0.01 is ",GSS(L,U,0.01)[0])
print("Hence IBS reach the tolerance condition with fewer iterations, on average")

def Modified_GSS_plus10(L,U,tol):
    a=L
    b=U

    newgr=gr*(1.1)
    f=lambda x: x*x
```

```
    iterations=0
    gss=[]
    c=b-(b-a)/newgr
    d=a+(b-a)/newgr
    while(b-a)>tol and iterations<1000:
        if f(c) < f(d):
            b=d
        else:
            a=c
        iterations+=1
        c=b-(b-a)/ newgr
        d=a+(b-a)/ newgr
    gss.append(iterations)
    gss.append((a+b)/2)
    return gss

def Modified_GSS_min10(L,U,tol):
    a=L
    b=U
    newgr=gr*(0.9)
    f=lambda x: x*x
    iterations=0
    gss=[]
    c=b-(b-a)/newgr
    d = a + (b - a) / newgr
    while (b - a) > tol and iterations<1000:
        if f(c) < f(d):
            b=d
        else:
            a=c
        iterations+=1
        c = b - (b - a) / newgr
        d = a + (b - a) / newgr
    gss.append(iterations)
    gss.append((a+b)/2)
    return gss

def graph():
    x_ibs=[]
    x_gss=[]
    y_ibs=[]
    y_gss=[]
    i=0.0000001
    for k in range(1,51):
        i= random.uniform(0.00000001, 1)
        t_avg_ibs=[]
        t_avg_gss=[]
        for j in range(1,51):
            L=random.randint(-100, 0)
            U=random.randint(0, 100)
            z=IBS(L,U,i)
            t_avg_ibs.append(z[0])
            t_avg_gss.append(GSS(L,U,i)[0])
            x_ibs.append(math. log10(sum(t_avg_ibs)/len(t_avg_ibs)))
            y_ibs.append(i)
            x_gss.append(math.log10(sum(t_avg_gss)/len(t_avg_gss)))
            y_gss.append(i)
    plt.plot(x_ibs, y_ibs, 'r.')
```

```
    plt.plot(x_gss, y_gss, '.')
    plt.xlabel('log10 (Average Iterations)')
    plt.ylabel('$Tol')
    plt.suptitle('Interval Bisection Search (Red) vs Golden Section Search (Blue)')
    plt.show()
    x_gssplus10=[]
    x_gssmin10=[]
    x_gss=[]
    y_gssplus10=[]
    y_gssmin10=[]
    y_gss=[]
    i=0.0000001
    for k in range(1,51):
        i= random.uniform(0.00000001, 1)
        t_avg_gssplus10=[]
        t_avg_gssmin10=[]
        t_avg_gss=[]
        for j in range(1,51):
            L=random.randint(-100, 0)
            U=random.randint(0, 100)
            t_avg_gssplus10.append(Modified_GSS_plus10(L,U,i)[0])
            t_avg_gssmin10.append(Modified_GSS_min10(L,U,i)[0])
            t_avg_gss.append(GSS(L,U,i)[0])
        x_gssplus10.append(sum(t_avg_gssplus10)/len(t_avg_gssplus10))
        y_gssplus10.append(i)
        x_gssmin10.append(sum(t_avg_gssmin10)/len(t_avg_gssmin10))
        y_gssmin10.append(i)
        x_gss.append(sum(t_avg_gss)/len(t_avg_gss))
        y_gss.append(i)
    plt.suptitle('Average iterations of GSS with the golden ratio by +10% and - 10% in blue and \n magenta
color respectively ')
    plt.plot(x_gssplus10, y_gssplus10, '.')
    plt.plot(x_gssmin10, y_gssmin10, '.', color='magenta')
    plt.plot(x_gss, y_gss, 'r.')
    plt.xlabel('log10 (Average Iterations)')
    plt.ylabel('$Tol')
    plt.show()
    return

graph()
```

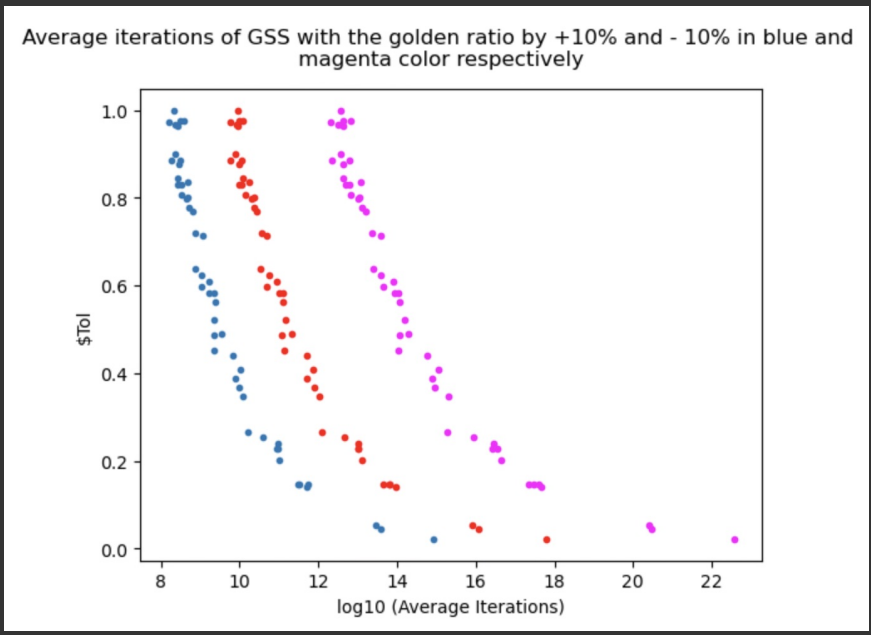
```
Part one of the problem 1
Generating
Part one of the problem 1
Randomly generated values of L[0] =  -10  U[0] =  4
Number of iterations for Interval Bisection Search to reach tolerance 0.01 is  11
Number of iterations for Golden Section Search to reach tolerance 0.01 is  16
Hence IBS reach the tolerance condition with fewer iterations, on average
```



1. IBS performs the best as the number of iterations are less than GSS.

2. IBS is faster than GSS.

3. In case of GSS when multiple zeros or optima exist, no guidance as to which will be chosen.



As we can see through graph if we increase the golden ratio by 10% the number of iterations are reduced hence performance is improved whereas if we decrease golden ratio 10% the number of iterations are increased hence performance is degraded.

2. Convexity in 1- and 2-Dimensions

1. Consider the objective function f(v)=(1−v)2 for a scalar variable v. Using first and second derivatives confirm or reject that f is convex. If you reject convexity everywhere, please specify for which values of v that f(v) is not convex.

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

$$f(v) = 1 + v^2 - 2v$$

$$\frac{d(f(v))}{dv} = f'(v) = 2v - 2$$

$$\frac{d(f'(v))}{dv} = f''(v) = 2$$

$f''(v)$ is always positive for all v.

$\therefore f(v)$ is concave upward for all v.

2. Suppose that two functions g(v) and h(v) are both convex functions. Confirm or reject that their sum is convex by the method above.

Let us take any two elements v_1 & v_2.

If $g(v) + h(v)$ is convex then below equation must hold

$$f(\alpha v_1 + (1-\alpha)v_2) + g(\alpha v_1 + (1-\alpha)v_2) \leq \alpha(f(v_1) + g(v_1)) + (1-\alpha)(f(v_1) + g(v_2))$$

Since f and g are convex functions, we have the following equation for any point v_1 & v_2.

$$f(\alpha v_1 + (1-\alpha)v_2) \leq \alpha f(v_1) + (1-\alpha) f(v_2)$$

↳ (1)

$$g(\alpha v_1 + (1-\alpha)v_2) \leq \alpha g(v_1) + (1-\alpha) g(v_2)$$ → ②

$$\alpha \in [0,1]$$

① + ②

$$f(\alpha v_1 + (1-\alpha)v_2) + g(\alpha v_1 + (1-\alpha)v_2) \leq \alpha f(v_1) + (1-\alpha) f(v_2) + \alpha g(v_1) + (1-\alpha) g(v_2)$$

$$\Rightarrow f(\alpha v_1 + (1-\alpha)v_2) + g(\alpha v_1 + (1-\alpha)v_2) \leq \alpha (f(v_1) + g(v_1)) + (1-\alpha)(f(v_2) + g(v_2))$$

∴ Sum of $f(v)$ & $g(v)$ is a convex function.

3. Let g(v)=(a−v2)2, where a is a fixed real number. Confirm or reject the claim that g(v) is convex for any constant a. If you reject convexity everywhere, please specify which for values of v that g(v) is not convex (as a function of a).

$$g(v) = (a - v^2)^2 = a^2 + v^4 - 2av^2$$

$$\frac{d(g(v))}{d(v)} = g'(v) = 4v^3 - 4av$$

$$\frac{d(g'(v))}{d(v)} = g''(v) = 12v^2 - 4a$$

$$\Rightarrow v^2 = \frac{4a}{12} \Rightarrow v = \sqrt{\frac{a}{3}}$$

If $v < \sqrt{\frac{a}{3}} \rightarrow g''(v) < 0 \rightarrow$ NOT CONVEX

If $v = \sqrt{\frac{a}{3}} \rightarrow g''(v) = 0$

If $v > \sqrt{\frac{a}{3}} \rightarrow g''(v) > 0 \rightarrow$ CONVEX

$\therefore g(v)$ is concave if $v > \sqrt{\frac{a}{3}}$ & $g(v)$ is not concave for $v < \sqrt{\frac{a}{3}}$.

4. For a vector v=[v1 v2]T, let p(v)=(1−v1)2+100∗(v2−v21)2. Calculate the gradient and Hessian of p, and confirm or reject that p is convex everywhere in ℝ2. If the function is not convex everywhere, please specify with a plot the region where convexity does not hold. What minimizing challenges arise for functions which are not convex?

$$p(v) = (1 - v_1)^2 + 100(v_2 - v_1^2)^2$$

$$= 1 + v_1^2 - 2v_1 + 100v_2^2 + 100v_1^4 - 200v_2v_1^2$$

$$= 100v_1^4 + 1 - 2v_1 + 100v_2^2 - 200v_2v_1^2 + v_1^2$$

$$\text{Gradient of } p(v) = \begin{bmatrix} \frac{dp(v)}{dv_1} \\ \frac{dp(v)}{dv_2} \end{bmatrix}$$

$$\frac{d(p(v))}{dv_1} = 400v_1(v_1^2 - v_2) + 2v_1 - 2$$

$$\frac{d(p(v))}{dv_2} = 200v_2 - 200v_1^2$$

$$\therefore \text{Gradient of } p(v) = \begin{bmatrix} 400 v_1 (v_1^2 - v_2) + 2(v_1 - 1) \\ 200 (v_2 - v_1^2) \end{bmatrix}$$

$$\frac{d^2(p(v))}{dv_1^2} = \frac{d(400 v_1 (v_1^2 - v_2) + 2v_1 - 2)}{dv_1}$$

$$= -400(v_2 - v_1^2) + 800 v_1^2 + 2$$

$$\frac{d^2(p(v))}{dv_1 . dv_2} = \frac{d(400 v_1 (v_1^2 - v_2) + 2v_1 - 2}{dv_2}$$

$$= -400 v_1$$

$$\frac{d^2(p(v))}{dv_2 . dv_1} = -400 v_1$$

$$\frac{d^2(p(v))}{dv_2^2} = \frac{d(200 v_2 - 200 v_1^2)}{dv_2} = 200$$

Hessian matrix of $p(v) = \begin{bmatrix} -400(v_2 - v_1^2) + 800v_1^2 + 2 & -400v_1 \\ -400v_1 & 200 \end{bmatrix}$

Eigenvalues of Hessian matrix are,

$$\lambda_1 = 600v_1^2 - \sqrt{360000v_1^4 - 240000v_1^2v_2 + 41200v_1^2 + 40000v_2^2 + 39600v_2 - 200v_2 + 101}$$

$$\lambda_2 = 600v_1^2 + \sqrt{360000v_1^4 - 240000v_1^2v_2 + 41200v_2^2 + 39600v_2 + 9801 - 200v_2 + 101}$$

Problem 3: Nelder-Mead Search and Gradient Descent

The Nelder-Mead Search is a direct search (no gradient) method to find optimizers for functions. It has implementations both in R and Python. Also, you will consider a conjugate gradient descent algorithm (CG), common to both language libraries. In this problem you will compare the performance of both algorithms for function $p(\mathbf{v})$ above. Note that $p(\mathbf{v})$ has a unique global minimum at $\mathbf{v} = [1, 1]^T$.

1. Please research the Nelder-Mead Algorithm (NMA) using Wiki and other sources. Explain how it works in 2 dimensions. In particular, what is a simplex in 2 dimensions, and how does NMA evolve the simplex on each iteration? Please provide a brief description of the algorithm, using a graphic if helpful. Do not provide a listing of an implementation.

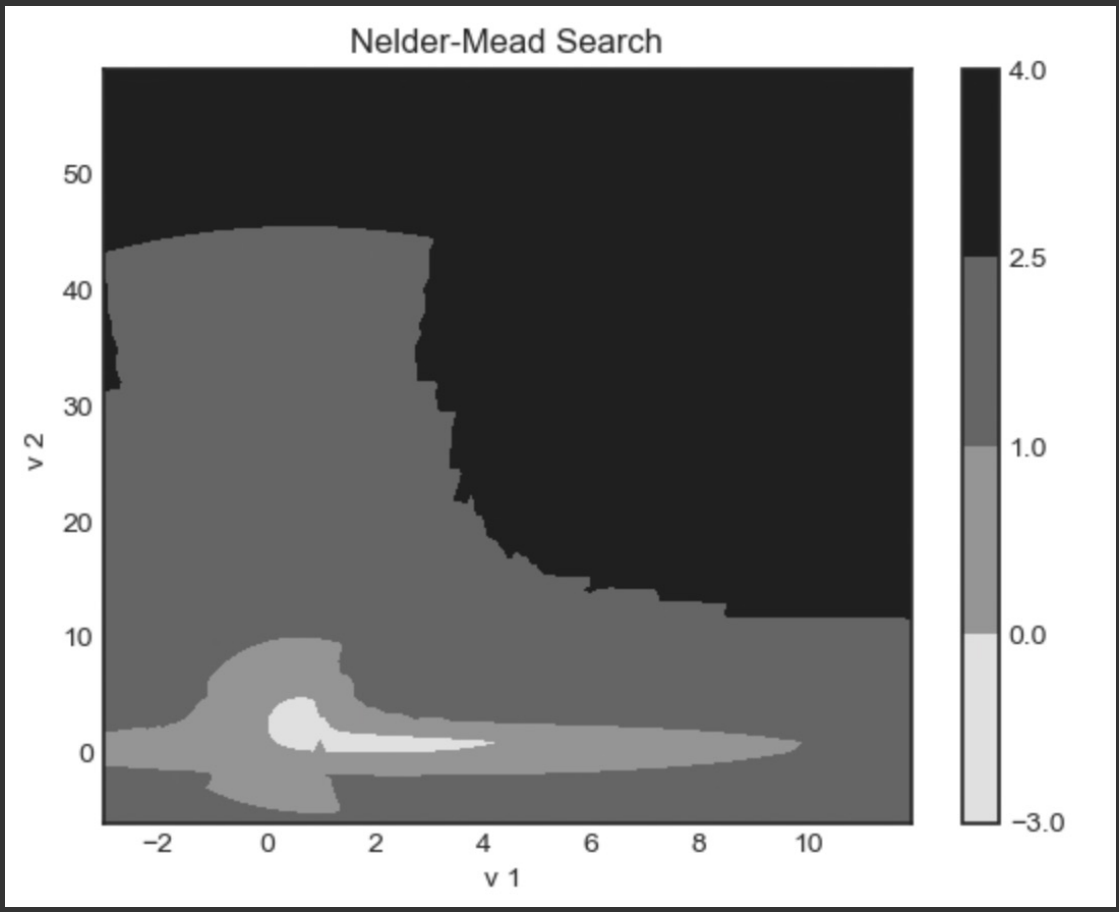
2. Run NMA and CG to search for the minimizer of p, as follows. Set the maximum number of iterations to 5 for both algorithms. Consider starting guesses on the grid $v_1 \in [-3, 12]$ (spacing 0.1) and $v_2 \in [-6, 60]$ (spacing 1). At the end of each trial record the squared distance between the output guess $\mathbf{v}_g$ and the optimizer $[1, 1]^T$ for each algorithm. Present your results in a 5-level contour map for each algorithm, with the log10(squared distance) as the altitude versus the starting position. Discuss your findings.

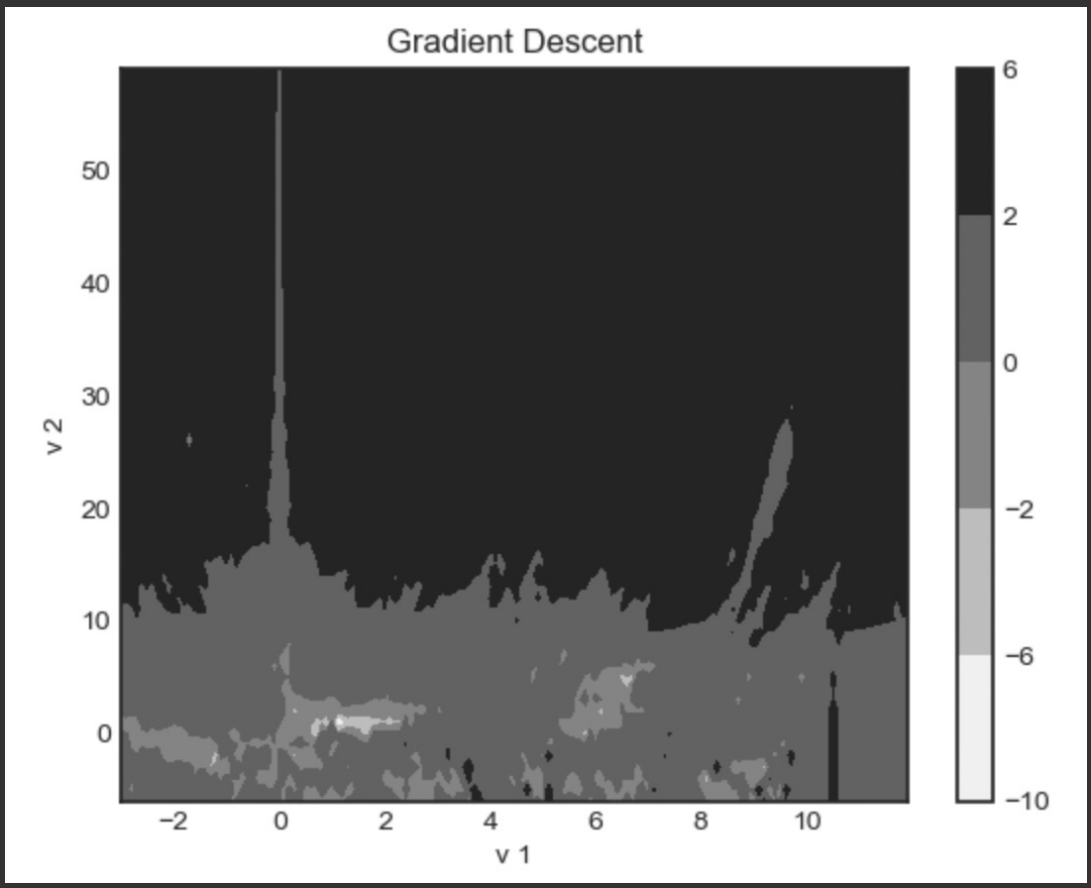
⇒ CODE:

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-white')
import numpy as np
from scipy.optimize import minimize
import math

def obj_func(v):
    v1=v[0]
    v2=v[1]
    return ((1-v1)**2)+100*((v2-v1**2)**2)


def dist(X,Y):
    return (math.hypot(X - 1, Y - 1))**2

def nelderMead(v1,v2):
    v0=[v1,v2]
    res = minimize(obj_func, v0, method='nelder-mead',options={'maxiter': 5,'xtol': 1e-8, 'disp': True})
    if res.fun==0.0:
        return 0
    else:
        return math.log10(dist(res.x[0],res.x[1]))

def cg(v1,v2):
    v0=[v1,v2]
    res1= minimize(obj_func, v0, method='CG', options={'maxiter': 5,'xtol': 1e-8, 'disp': True})
    if res1.fun==0:
        return 0
    else:
        return math.log10(dist(res1.x[0],res1.x[1]))

def graph():
    x=np.arange(-3,12,0.1)
    y=np.arange(-6,60,1)
    x1 = [round(num, 1) for num in x]
    Z=np.ndarray((len(y),len(x1)))
    Z1=np.ndarray((len(y),len(x1)))
    for i in range(0,len(x1)):
        for j in range(0,len(y)):
            Z[j][i]=nelderMead(x1[i],y[j])
            Z1[j][i]=cg(x1[i],y[j])
    levels=[-3.0,0,1.0,2.5,4.0]
```

```
    fig,ax=plt.subplots(1,1)
    cp = ax.contourf(x1, y, Z,levels)
    fig.colorbar(cp)
    ax.set_title('Nelder-Mead Search')
    ax.set_xlabel('v 1 ')
    ax.set_ylabel('v 2 ')
    plt.show()
    levels1=[-10,-6,-2,0,2,6]
    fig,ax=plt.subplots(1,1)
    cp = ax.contourf(x1, y, Z1,levels1)
    fig.colorbar(cp)
    ax.set_title('Gradient Descent')
    ax.set_xlabel('v 1')
    ax.set_ylabel('v 2')
    plt.show()

graph()
```

⇒ OUTPUT:





CG is faster and more precise than NMA as we can see through maps. The distance between the minima and point after iterations are very distributed in NMA whereas CG is close to minima

Nelder-Neas Algorithm is a simplex method for finding a local minimum of a function of several variables has been devised by Nelder and Mead. For two variables, a simplex is a triangle, and the method is a pattern search that compares function values at the three vertices of a triangle. The worst vertex, where f(x,y) is largest, is rejected and replaced with a new vertex. A new triangle is formed and the search is continued. The process generates a sequence of triangles (which might have different shapes), for which the function values at the vertices get smaller and smaller. The size of the triangles is reduced and the coordinates of the minimum point are found.

The algorithm is stated using the term simplex (a generalized triangle in N dimensions) and will find the minimum of a function of N variables. It is effective and computationally compact.

Initial Triangle BGW

Let f(x,y) be the function that is to be minimized. To start, we are given three vertices of a triangle: v[k]=(x[k],y[k]) , k=1,2,3 . The function f(x,y) is then evaluated at each of the three points: z[k]= f(x[k],y[k]) for k=1,2,3 . The subscripts are then reordered so that z1<=z2<=z3 . We use the notation

B=(x1,y1), G=(x2,y2), W=(x3,y3)
to help remember that B is the best vertex, G is good (next to best), and W is the worst vertex.

For two variables, a simplex is a triangle, and the method is a pattern search that compares function values at the three vertices of a triangle. The worst vertex, where f(x,y) is largest, is rejected and replaced with a new vertex. A new triangle is formed and the search is continued. The process generates a sequence of triangles (which might have different shapes), for which the function values at the vertices get smaller and smaller. The size of the triangles is reduced and the coordinates of the minimum point are found.

The algorithm is stated using the term simplex (a generalized triangle in N dimensions) and will find the minimum of a function of N variables. It is effective and computationally compact.

Midpoint of the Good Side

The construction process uses the midpoint of the line segment joining B and G . It is found by averaging the coordinates:

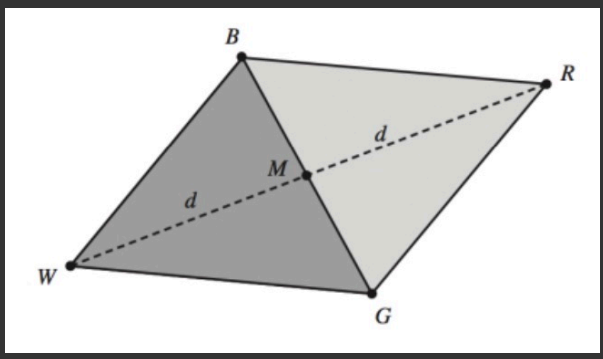
M = (B+G)/2 = ((x1+x2)/2,(y1+y2)/2).

Reflection Using the Point R

The function decreases as we move along the side of the triangle from W to B, and it decreases as we move along the side from W to G. Hence it is feasible that f(x,y) takes on smaller values at points that lie away from W on the opposite side of the line between B and G. We choose a test point R that is obtained by “reflecting”

the triangle through the side . To determine R, we first find the midpoint M of the side . Then draw the line segment from W to M and call its length d. This last segment is extended a
distance d through M to locate the point R .The vector formula for R is

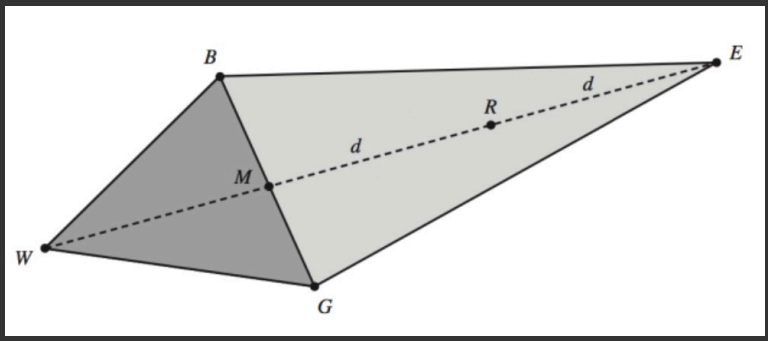
R=M+(M-W) = 2M-W.



Expansion Using the Point E

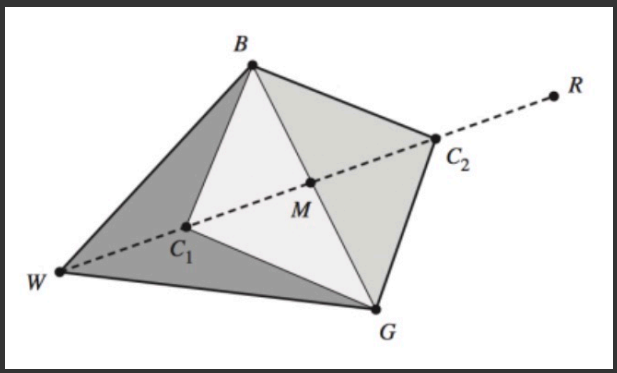
If the function value at R is smaller than the function value at W , then we have moved in the correct direction toward the minimum. Perhaps the minimum is just a bit farther than the point R. So we extend the line segment through M and R to the point E. This forms an expanded triangle BGE. The point E is found by moving an additional distance d along the line joining M and R .If the function value at E is less than the function value at R, then we have found a better vertex than R. The vector formula for E is

$$E=R+(R-M)= 2R-M.$$



Contraction Using the Point C

If the function values at R and W are the same, another point must be tested. Perhaps the function is smaller at M, but we cannot replace W with M because we must have a triangle. Consider the two midpoints C1 and C2 of the line segments and , respectively. The point with the smaller function value is called C, and the new triangle is BGC. Note. The choice between C1 and C2 might seem inappropriate for the two-dimensional case, but it is important in higher dimensions.



Shrink toward B

If the function value at C is not less than the value at W, the points G and W must be shrunk toward B.The point G is replaced with M, and W is replaced with S, which is the midpoint of the line segment joining B with W.

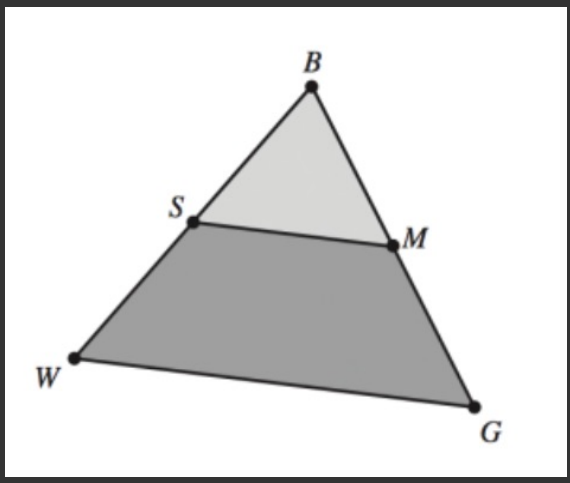


Fig: Shrinking the triangle towards B

The Nelder-Mead method uses a geometrical shape called a simplex as its ‘vehicle’ of sorts to search the domain. This is why the technique is also called the Simplex search method. In layman’s terms, a simplex is the n-dimensional version of a ‘triangle’.