Welcome to C277B: Machine Learning Algorithms

Homework assignment #1: Local Optimization Methods

Student Name: Charis Liao

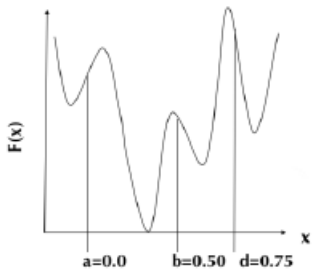
In this jupyter notebook, you will find answers to each problem in homework 1 of the Machine Learning Algorithm course. Concepts covered includes:

- Bisection
- Golden Section
- Local optimization using 1st and quasi-2nd order methods
- Steepest Descent
- Congugate Gradient
- BFGS
- Stochastic Gradient
- Stochastic Gradient Descent with Momentum (SGDM)

Due Date: Feb 7, 2023

1. Bisection vs. Golden Section

In class we used the simple bisection method to take the first step in isolating at least one minimum for the function shown. This first step in placement of `d` reduced the original interval `[a,b,c] = 1.0` to `[a,b,d] = 0.75`. But in general. the average size interval `<L>` after Step 1 is determined by the equal probability of placing point d in either sub-interval, such that `<L1>` = P(left-interval) x `1/2` + P(right-interval)x `3/4` = 0.625 (since we can't a `priori` know the best half)



(a) For step 2, place point `e` at the bisector of larger interval `[a,b]`. Why is this better than `[b,d]`?

A larger interval has a higher probability of containing a minimum.

(b) What is the new interval and how much is the search space reduced?

If `e < a` and `e < b`, then the new interval is `[a, e, b]`. The search space is reduced by `1/3`, so the new search area is `2/3` of the original search space.

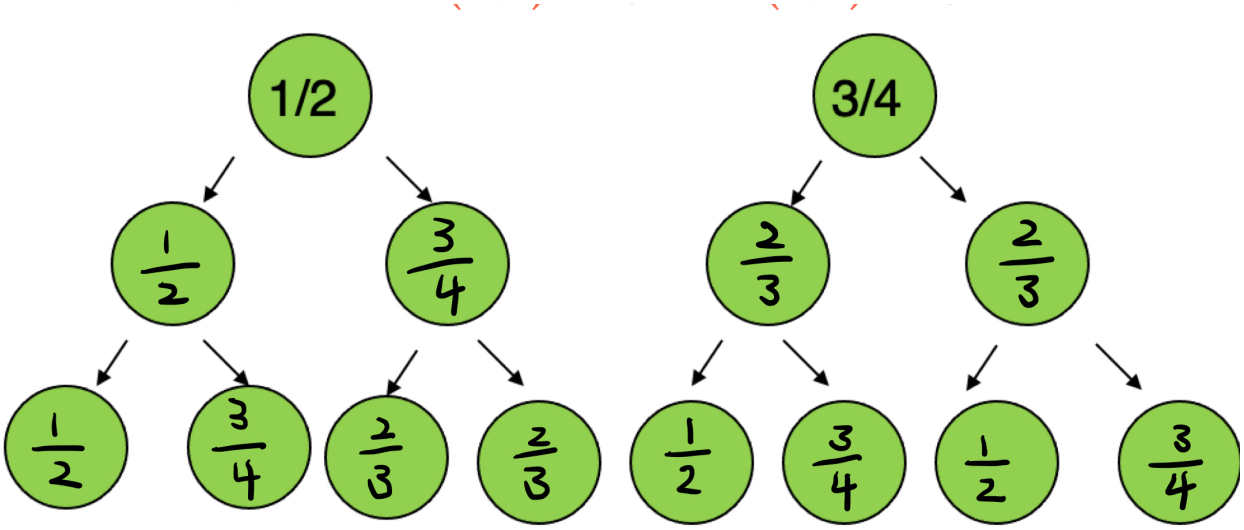
(c) For step 3, reduce the size of the interval from step 2 by placing point `f` at the bisection of your choice.

`f` was placed in between a and e.

(d) Fill in the tree for all possible size intervals for steps 2 and 3. Write your answers in ratios to the interval size of the previous steps.

My answers will be from top to bottom and left to right

Step 1: 1/2 and 3/4.
Step 2: 1/2, 3/4, 2/3, 2/3.
Step 3: 1/2, 3/4, 2/3, 2/3, 1/2, 3/4, 1/2, 3/4.



In [1]:
```
step3 = 0.125 *((0.5**3)+(0.5**2 * (3/4))+((3/4)*(2/3)*2*0.5)+((3/4)*(2/3)*(1+2 * (3/4))))
step3
```

Out[1]:
```
0.2578125
```

(e) What is average size of interval at steps 2 and 3?

Step 2 New interval size:

$$P(0.25) \times \frac{1}{2} \times \frac{1}{2} + P(0.25) \times \frac{3}{4} \times \frac{1}{2} + P(0.25) \times \frac{2}{3} \times \frac{3}{4} + P(0.25) \times \frac{2}{3} \times \frac{3}{4} = 0.46875$$

Step 3 New interval size:

$$P(0.125) \times \frac{1}{2} \times \frac{1}{2} + P(0.125) \times \frac{3}{4} \times \frac{1}{2} + P(0.125) \times \frac{2}{3} \times \frac{3}{4} + P(0.125) \times \frac{2}{3} \times \frac{3}{4} + P(0.125) \times \frac{1}{2} \times \frac{2}{3} + P(0.125$$

(f) How much does Golden Section improve over Bisection at each step? Please use a chart of steps v.s. different methods to show their difference.

Avg. Step Size	Golden Section	Bisection
Step 1	0.618	0.6250
Step 2	0.382	0.4688
Step 3	0.236	0.2578

2. Local optimization using 1st and quasi-2nd order methods

You will solve following optimization problem using a python code you develop for the steepest descents method! For the function

$$f(x, y) = x^4 - x^2 + y^2 + 2xy - 2$$

there are three stationary points found over the range x = [2,2] and y = [-2, 2].

In [2]:
```
# A timing decorator.
import time

def timeit(f):

    def timed(*args, **kw):
```

```
        ts = time.time()
        result = f(*args, **kw)
        te = time.time()

        print('func:%r took: %2.4f sec' % (f.__name__,  te-ts))
        return result

    return timed
```

(a) Starting from point `(1.5, 1.5)`, and with stepsize = 0.1, determine new (x, y) position using one step of the steepest descent algorithm (check against the debugging output). Is it a good optimization step? Depending on this this outcome, how will you change the stepsize in the next step?

In [3]:
```
from pylab import *
import numpy.linalg as LA
import numpy as np
def function(point):
    x = point[0]
    y = point[1]
    return x**4 - x**2 + y**2 + 2*x*y - 2

def first_deriv(point):
    x = point[0]
    y = point[1]
    return np.array([4*x**3 - 2*x + 2*y , 2*x + 2*y])


def new_position(func, first_derivative, starting_point, stepsize):
    new_point = starting_point - (stepsize * first_derivative(starting_point))

    print(f'new point: {new_point} , starting point: {starting_point}')
    print(f'new output: {func(new_point)} , starting output: {func(starting_point)}')
```

In [4]:
```
new_position(function, first_deriv, np.array([1.5, 1.5]), 0.1)
```

```
new point: [0.15 0.9 ] , starting point: [1.5 1.5]
new output: -0.9419937500000004 , starting output: 7.5625
```

It is a good optimization step because the new output is less than the starting output. Since the outcome is desired, we shall increase the stepsize by multiplying 1.2.

(b) Implement the steepest decent using the provided template. Continue execting steepest descent. How many steps does it take to converge to the local minimum to tolernace = 1×10^{-5} of the gradient (check again debugging output and compare code timings)?

Note: You don't need to use line seach, just take one step in the search direction, and use the following stepsize update:

$$\lambda = \begin{cases} 1.2\lambda & \text{for a good step} \\ 0.5\lambda & \text{for a bad step} \end{cases}$$

In [5]:
```
@timeit
def steepest_descent(func,first_derivate,starting_point,stepsize,tol):
    # evaluate the gradient at starting point

    count=0
    visited=[starting_point]
    deriv = first_derivate(starting_point)
    new_point = starting_point

    while LA.norm(deriv) > tol and count < 1e6:
        # calculate new point position

        prev = new_point
        new_point = prev - (stepsize * first_derivate(prev))
        deriv = first_derivate(new_point)
        visited.append(new_point)


        if func(new_point) < func(prev):
```

```
            # the step makes function evaluation lower - it is a good step. what do you do?
            stepsize = 1.2 * stepsize

        else:
            # the step makes function evaluation higher - it is a bad step. what do you do?
            stepsize = 0.5 * stepsize
        count+=1
    # return the results
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.asarray(visited), "count": c
```

In [6]:
```
steepest_descent(function, first_deriv, np.array([1.5, 1.5]), 0.1, 1e-5)
```

```
func:'steepest_descent' took: 0.0009 sec
```

Out[6]:
```
{'x': array([1.5, 1.5]),
 'evaluation': 7.5625,
 'path': array([[ 1.5       ,  1.5       ],
        [ 0.15      ,  0.9       ],
        [-0.03162   ,  0.648     ],
        [-0.22733235,  0.47048256],
        [-0.4603766 ,  0.38644985],
        [-0.73063964,  0.41710875],
        [-0.91361447,  0.57314178],
        [-0.89067253,  0.77647099],
        [-1.072703  ,  0.85831194],
        [-0.88004038,  0.93513214],
        [-1.07440969,  0.91144369],
        [-0.8191814 ,  0.99553056],
        [-1.00371455,  0.95003442],
        [-0.98247032,  0.96665308],
        [-1.00196259,  0.97252925],
        [-0.98533102,  0.98565078],
        [-1.0162044 ,  0.98547972],
        [-0.99022477,  0.99369805],
        [-1.00370679,  0.9925832 ],
        [-0.9936794 ,  0.99686773],
        [-1.00672832,  0.99539405],
        [-0.99782619,  0.99801346],
        [-1.00028169,  0.99796153],
        [-0.99913443,  0.99873366],
        [-1.00035525,  0.9988937 ],
        [-0.99897351,  0.99959411],
        [-1.00010453,  0.99944541],
        [-0.99979477,  0.99963492],
        [-1.00002278,  0.99969008],
        [-0.9998473 ,  0.99982783],
        [-1.00014104,  0.9998375 ],
        [-0.99992545,  0.99991291],
        [-1.0000106 ,  0.99991665],
        [-0.99996182,  0.99995026],
        [-1.00002241,  0.99995522],
        [-0.99998875,  0.99996964],
        [-0.99999542,  0.99997456],
        [-0.99999464,  0.99998101],
        [-0.99999754,  0.99998606],
        [-0.99999681,  0.99999117],
        [-1.00000061,  0.99999418],
        [-0.99999493,  0.9999983 ],
        [-1.00000251,  0.99999722],
        [-0.99999661,  0.99999926],
        [-1.00000408,  0.99999804],
        [-0.99999892,  0.99999943]]),
 'count': 45}
```

It took 45 steps to converge to the local minimum.

(c) Compare your steepest descent code against conjugate gradients (CG), and BFGS to determine the local minimum starting from $(1.5, 1.5)$. In terms of number of steps, are conjugate gradients and/or BFGS more efficient than steepest descents?

Note: See SciPy documentation on how to use CG and BFGS, examples available at the end of webpage:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.minimize.html

In [7]:
```
from scipy.optimize import minimize
x0 = np.array([1.5, 1.5])

minimize(function, x0, method = "CG", options={'disp': True, 'gtol': 1e-5})
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 9
         Function evaluations: 78
         Gradient evaluations: 26
```

Out[7]:
```
     fun: -2.99999999999959
     jac: array([ 2.08616257e-07, -1.10268593e-06])
 message: 'Optimization terminated successfully.'
    nfev: 78
     nit: 9
    njev: 26
  status: 0
 success: True
       x: array([-0.99999984,  0.99999929])
```

In [8]:
```
minimize(function, x0, method = "BFGS", options={'disp': True, 'gtol': 1e-5})
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 7
         Function evaluations: 24
         Gradient evaluations: 8
```

Out[8]:
```
      fun: -2.9999999999998255
 hess_inv: array([[ 0.12457729, -0.12457659],
       [-0.12457659,  0.62569812]])
      jac: array([-1.63912773e-06, -2.98023224e-08])
  message: 'Optimization terminated successfully.'
     nfev: 24
      nit: 7
     njev: 8
   status: 0
  success: True
        x: array([ 0.99999979, -0.9999998 ])
```

In terms of number of steps, both CG and BFGS is more efficient with a step count of 7.

3. Local optimization and machine learning using Stochastic Gradient Descent (SGD).

The Rosenbrock Banana Function looks innocuous enough
$f(x, y) = (1 - x)^2 + 10(y - x^2)^2$
with only one (global) minumum at $(x, y) = (1.0, 1.0)$!

(a) Starting at $x = -0.5$ and $y = 1.5$, and using your code for steepest descents with stepsize = 0.1, how many steps to converge to the minimum? Use a tolerance = 1×10^{-5}

In [9]:
```
def RosenbrockBanana(point):
    x = point[0]
    y = point[1]
    return (1 - x)**2 + 10 * (y - x**2)**2

def RosenbrockBanana_Deriv(point):
    x = point[0]
    y = point[1]
    return np.array([-2 * (1-x) - 40*x*(y-x**2), 20*(y-x**2)])
```

In [10]:
```
steepest_descent(RosenbrockBanana, RosenbrockBanana_Deriv, np.array([-0.5, 1.5]), 0.01, 1e-5)
```

```
func:'steepest_descent' took: 0.0228 sec
```

Out[10]:
```
{'x': array([-0.5,  1.5]),
 'evaluation': 17.875,
 'path': array([[-0.5       ,  1.5       ],
        [-0.72      ,  1.25      ],
        [-0.93156096,  1.074416  ],
        ...,
        [ 0.99999058,  0.99998155],
        [ 0.99999099,  0.99998145],
        [ 0.99999095,  0.9999816 ]]),
 'count': 1165}
```

From the result above, 8 steps seemed to be the answer. However, I noticed that within 8 steps, the minimum is converging to `nan` and `-inf` which seemed odd. Therefore, I would say that this is not converging. If we change the step size to 0.01, then it will converge.

(b) By adding a small amount of stochastic noise to the gradient at every step (In your code add a random vector that is the same norm as the gradient at that step), which is equivalent to a small batch derivative of any loss function in deep learning, implement your own stochastic gradient descent code by modifying on your steepest descent code, and run SGD algorithm. (Check against debugging outputs.)

In [11]:
```
@timeit
def stochastic_gradient_descent(func,first_derivate,starting_point,stepsize,tol=1e-5,stochastic_injecti
    '''stochastic_injection: controls the magnitude of stochasticity (multiplied with stochastic_deriv)
        0 for no stochasticity, equivalent to SD.
        Use 1 in this homework to run SGD
    '''
    # evaluate the gradient at starting point

    count=0
    visited=[starting_point]
    deriv = first_derivate(starting_point)
    new_point = starting_point
    while LA.norm(deriv) > tol and count < 1e5:


        if stochastic_injection>0:
            # formulate a stochastic_deriv that is the same norm as your gradient
            random_vector = np.random.normal(0,1,2)
            stochastic_deriv =  random_vector/LA.norm(random_vector)*LA.norm(deriv)

        else:
            stochastic_deriv=np.zeros(len(starting_point))
        direction=-(deriv+stochastic_injection*stochastic_deriv)


        # calculate new point position
        prev = new_point
        new_point = prev + (stepsize*direction)
        deriv = first_derivate(new_point)
        visited.append(new_point)

        if func(new_point) < func(prev):
            # the step makes function evaluation lower - it is a good step. what do you do?
            stepsize = 1.2 * stepsize

        else:
            # the step makes function evaluation higher - it is a bad step. what do you do?
            stepsize = 0.5 * stepsize
        count+=1
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.asarray(visited), "count": c
```

In [12]:
```
stochastic_gradient_descent(RosenbrockBanana,RosenbrockBanana_Deriv,np.array([-0.5, 1.5]),0.01,tol=1e-5
```

```
func:'stochastic_gradient_descent' took: 0.0696 sec
```

Out[12]:
```
{'x': array([-0.5,  1.5]),
 'evaluation': 17.875,
 'path': array([[-0.5       ,  1.5       ],
       [-0.51009564,  1.50853464],
       [-0.9383711 ,  1.57920721],
       ...,
       [ 0.99998894,  0.99997777],
       [ 0.99998925,  0.99997799],
       [ 0.99998934,  0.99997826]]),
 'count': 2015}
```

(c) Evaluate how much better or worse is the SGD convergence against the CG and BFGS method to find the global minimum, in terms of number of steps. Converge function/gradient to tolerance = 1×10^{-5}

minimize(RosenbrockBanana, np.array([-0.5, 1.5]), method = "CG", options={'disp': True, 'gtol': 1e-5})

In [13]:
```
minimize(RosenbrockBanana, np.array([-0.5, 1.5]), method = "BFGS", options={'disp': True, 'gtol': 1e-5})
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 22
         Function evaluations: 93
         Gradient evaluations: 31
```

Out[13]:
```
      fun: 1.6856836004019217e-13
 hess_inv: array([[0.50988602, 1.01962714],
       [1.01962714, 2.08896666]])
      jac: array([ 1.15312325e-07, -1.29424893e-08])
  message: 'Optimization terminated successfully.'
     nfev: 93
      nit: 22
     njev: 31
   status: 0
  success: True
        x: array([0.99999959, 0.99999917])
```

CG and BFGS are better than SDG in terms of how many iterations they've all gone through before they converged.

number of steps	SGD	CG	BFGS
	2003	22	20

(d) Can you draw a firm conclusion on the outcome with just one run of each method? If not, explain why.

Yes, I can draw a firm conclusion with just one run of each method because they all converged. As long as they converged, there's at least one local minimum.

(e) Run all the algorithms multiple times starting at different (x, y) positions to understand the average performance of each. Explain the relative performance of the non-stochastic and stochastic methods on the Rosenbrock Banana Function.

Steepest Descent

In [14]:
```
def avg_performance_SD(iterations, starting_point, function, first_deriv):
    count = 0
    for i in range(iterations):
        count += steepest_descent(function, first_deriv, starting_point, 0.01, 1e-5)['count']
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} steps")
```

In [15]:
```
avg_performance_SD(10, np.array([1.5, 1.5]), RosenbrockBanana, RosenbrockBanana_Deriv)
```

```
func:'steepest_descent' took: 0.0255 sec
func:'steepest_descent' took: 0.0186 sec
func:'steepest_descent' took: 0.0164 sec
func:'steepest_descent' took: 0.0189 sec
func:'steepest_descent' took: 0.0200 sec
func:'steepest_descent' took: 0.0167 sec
func:'steepest_descent' took: 0.0160 sec
func:'steepest_descent' took: 0.0166 sec
func:'steepest_descent' took: 0.0174 sec
func:'steepest_descent' took: 0.0195 sec
The average performance (step counts) is 1203.0 steps
```

Stochastic Gradient Descent (SGD)

In [16]:
```
def avg_performance_SGD(iterations, starting_point, function, first_deriv):
    count = 0
    for i in range(iterations):
        count += stochastic_gradient_descent(function, first_deriv, starting_point,0.01,tol=1e-5,stocha
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} steps")
```

In [17]:
```
avg_performance_SGD(10, np.array([1.5, 1.5]), RosenbrockBanana, RosenbrockBanana_Deriv)
```

```
func:'stochastic_gradient_descent' took: 0.0832 sec
func:'stochastic_gradient_descent' took: 0.0724 sec
func:'stochastic_gradient_descent' took: 0.0836 sec
func:'stochastic_gradient_descent' took: 0.0440 sec
func:'stochastic_gradient_descent' took: 0.0732 sec
func:'stochastic_gradient_descent' took: 0.0521 sec
func:'stochastic_gradient_descent' took: 0.0901 sec
func:'stochastic_gradient_descent' took: 0.0584 sec
func:'stochastic_gradient_descent' took: 0.0675 sec
func:'stochastic_gradient_descent' took: 0.0482 sec
The average performance (step counts) is 2255.1 steps
```

CG

In [18]:
```
def avg_performance_CG(iterations, starting_point, function):
    count = 0
    for i in range(iterations):
        count += minimize(function, starting_point, method = "CG", options={'disp': True, 'gtol': 1e-5}
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} steps")
```

In [19]:
```
avg_performance_CG(10, np.array([1.5, 1.5]), RosenbrockBanana)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 79
         Gradient evaluations: 26
The average performance (step counts) is 10.0 steps
```

BFGS

In [20]:
```
def avg_performance_BFGS(iterations, starting_point, function):
    count = 0
    for i in range(iterations):
        count += minimize(function, starting_point, method = "BFGS", options={'disp': True, 'gtol': 1e-
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} steps")
```

In [21]:
```
avg_performance_BFGS(10, np.array([1.5, 1.5]), RosenbrockBanana)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 10
         Function evaluations: 36
         Gradient evaluations: 12
The average performance (step counts) is 10.0 steps
```

Average Performance	SD	SGD	CG	BFGS
Number of steps	1203	1948	10	10

Apparently, Stochastic Gradient Descent takes more average steps than Steepest Descent. This might due to the fact that the gradient for SGD is not computed for the entire dataset, and only for one random point on each iteration, the updates have higher variance. Higher vairance might cause the algorithm to take more steps to reach the minimum.

4. Stochastic Gradient Descent with Momentum (SGDM).

The Rosenbrock Banana Function with one minimum is not the best way to illustrate the power of the SGD or SGDM method. Hence we next investigate the Three-Hump Camel function.
$f(x,y) = 2x^2 - 1.05x^4 + \frac{x^6}{6} + xy + y^2 \; x \in [-2,2], y \in [-2,2].$

which is a convex function with three minima. This defines our first "multiple minima" problem where there is a global solution as well as two less optimal solutions.

(a) Utilize SGD to find the *global minimum*, and compare it to CG or BFGS as you did in (2e). Starting from [-1.5, -1.5], converge function and gradient to tolerance = 1×10^{-5} with stepsize = 0.1. On average, did you get a better result in finding the global minimum with SGD in terms of fewer steps on average?

In [22]:
```
def threeHumpCamel(point):
    x = point[0]
    y = point[1]
    return 2*x**2 - 1.05*x**4 +(x**6/6) + x*y + y**2

def first_deriv_THC(point):
    x = point[0]
    y = point[1]
    return np.array([x**5 + 4.2*x**3 + 4*x + y, x+2*y])
```

In [23]:
```
stochastic_gradient_descent(threeHumpCamel,first_deriv_THC,np.array([-1.5, 1.5]),0.1,tol=1e-5,stochasti
```

```
func:'stochastic_gradient_descent' took: 0.0023 sec
```

Out[23]:
```
{'x': array([-1.5,  1.5]),
 'evaluation': 1.0828125000000002,
 'path': array([[-1.50000000e+00,  1.50000000e+00],
       [-8.44936181e-01,  3.09210589e+00],
       [-8.34647187e-01,  2.55217772e+00],
       [-5.90334041e-01,  2.63135630e+00],
       [-5.37768135e-01,  2.62896928e+00],
       [-4.87412964e-01,  2.50515638e+00],
       [-5.65892470e-01,  2.44796395e+00],
       [-6.31863766e-01,  2.40529178e+00],
       [-7.06316561e-01,  2.33217389e+00],
       [-7.13354666e-01,  2.09004283e+00],
       [-5.99174113e-01,  1.80322375e+00],
       [-3.77601950e-01,  1.65141330e+00],
       [-3.93818440e-01,  1.65005670e+00],
       [-2.36042922e-01,  1.58150100e+00],
       [-4.59869539e-01,  1.50112483e+00],
       [-1.44240035e-01,  1.29853119e+00],
       [-1.42723025e-01,  1.29808818e+00],
       [-1.73006632e-01,  1.30345351e+00],
       [-2.10187603e-01,  9.73248790e-01],
       [-2.82538580e-01,  7.10391080e-01],
       [-1.79689307e-01,  4.92794665e-01],
       [-6.38626046e-02,  4.43966749e-01],
       [-1.47045319e-01,  4.31979722e-01],
       [-5.89387478e-02,  4.19652065e-01],
       [-1.25881965e-01,  4.20968103e-01],
       [ 5.05397745e-02,  1.71909431e-01],
       [-7.11392953e-02,  2.14289094e-01],
       [ 4.74177592e-02,  1.74787846e-01],
       [ 6.97198166e-02,  6.90781941e-02],
       [ 1.98059400e-03, -5.79963007e-02],
       [ 2.51863156e-02, -5.86738536e-02],
       [ 4.13513972e-02, -2.18501793e-02],
       [-2.88487413e-02,  2.69256309e-02],
       [ 3.73261236e-02, -1.15247851e-02],
       [ 1.32186996e-02, -4.38526317e-02],
       [-3.91538227e-03, -1.22483887e-02],
       [ 3.01600999e-03,  8.72609642e-03],
       [-4.18537643e-03,  1.19434477e-02],
       [-9.58159023e-04,  1.12356014e-02],
       [-7.66753817e-03,  6.34130545e-03],
       [-7.78774070e-03,  4.53603513e-03],
       [-6.55879809e-03,  8.55434604e-03],
       [-8.96693810e-04, -3.33841299e-03],
       [ 3.76636360e-03, -4.34614562e-03],
       [ 1.19661718e-03, -4.89338337e-04],
       [-1.08329004e-03, -6.92550238e-05],
       [ 1.77089608e-03, -2.70018781e-04],
       [-5.11133598e-04, -4.90552175e-04],
       [-5.69712577e-04, -3.22829370e-05],
       [-3.36619968e-04,  5.95297057e-04],
       [ 3.09470737e-05,  6.39138951e-04],
       [-6.76482846e-05,  2.34002225e-04],
       [-7.36829486e-05,  9.69089317e-05],
       [-8.05493075e-05,  7.48896342e-05],
       [ 7.05704855e-06, -3.08570821e-07],
       [-7.28589770e-07,  6.28648104e-06],
       [-6.24330568e-06,  1.18065362e-06],
       [-2.03537748e-06, -2.43090994e-06],
       [-1.63118838e-06,  1.00992001e-06]]),
 'count': 58}
```

In [24]:
```
minimize(threeHumpCamel, np.array([-1.5, 1.5]), method = "CG", options={'disp': True, 'gtol': 1e-5})
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 45
         Gradient evaluations: 15
```

Out[24]:
```
     fun: 0.2986384422415318
     jac: array([1.92224979e-06, 4.34368849e-06])
 message: 'Optimization terminated successfully.'
    nfev: 45
     nit: 8
    njev: 15
  status: 0
 success: True
       x: array([-1.74755237,  0.87377835])
```

In [25]:
```
minimize(threeHumpCamel, np.array([-1.5, 1.5]), method = "BFGS", options={'disp': True, 'gtol': 1e-5})
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 9
         Function evaluations: 33
         Gradient evaluations: 11
```

Out[25]:
```
      fun: 0.298638442236946
 hess_inv: array([[ 0.08662932, -0.04039265],
       [-0.04039265,  0.52808206]])
      jac: array([-1.11758709e-06, -4.47034836e-07])
  message: 'Optimization terminated successfully.'
     nfev: 33
      nit: 9
     njev: 11
   status: 0
  success: True
        x: array([-1.74755242,  0.87377597])
```

In [26]:
```
avg_performance_SGD(10, np.array([-1.5,-1.5]), threeHumpCamel, first_deriv_THC)
```

```
func:'stochastic_gradient_descent' took: 0.0116 sec
func:'stochastic_gradient_descent' took: 0.0058 sec
func:'stochastic_gradient_descent' took: 0.0019 sec
func:'stochastic_gradient_descent' took: 0.0027 sec
func:'stochastic_gradient_descent' took: 0.0026 sec
func:'stochastic_gradient_descent' took: 0.0013 sec
func:'stochastic_gradient_descent' took: 0.0029 sec
func:'stochastic_gradient_descent' took: 0.0017 sec
func:'stochastic_gradient_descent' took: 0.0029 sec
func:'stochastic_gradient_descent' took: 0.0022 sec
The average performance (step counts) is 74.4 steps
```

In [27]:
```
avg_performance_CG(10, np.array([1.5, 1.5]), threeHumpCamel)
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
The average performance (step counts) is 7.0 steps
```

In [28]:

```
avg_performance_BFGS(10, np.array([1.5, 1.5]), threeHumpCamel)
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
The average performance (step counts) is 8.0 steps
```

Average Performance	SDG	CG	BFGS
threeHumpCamel	68.7	7	8
RosenbrockBanana.	1948	10	10

Yes, on average, I did get a better result in finding the global minimum with SGD in terms of fewer steps.

(b) Implement the SGDM algorithm with momemtum $\gamma = 0.9$. Now use SGD with Momentum to find the global minimum. Again start from $[-1.5, -1.5]$ with stepsize = 0.1 and converge function and gradient to tolerance = 1×10^{-5}. On average, did you get a better result using SGDM compared to SGD, CG, or BFGS in finding the global minimum in terms of fewer steps?

In [33]:
```
@timeit
def SGDM(func,first_derivate,starting_point,stepsize,momentum=0.9,tol=1e-5,stochastic_injection=1):
    # evaluate the gradient at starting point


    count=0
    visited=[starting_point]
    deriv = first_derivate(starting_point)
```

```
    new_point = starting_point

    previous_direction = np.array([0,0])
    while LA.norm(deriv) > tol and count < 1e5:
        if stochastic_injection>0:
            # formulate a stochastic_deriv that is the same norm as your gradient
            random_vector = np.random.normal(0,1,2)
            stochastic_deriv =  random_vector/LA.norm(random_vector)*LA.norm(deriv)

        else:
            stochastic_deriv=np.zeros(len(starting_point))


        direction=-(deriv+stochastic_injection*stochastic_deriv) + (momentum*previous_direction)

        # calculate new point position
        prev = new_point
        new_point = prev + (stepsize*direction)
        deriv = first_derivate(new_point)
        visited.append(new_point)
        previous_direction = direction

        if func(new_point) < func(prev):
            # the step makes function evaluation lower - it is a good step. what do you do?
            stepsize = 1.2 * stepsize
        else:
            # the step makes function evaluation higher - it is a bad step. what do you do?
            # if stepsize is too small, clear previous direction because we already know that is not a
            stepsize = 0.5 * stepsize
            if stepsize<1e-5:
                previous_direction=previous_direction-previous_direction

            else:
                # do the same as SGD here
                previous_direction = new_point - prev
        count+=1
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.asarray(visited), 'count': c
```

In [39]:
```
SGDM(threeHumpCamel,first_deriv_THC,np.array([-1.5, -1.5]),0.1,momentum=0.9,tol=1e-5,stochastic_inject
```

```
func:'SGDM' took: 0.0031 sec
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:9: RuntimeWarning: over
flow encountered in double_scalars
  return np.array([x**5 + 4.2*x**3 + 4*x + y, x+2*y])
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:4: RuntimeWarning: over
flow encountered in double_scalars
  return 2*x**2 - 1.05*x**4 +(x**6/6) + x*y + y**2
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:9: RuntimeWarning: inva
lid value encountered in double_scalars
  return np.array([x**5 + 4.2*x**3 + 4*x + y, x+2*y])
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:4: RuntimeWarning: inva
lid value encountered in double_scalars
  return 2*x**2 - 1.05*x**4 +(x**6/6) + x*y + y**2
```

Out[39]:
```
{'x': array([-1.5, -1.5]),
 'evaluation': 5.5828125,
 'path': array([[-1.50000000e+00, -1.50000000e+00],
        [ 3.46638732e-01,  1.70720635e+00],
        [ 1.35300876e+00,  4.78140372e+00],
        [-8.55061189e-01,  2.83194497e+00],
        [-3.18920211e+00,  7.45882539e-04],
        [ 4.24268673e+00, -1.42072946e+01],
        [-1.90734112e+01,  1.57250799e+01],
        [ 3.06934502e+03, -1.14916479e+04],
        [-1.21093030e+15, -1.22577014e+15],
        [ 8.22019491e+72,  5.36121282e+72],
        [           -inf,             inf]]),
 'count': 10}
```

In [31]:
```
def avg_SDGM(iterations, starting_point, function, first_deriv):
    count = 0
    for i in range(iterations):
        count +=  SGDM(function,first_deriv,starting_point,0.1,momentum=0.9,tol=1e-5,stochastic_injecti
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} steps")
```

In [32]:
```
avg_SDGM(10, np.array([-1.5, -1.5]), threeHumpCamel, first_deriv_THC)
```

```
func:'SGDM' took: 0.0036 sec
func:'SGDM' took: 0.0009 sec
func:'SGDM' took: 0.0002 sec
func:'SGDM' took: 0.0193 sec
func:'SGDM' took: 0.0018 sec
func:'SGDM' took: 0.0125 sec
func:'SGDM' took: 0.0002 sec
func:'SGDM' took: 0.0010 sec
func:'SGDM' took: 0.0188 sec
func:'SGDM' took: 0.0002 sec
The average performance (step counts) is 131.4 steps
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:9: RuntimeWarning: over
flow encountered in double_scalars
  return np.array([x**5 + 4.2*x**3 + 4*x + y, x+2*y])
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:4: RuntimeWarning: over
flow encountered in double_scalars
  return 2*x**2 - 1.05*x**4 +(x**6/6) + x*y + y**2
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:4: RuntimeWarning: inva
lid value encountered in double_scalars
  return 2*x**2 - 1.05*x**4 +(x**6/6) + x*y + y**2
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/2216846355.py:9: RuntimeWarning: inva
lid value encountered in double_scalars
  return np.array([x**5 + 4.2*x**3 + 4*x + y, x+2*y])
/var/folders/8n/jmp5zkxn59j1qkmb_yk98xqw0000gn/T/ipykernel_42149/26002776.py:22: RuntimeWarning: inval
id value encountered in add
  direction=-(deriv+stochastic_injection*stochastic_deriv) + (momentum*previous_direction)
```

Average Performance	SGDM	SGD	CG	BFGS
	131.4	68.7	7	8

On average, I did not find a better result using SGDM compared to SGD, CG, or BFGS in finding a minimum in terms of fewer steps. However, it reaches global minimum more often for SGDM.

You've reached the end of HW1

In [ ]:
```
```