Welcome to C277B: Machine Learning Algorithms

Homework assignment #1: Local Optimization Methods

Student Name: Charis Liao

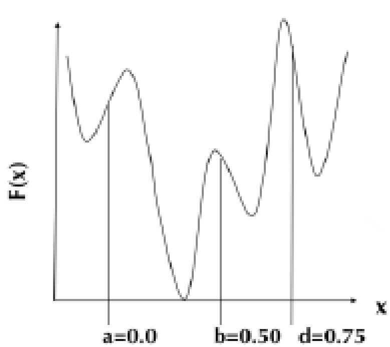
In this jupyter notebook, you will find answers to each problem in homework 1 of the Machine Learning Algorithm course. Concepts covered includes:

- 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.

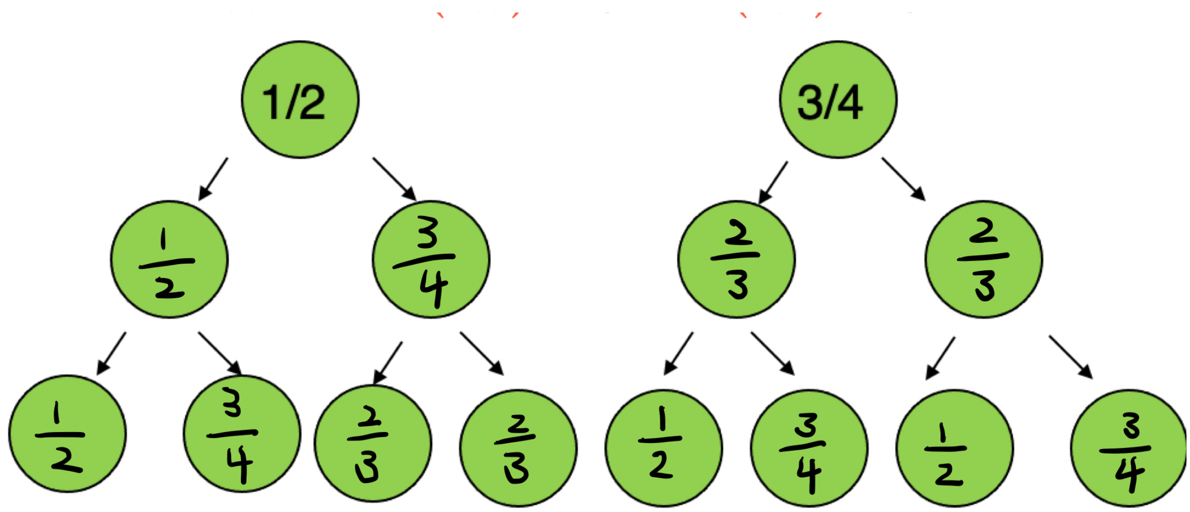
(d) Fill in the tree for all possible size intervals for steps 2 and 3. Write your answers in ratios to the interval size of the previous steps.

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
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:

$$P(0.125) \times \frac{1}{2} \times \frac{1}{2} + P(0.125) \times \frac{3}{4} \times \frac{1}{2} + P(0.125) \times \frac{2}{3} \times \frac{3}{4} + P(0.125) \times \frac{2}{3} \times \frac{3}{4} + P($$
$$+ P(0.125) \times \frac{1}{2} \times \frac{2}{3} + P(0.125) \times \frac{3}{4} \times \frac{2}{3} = 0.2578125$$

(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

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_poi
```

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:

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

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. wha
            stepsize = 1.2 * stepsize

        else:
            # the step makes function evaluation higher - it is a bad step. wha
            stepsize = 0.5 * stepsize
        count+=1
    # return the results
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.asar
```

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

```
func:'steepest_descent' took: 0.0012 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})
```

```
/opt/anaconda3/envs/msse-python/lib/python3.9/site-packages/scipy/__init__.py:
146: UserWarning: A NumPy version >=1.16.5 and <1.23.0 is required for this ve
rsion of SciPy (detected version 1.23.1
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
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 [13]:
```
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 [73]:
```
steepest_descent(RosenbrockBanana, RosenbrockBanana_Deriv, np.array([-0.5, 1.5]
```

```
func:'steepest_descent' took: 0.0223 sec
<ipython-input-13-fb626a120ae9>:9: RuntimeWarning: overflow encountered in dou
ble_scalars
  return np.array([-2 * (1-x) - 40*x*(y-x**2), 20*(y-x**2)])
<ipython-input-13-fb626a120ae9>:4: RuntimeWarning: overflow encountered in dou
ble_scalars
  return (1 - x)**2 + 10 * (y - x**2)**2
<ipython-input-5-99595690c15f>:14: RuntimeWarning: invalid value encountered i
n subtract
  new_point = prev - (stepsize * first_derivate(prev))
```

Out[73]:
```
{'x': array([-0.5,  1.5]),
 'evaluation': 17.875,
 'path': array([[-5.00000000e-001,  1.50000000e+000],
        [-2.70000000e+000, -1.00000000e+000],
        [ 4.24360000e+001,  7.29000000e+000],
        [-7.60696243e+004,  9.04052048e+002],
        [ 2.20091744e+014,  1.44664761e+009],
        [-2.66533168e+042,  6.05504695e+027],
        [ 2.36681219e+126,  4.43999561e+083],
        [            -inf,  1.75056248e+251],
        [             nan,              inf]]),
 'count': 8}
```

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 [42]:
```
@timeit
def stochastic_gradient_descent(func,first_derivate,starting_point,stepsize,tol
    '''stochastic_injection: controls the magnitude of stochasticity (multiplie
        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 gradie
            random_vector = np.random.normal(0,1,2)
            stochastic_deriv =  random_vector/LA.norm(random_vector)*LA.norm(de

        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. wha
            stepsize = 1.2 * stepsize

        else:
            # the step makes function evaluation higher - it is a bad step. wha
            stepsize = 0.5 * stepsize
        count+=1
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.asar
```

In [76]:
```
stochastic_gradient_descent(RosenbrockBanana,RosenbrockBanana_Deriv,np.array([-
```

```
func:'stochastic_gradient_descent' took: 0.0601 sec
```

Out[76]:
```
{'x': array([-0.5,  1.5]),
 'evaluation': 17.875,
 'path': array([[-0.5       ,  1.5       ],
        [ 0.4868934 , -0.03371305],
        [-0.08100894, -0.20768994],
        ...,
        [ 0.9999892 ,  0.9999774 ],
        [ 0.99998921,  0.99997741],
        [ 0.99998924,  0.99997804]]),
 'count': 2003}
```

(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}

In [44]:
```
minimize(RosenbrockBanana, np.array([-0.5, 1.5]), method = "CG", options={'disp
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 132
         Gradient evaluations: 44
```

Out[44]:
```
     fun: 2.0711814827200667e-13
     jac: array([ 4.94555024e-08, -2.45172016e-08])
 message: 'Optimization terminated successfully.'
    nfev: 132
     nit: 20
    njev: 44
  status: 0
 success: True
       x: array([0.99999955, 0.99999908])
```

In [45]:
```
minimize(RosenbrockBanana, np.array([-0.5, 1.5]), method = "BFGS", options={'di
```

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

Out[45]:
```
      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 [78]:
```
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, 
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} ste
```

In [79]:
```
avg_performance_SD(10, np.array([1.5, 1.5]), RosenbrockBanana, RosenbrockBanana
```

```
func:'steepest_descent' took: 0.0383 sec
func:'steepest_descent' took: 0.0191 sec
func:'steepest_descent' took: 0.0176 sec
func:'steepest_descent' took: 0.0151 sec
func:'steepest_descent' took: 0.0149 sec
func:'steepest_descent' took: 0.0178 sec
func:'steepest_descent' took: 0.0157 sec
func:'steepest_descent' took: 0.0153 sec
func:'steepest_descent' took: 0.0144 sec
func:'steepest_descent' took: 0.0139 sec
The average performance (step counts) is 1203.0 steps
```

Stochastic Gradient Descent (SGD)

In [80]:
```
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_po
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} ste
```

In [81]:
```
avg_performance_SGD(10, np.array([1.5, 1.5]), RosenbrockBanana, RosenbrockBanar
```

```
func:'stochastic_gradient_descent' took: 0.0834 sec
func:'stochastic_gradient_descent' took: 0.0482 sec
func:'stochastic_gradient_descent' took: 0.0423 sec
func:'stochastic_gradient_descent' took: 0.0410 sec
func:'stochastic_gradient_descent' took: 0.0470 sec
func:'stochastic_gradient_descent' took: 0.0722 sec
func:'stochastic_gradient_descent' took: 0.0421 sec
func:'stochastic_gradient_descent' took: 0.0338 sec
func:'stochastic_gradient_descent' took: 0.0328 sec
func:'stochastic_gradient_descent' took: 0.0294 sec
The average performance (step counts) is 1947.6 steps
```

CG

In [82]:
```
def avg_performance_CG(iterations, starting_point, function):
    count = 0
    for i in range(iterations):
        count += minimize(function, starting_point, method = "CG", options={'di
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} ste
```

In [83]:
```
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 [84]:
```
def avg_performance_BFGS(iterations, starting_point, function):
    count = 0
    for i in range(iterations):
        count += minimize(function, starting_point, method = "BFGS", options={'
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} ste
```

In [85]:
```
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 [77]:
```
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 [87]:
```
avg_performance_SGD(10, np.array([-1.5,-1.5]), threeHumpCamel, first_deriv_THC)
```

```
func:'stochastic_gradient_descent' took: 0.0062 sec
func:'stochastic_gradient_descent' took: 0.0051 sec
func:'stochastic_gradient_descent' took: 0.0020 sec
func:'stochastic_gradient_descent' took: 0.0021 sec
func:'stochastic_gradient_descent' took: 0.0025 sec
func:'stochastic_gradient_descent' took: 0.0015 sec
func:'stochastic_gradient_descent' took: 0.0016 sec
func:'stochastic_gradient_descent' took: 0.0025 sec
func:'stochastic_gradient_descent' took: 0.0024 sec
func:'stochastic_gradient_descent' took: 0.0042 sec
The average performance (step counts) is 68.7 steps
```

In [88]:
```
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 [89]:
```
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 [142…

```
@timeit
def SGDM(func,first_derivate,starting_point,stepsize,momentum=0.9,tol=1e-5,stoc
    # 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 gradie
            random_vector = np.random.normal(0,1,2)
            stochastic_deriv =  random_vector/LA.norm(random_vector)*LA.norm(de

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


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

        # 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. wha
            stepsize = 1.2 * stepsize
        else:
            # the step makes function evaluation higher - it is a bad step. wha
            # if stepsize is too small, clear previous direction because we al

            if stepsize<1e-5:
                previous_direction=previous_direction-previous_direction

            else:
                # do the same as SGD here
                stepsize = 0.5 * stepsize
        count+=1
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.asar
```

In [148…

```
SGDM(threeHumpCamel,first_deriv_THC,np.array([-1.5, -1.5]),0.1,momentum=0.9,tc
```

```
func:'SGDM' took: 0.0225 sec
```

Out[148]:
```
{'x': array([-1.5, -1.5]),
 'evaluation': 5.5828125,
 'path': array([[-1.50000000e+00, -1.50000000e+00],
        [-9.72229420e-01,  6.85913376e-01],
        [ 2.58475167e-01,  2.09131023e+00],
        [ 7.45873683e-01,  2.15381226e+00],
        [ 9.24637705e-01,  2.22691904e+00],
        [ 1.02488851e+00,  2.16427343e+00],
        [ 1.15138379e+00,  1.96655596e+00],
        [ 7.50293980e-01,  1.37577286e+00],
        [ 1.63255620e-01,  4.58484876e-01],
        [-4.85716050e-01, -5.21546969e-01],
        [-7.78192868e-01, -9.61686414e-01],
        [-8.32068798e-01, -1.09333272e+00],
        [-8.52350442e-01, -1.15941449e+00],
        [-8.43284676e-01, -1.19867030e+00],
        [-8.38754431e-01, -1.20914770e+00],
        [-8.36771113e-01, -1.21372937e+00],
        [-8.34947716e-01, -1.21676564e+00],
        [-8.32899004e-01, -1.21680754e+00],
        [-8.30681813e-01, -1.21589007e+00],
        [-8.25786574e-01, -1.21362371e+00],
        [-8.19774458e-01, -1.20902567e+00],
        [-8.11678585e-01, -1.20521608e+00],
        [-8.00982755e-01, -1.20246427e+00],
        [-7.88850541e-01, -1.19634915e+00],
        [-7.70472808e-01, -1.19058536e+00],
        [-7.43633263e-01, -1.18374179e+00],
        [-7.12441310e-01, -1.17042003e+00],
        [-6.74135356e-01, -1.15860495e+00],
        [-6.33160043e-01, -1.14130877e+00],
        [-5.77595991e-01, -1.12030540e+00],
        [-5.05960999e-01, -1.09106586e+00],
        [-4.19109713e-01, -1.04922156e+00],
        [-3.21122642e-01, -9.92074243e-01],
        [-2.15946662e-01, -9.20780264e-01],
        [-1.04654359e-01, -8.39923142e-01],
        [ 2.38378568e-02, -7.52508216e-01],
        [ 1.71386854e-01, -6.51461962e-01],
        [ 3.28746038e-01, -5.32077907e-01],
        [ 4.87703339e-01, -3.95700321e-01],
        [ 5.59004038e-01, -3.35306350e-01],
        [ 5.85409281e-01, -3.11205363e-01],
        [ 5.97134231e-01, -3.01123358e-01],
        [ 6.02172191e-01, -2.95912172e-01],
        [ 6.03367837e-01, -2.93694512e-01],
        [ 6.03887999e-01, -2.92795085e-01],
        [ 6.03863922e-01, -2.92328819e-01],
        [ 6.03823802e-01, -2.91759702e-01],
        [ 6.03778890e-01, -2.91121391e-01],
        [ 6.03403023e-01, -2.90215901e-01],
        [ 6.02729014e-01, -2.88957367e-01],
        [ 6.01380005e-01, -2.87789383e-01],
        [ 5.99922728e-01, -2.86548452e-01],
        [ 5.97517613e-01, -2.85551362e-01],
        [ 5.93901775e-01, -2.84853126e-01],
        [ 5.88644625e-01, -2.84268702e-01],
        [ 5.81358844e-01, -2.83787570e-01],
        [ 5.71652835e-01, -2.83621032e-01],
        [ 5.61086529e-01, -2.83865857e-01],
```

```
[ 5.47888931e-01, -2.82938415e-01],
[ 5.30965993e-01, -2.81030788e-01],
[ 5.09495066e-01, -2.79725614e-01],
[ 4.84530574e-01, -2.76383297e-01],
[ 4.56171379e-01, -2.74686712e-01],
[ 4.23243240e-01, -2.75037110e-01],
[ 3.82795295e-01, -2.75372873e-01],
[ 3.37513559e-01, -2.73098639e-01],
[ 2.83624077e-01, -2.70689826e-01],
[ 2.21641316e-01, -2.69368184e-01],
[ 1.51869543e-01, -2.65101104e-01],
[ 7.44391534e-02, -2.60779871e-01],
[-1.05647861e-02, -2.52831761e-01],
[-4.74862120e-02, -2.47169005e-01],
[-6.40672254e-02, -2.43466610e-01],
[-7.12128596e-02, -2.41129791e-01],
[-7.43608920e-02, -2.40120535e-01],
[-7.57849661e-02, -2.39530063e-01],
[-7.64345270e-02, -2.39203547e-01],
[-7.67339956e-02, -2.39042991e-01],
[-7.68492265e-02, -2.38968664e-01],
[-7.68916669e-02, -2.38934672e-01],
[-7.68882727e-02, -2.38923902e-01],
[-7.68734054e-02, -2.38911733e-01],
[-7.68532065e-02, -2.38883244e-01],
[-7.68323179e-02, -2.38837920e-01],
[-7.68099361e-02, -2.38788790e-01],
[-7.67874359e-02, -2.38733784e-01],
[-7.67369271e-02, -2.38674770e-01],
[-7.66487749e-02, -2.38609184e-01],
[-7.65088452e-02, -2.38509635e-01],
[-7.63330005e-02, -2.38346278e-01],
[-7.60775620e-02, -2.38132806e-01],
[-7.57571781e-02, -2.37914285e-01],
[-7.53964614e-02, -2.37688694e-01],
[-7.50090311e-02, -2.37351367e-01],
[-7.45925301e-02, -2.36874154e-01],
[-7.41620151e-02, -2.36248155e-01],
[-7.36208234e-02, -2.35604788e-01],
[-7.29557286e-02, -2.34948168e-01],
[-7.21013512e-02, -2.33958755e-01],
[-7.11744888e-02, -2.32605544e-01],
[-6.98803663e-02, -2.31168990e-01],
[-6.83389036e-02, -2.29146002e-01],
[-6.61936880e-02, -2.26518225e-01],
[-6.38077534e-02, -2.23732381e-01],
[-6.06332034e-02, -2.20744078e-01],
[-5.70230776e-02, -2.17626761e-01],
[-5.32949616e-02, -2.13604114e-01],
[-4.91873575e-02, -2.09322140e-01],
[-4.45264868e-02, -2.03303140e-01],
[-3.96659633e-02, -1.96609756e-01],
[-3.48025383e-02, -1.88403294e-01],
[-2.99444198e-02, -1.78980425e-01],
[-2.49668499e-02, -1.68527322e-01],
[-1.76772142e-02, -1.54922274e-01],
[-1.03853168e-02, -1.39494302e-01],
[-3.28042103e-03, -1.21452019e-01],
[ 4.49366463e-03, -9.90761868e-02],
[ 1.23610958e-02, -7.45310290e-02],
```

```
[ 2.22176173e-02, -4.68070485e-02],
[ 3.31446168e-02, -1.65896466e-02],
[ 4.24032178e-02,  1.52721720e-02],
[ 4.63386548e-02,  2.99790421e-02],
[ 4.73015225e-02,  3.69796741e-02],
[ 4.74721481e-02,  4.03253006e-02],
[ 4.71814175e-02,  4.17637894e-02],
[ 4.68745321e-02,  4.23200983e-02],
[ 4.66647939e-02,  4.25054156e-02],
[ 4.63380504e-02,  4.27098825e-02],
[ 4.59919561e-02,  4.28758623e-02],
[ 4.54670582e-02,  4.29786691e-02],
[ 4.49139355e-02,  4.30395337e-02],
[ 4.43327522e-02,  4.30349221e-02],
[ 4.35249153e-02,  4.28259904e-02],
[ 4.23390797e-02,  4.24926125e-02],
[ 4.10732402e-02,  4.19583184e-02],
[ 3.93164113e-02,  4.13943848e-02],
[ 3.73241770e-02,  4.08777755e-02],
[ 3.51635609e-02,  3.99755992e-02],
[ 3.25395812e-02,  3.91664098e-02],
[ 2.90731435e-02,  3.83494713e-02],
[ 2.49078860e-02,  3.76602867e-02],
[ 1.95339787e-02,  3.65313601e-02],
[ 1.28318911e-02,  3.49067417e-02],
[ 4.75372785e-03,  3.29214239e-02],
[-4.56947439e-03,  3.06984998e-02],
[-1.48612573e-02,  2.82723566e-02],
[-2.56853354e-02,  2.57202595e-02],
[-2.99889295e-02,  2.44765666e-02],
[-3.17447330e-02,  2.37086755e-02],
[-3.24212272e-02,  2.34403506e-02],
[-3.26838504e-02,  2.32654158e-02],
[-3.27641881e-02,  2.31635731e-02],
[-3.27823575e-02,  2.31055552e-02],
[-3.27839480e-02,  2.30726185e-02],
[-3.27715830e-02,  2.30388270e-02],
[-3.27582574e-02,  2.30020668e-02],
[-3.27401502e-02,  2.29529869e-02],
[-3.27065118e-02,  2.29110036e-02],
[-3.26698703e-02,  2.28673892e-02],
[-3.25971500e-02,  2.28080005e-02],
[-3.24899944e-02,  2.27618482e-02],
[-3.23728865e-02,  2.26999665e-02],
[-3.22041681e-02,  2.26004144e-02],
[-3.19962569e-02,  2.25238068e-02],
[-3.17710429e-02,  2.24450161e-02],
[-3.14975475e-02,  2.24012448e-02],
[-3.11466438e-02,  2.22837358e-02],
[-3.06201214e-02,  2.21536712e-02],
[-2.99417378e-02,  2.20856963e-02],
[-2.91879778e-02,  2.20611417e-02],
[-2.83034640e-02,  2.21254652e-02],
[-2.71049031e-02,  2.20846680e-02],
[-2.56692330e-02,  2.21670325e-02],
[-2.38280318e-02,  2.20831909e-02],
[-2.16265313e-02,  2.21345809e-02],
[-1.88403746e-02,  2.21446622e-02],
[-1.58526864e-02,  2.20576604e-02],
[-1.24201234e-02,  2.20597955e-02],
```

```
[-8.73937491e-03,  2.20348124e-02],
[-4.44219316e-03,  2.18113391e-02],
[ 2.67937791e-04,  2.15545108e-02],
[ 2.13415138e-03,  2.11845161e-02],
[ 2.81641013e-03,  2.09547658e-02],
[ 3.04478319e-03,  2.07853136e-02],
[ 3.18948502e-03,  2.05532338e-02],
[ 3.22298191e-03,  2.02427268e-02],
[ 3.20914867e-03,  1.99257401e-02],
[ 3.08325814e-03,  1.93999130e-02],
[ 2.80577561e-03,  1.88390129e-02],
[ 2.28471082e-03,  1.80653077e-02],
[ 1.47366278e-03,  1.70487929e-02],
[ 4.93568656e-04,  1.56293162e-02],
[-7.45360554e-04,  1.37847820e-02],
[-1.99302083e-03,  1.15576672e-02],
[-3.51506792e-03,  8.96678933e-03],
[-5.23921167e-03,  6.00270371e-03],
[-6.80780780e-03,  2.82659754e-03],
[-7.47430172e-03,  1.31182204e-03],
[-7.70620130e-03,  7.16975087e-04],
[-7.74807063e-03,  4.88926824e-04],
[-7.72732662e-03,  3.92645250e-04],
[-7.70415690e-03,  3.42334905e-04],
[-7.65818764e-03,  2.98507517e-04],
[-7.60842411e-03,  2.58174424e-04],
[-7.54541577e-03,  2.34048304e-04],
[-7.43787251e-03,  2.06493163e-04],
[-7.28424075e-03,  1.62413504e-04],
[-7.06515870e-03,  1.08278931e-04],
[-6.76287342e-03,  5.51578501e-05],
[-6.43734626e-03,  3.31791369e-06],
[-6.07672053e-03, -1.36383734e-05],
[-5.62218569e-03,  2.72122562e-05],
[-5.02291213e-03,  7.72143821e-05],
[-4.29378367e-03,  9.15117375e-05],
[-3.42485935e-03,  6.51179515e-05],
[-2.45977354e-03,  4.81397767e-07],
[-1.36267486e-03, -1.06985952e-04],
[-1.40978697e-04, -2.47790670e-04],
[ 1.18070732e-03, -3.88872306e-04],
[ 1.77536412e-03, -4.51514735e-04],
[ 2.02653965e-03, -4.70213133e-04],
[ 2.13923846e-03, -4.77288495e-04],
[ 2.18289068e-03, -4.82118003e-04],
[ 2.20248169e-03, -4.84049102e-04],
[ 2.21084164e-03, -4.85865993e-04],
[ 2.21458925e-03, -4.86886525e-04],
[ 2.21627357e-03, -4.87333728e-04],
[ 2.21679779e-03, -4.87566322e-04],
[ 2.21698470e-03, -4.87621407e-04],
[ 2.21688071e-03, -4.87592703e-04],
[ 2.21662850e-03, -4.87583123e-04],
[ 2.21624455e-03, -4.87504904e-04],
[ 2.21563038e-03, -4.87460468e-04],
[ 2.21494053e-03, -4.87352714e-04],
[ 2.21404013e-03, -4.87111572e-04],
[ 2.21306561e-03, -4.86914186e-04],
[ 2.21192581e-03, -4.86559380e-04],
[ 2.21028261e-03, -4.86407676e-04],
```

```
[ 2.20849535e-03, -4.86191421e-04],
[ 2.20617354e-03, -4.85648784e-04],
[ 2.20291522e-03, -4.85423432e-04],
[ 2.19883498e-03, -4.84735976e-04],
[ 2.19374624e-03, -4.83462528e-04],
[ 2.18676187e-03, -4.82248216e-04],
[ 2.17748293e-03, -4.80782188e-04],
[ 2.16576941e-03, -4.80218821e-04],
[ 2.15068434e-03, -4.80308094e-04],
[ 2.13385460e-03, -4.79450853e-04],
[ 2.11223859e-03, -4.79536148e-04],
[ 2.08564063e-03, -4.78105306e-04],
[ 2.05534468e-03, -4.74560546e-04],
[ 2.01664317e-03, -4.71024811e-04],
[ 1.96982513e-03, -4.64505872e-04],
[ 1.91600685e-03, -4.53985501e-04],
[ 1.84884177e-03, -4.40993522e-04],
[ 1.77495145e-03, -4.24024320e-04],
[ 1.68906528e-03, -4.00145514e-04],
[ 1.59478193e-03, -3.70796293e-04],
[ 1.48349863e-03, -3.32001135e-04],
[ 1.34489503e-03, -2.93360429e-04],
[ 1.17810244e-03, -2.60533707e-04],
[ 9.98066335e-04, -2.27208423e-04],
[ 7.96339860e-04, -1.82473759e-04],
[ 5.67711893e-04, -1.46023309e-04],
[ 3.09378844e-04, -1.16184147e-04],
[ 2.20690776e-05, -8.91386840e-05],
[-2.87295404e-04, -5.90009206e-05],
[-4.26704823e-04, -4.43003926e-05],
[-4.84617188e-04, -3.84488126e-05],
[-5.10692147e-04, -3.57610993e-05],
[-5.21925673e-04, -3.50883315e-05],
[-5.26181241e-04, -3.46463961e-05],
[-5.27767280e-04, -3.42341791e-05],
[-5.28286830e-04, -3.39811131e-05],
[-5.28496441e-04, -3.38086359e-05],
[-5.28541157e-04, -3.37312694e-05],
[-5.28536155e-04, -3.36908292e-05],
[-5.28526098e-04, -3.36546031e-05],
[-5.28504770e-04, -3.36275296e-05],
[-5.28439621e-04, -3.36006615e-05],
[-5.28323660e-04, -3.35466177e-05],
[-5.28139112e-04, -3.34648514e-05],
[-5.27864517e-04, -3.33704031e-05],
[-5.27510588e-04, -3.32118110e-05],
[-5.27064918e-04, -3.29714038e-05],
[-5.26502882e-04, -3.26299499e-05],
[-5.25894840e-04, -3.22646395e-05],
[-5.25239753e-04, -3.18260901e-05],
[-5.24405432e-04, -3.14349166e-05],
[-5.23490845e-04, -3.09075658e-05],
[-5.22500750e-04, -3.03457747e-05],
[-5.21319887e-04, -2.95070772e-05],
[-5.20046133e-04, -2.84832580e-05],
[-5.18215922e-04, -2.70918593e-05],
[-5.15966501e-04, -2.58322836e-05],
[-5.12790038e-04, -2.45393638e-05],
[-5.08447596e-04, -2.31708327e-05],
[-5.03022764e-04, -2.20799714e-05],
```

```
[-4.96928896e-04, -2.01975129e-05],
[-4.88808621e-04, -1.81624587e-05],
[-4.78629672e-04, -1.49299201e-05],
[-4.66559930e-04, -1.00576675e-05],
[-4.51193433e-04, -3.79513102e-06],
[-4.34535959e-04,  3.88616513e-06],
[-4.16552973e-04,  1.29655935e-05],
[-3.95885590e-04,  2.50510541e-05],
[-3.73446278e-04,  3.77304076e-05],
[-3.44420797e-04,  5.13971701e-05],
[-3.11973005e-04,  6.44471155e-05],
[-2.71912991e-04,  8.07603896e-05],
[-2.28141530e-04,  9.70358177e-05],
[-1.75302806e-04,  1.14520756e-04],
[-1.15573211e-04,  1.30783475e-04],
[-4.88545104e-05,  1.49313617e-04],
[ 2.30736240e-05,  1.69296675e-04],
[ 5.54938480e-05,  1.78248986e-04],
[ 6.88824099e-05,  1.82257990e-04],
[ 7.41240874e-05,  1.83876032e-04],
[ 7.61403743e-05,  1.84616235e-04],
[ 7.70677726e-05,  1.84814996e-04],
[ 7.74622916e-05,  1.84806225e-04],
[ 7.75965606e-05,  1.84758713e-04],
[ 7.76505131e-05,  1.84712461e-04],
[ 7.76722118e-05,  1.84693520e-04],
[ 7.76755698e-05,  1.84683897e-04],
[ 7.76795876e-05,  1.84673005e-04],
[ 7.76827699e-05,  1.84652848e-04],
[ 7.76803529e-05,  1.84620044e-04],
[ 7.76669128e-05,  1.84585558e-04],
[ 7.76543827e-05,  1.84538917e-04],
[ 7.76208510e-05,  1.84480792e-04],
[ 7.75634497e-05,  1.84417121e-04],
[ 7.74892331e-05,  1.84320780e-04],
[ 7.73936197e-05,  1.84223251e-04],
[ 7.72964339e-05,  1.84100573e-04],
[ 7.71925878e-05,  1.83931235e-04],
[ 7.70864886e-05,  1.83738889e-04],
[ 7.69068296e-05,  1.83482140e-04],
[ 7.67072688e-05,  1.83133921e-04],
[ 7.64939852e-05,  1.82755213e-04],
[ 7.61617889e-05,  1.82348643e-04],
[ 7.56984428e-05,  1.81925747e-04],
[ 7.50944194e-05,  1.81497630e-04],
[ 7.44412872e-05,  1.80873066e-04],
[ 7.37672265e-05,  1.80143181e-04],
[ 7.30837497e-05,  1.79232830e-04],
[ 7.23582999e-05,  1.77990345e-04],
[ 7.16384002e-05,  1.76481881e-04],
[ 7.05711726e-05,  1.74380979e-04],
[ 6.90372339e-05,  1.71566408e-04],
[ 6.73133447e-05,  1.68585114e-04],
[ 6.55667289e-05,  1.64947924e-04],
[ 6.38183006e-05,  1.60744991e-04],
[ 6.18452241e-05,  1.55250549e-04],
[ 5.93823230e-05,  1.49524798e-04],
[ 5.58127591e-05,  1.41930954e-04],
[ 5.19725093e-05,  1.32407533e-04],
[ 4.65809968e-05,  1.22380774e-04],
```

```
[ 4.05329556e-05,  1.11738707e-04],
[ 3.30560128e-05,  1.00641465e-04],
[ 2.53059537e-05,  8.81377637e-05],
[ 1.48972467e-05,  7.28673310e-05],
[ 3.98809339e-06,  5.49156971e-05],
[-7.57215544e-06,  3.41122750e-05],
[-2.00440613e-05,  1.16435798e-05],
[-3.22649728e-05, -1.30818023e-05],
[-3.78209694e-05, -2.38544008e-05],
[-3.94265999e-05, -2.86928779e-05],
[-3.96904327e-05, -3.05987762e-05],
[-3.97340731e-05, -3.12637312e-05],
[-3.97391358e-05, -3.14794430e-05],
[-3.96770055e-05, -3.15564394e-05],
[-3.95715570e-05, -3.16611700e-05],
[-3.94109509e-05, -3.17014422e-05],
[-3.92202746e-05, -3.16688652e-05],
[-3.90189406e-05, -3.15764715e-05],
[-3.87174602e-05, -3.15201735e-05],
[-3.82819210e-05, -3.13127204e-05],
[-3.76778212e-05, -3.09150872e-05],
[-3.70422323e-05, -3.04051287e-05],
[-3.61592474e-05, -2.96150183e-05],
[-3.52253861e-05, -2.87083443e-05],
[-3.41550615e-05, -2.78010840e-05],
[-3.30119778e-05, -2.65941247e-05],
[-3.17043340e-05, -2.53797808e-05],
[-2.99874733e-05, -2.35471487e-05],
[-2.77150571e-05, -2.09863209e-05],
[-2.51900825e-05, -1.77132028e-05],
[-2.24702537e-05, -1.41636560e-05],
[-1.94989495e-05, -9.84180583e-06],
[-1.62448344e-05, -5.25414187e-06],
[-1.24474902e-05,  2.97603718e-07],
[-7.75608474e-06,  6.67341148e-06],
[-2.61679280e-06,  1.36779788e-05],
[-2.37941513e-07,  1.66472467e-05],
[ 7.67203136e-07,  1.78369326e-05],
[ 1.15711712e-06,  1.83560417e-05],
[ 1.33524699e-06,  1.85877648e-05],
[ 1.40969204e-06,  1.86690917e-05],
[ 1.44489367e-06,  1.87041158e-05],
[ 1.45967108e-06,  1.87203006e-05],
[ 1.46597979e-06,  1.87247570e-05],
[ 1.46917914e-06,  1.87259922e-05],
[ 1.47004381e-06,  1.87260641e-05],
[ 1.47017381e-06,  1.87255323e-05],
[ 1.47038809e-06,  1.87249002e-05],
[ 1.47087300e-06,  1.87236269e-05],
[ 1.47171084e-06,  1.87216111e-05],
[ 1.47294603e-06,  1.87184608e-05],
[ 1.47376767e-06,  1.87152103e-05],
[ 1.47519965e-06,  1.87107919e-05],
[ 1.47558951e-06,  1.87034725e-05],
[ 1.47562317e-06,  1.86957488e-05],
[ 1.47408771e-06,  1.86837162e-05],
[ 1.47132981e-06,  1.86662329e-05],
[ 1.46745007e-06,  1.86420361e-05],
[ 1.45950550e-06,  1.86099934e-05],
[ 1.44454047e-06,  1.85724727e-05],
```

```
           [ 1.42129255e-06,  1.85301023e-05],
           [ 1.38895371e-06,  1.84748275e-05],
           [ 1.34471568e-06,  1.84044800e-05],
           [ 1.29313851e-06,  1.83127445e-05],
           [ 1.22775409e-06,  1.81958232e-05],
           [ 1.15039804e-06,  1.80471852e-05],
           [ 1.04641857e-06,  1.78684446e-05],
           [ 9.29382946e-07,  1.76773415e-05],
           [ 8.12779506e-07,  1.74534121e-05],
           [ 6.98414261e-07,  1.71935674e-05],
           [ 5.73150878e-07,  1.68642511e-05],
           [ 4.38473640e-07,  1.64528947e-05],
           [ 3.12339013e-07,  1.59735400e-05],
           [ 1.96191912e-07,  1.54029436e-05],
           [ 9.49080632e-08,  1.47580299e-05],
           [-4.14533388e-08,  1.39550461e-05],
           [-2.56019386e-07,  1.29816466e-05],
           [-4.68522682e-07,  1.19170723e-05],
           [-7.10969033e-07,  1.07715409e-05],
           [-9.49510985e-07,  9.52047117e-06],
           [-1.18359416e-06,  8.15728650e-06],
           [-1.39744012e-06,  6.66319454e-06],
           [-1.70453252e-06,  4.99808738e-06]]),
 'count': 440}
```

In [150…

```python
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
    average_steps = count / iterations
    return print(f"The average performance (step counts) is {average_steps} ste
```

In [151…

```python
avg_SDGM(10, np.array([-1.5, -1.5]), threeHumpCamel, first_deriv_THC)
```

```
func:'SGDM' took: 0.0022 sec
func:'SGDM' took: 0.0261 sec
func:'SGDM' took: 0.0015 sec
func:'SGDM' took: 0.0136 sec
func:'SGDM' took: 0.0148 sec
func:'SGDM' took: 0.0004 sec
func:'SGDM' took: 0.0003 sec
func:'SGDM' took: 0.0007 sec
func:'SGDM' took: 0.0182 sec
func:'SGDM' took: 0.0002 sec
The average performance (step counts) is 174.7 steps
```

```
<ipython-input-77-96b19697719a>:9: RuntimeWarning: invalid value encountered i
n double_scalars
  return np.array([x**5 + 4.2*x**3 + 4*x + y, x+2*y])
<ipython-input-77-96b19697719a>:4: RuntimeWarning: invalid value encountered i
n double_scalars
  return 2*x**2 - 1.05*x**4 +(x**6/6) + x*y + y**2
<ipython-input-77-96b19697719a>:9: RuntimeWarning: overflow encountered in dou
ble_scalars
  return np.array([x**5 + 4.2*x**3 + 4*x + y, x+2*y])
<ipython-input-77-96b19697719a>:4: RuntimeWarning: overflow encountered in dou
ble_scalars
  return 2*x**2 - 1.05*x**4 +(x**6/6) + x*y + y**2
<ipython-input-142-d39edd234424>:22: RuntimeWarning: invalid value encountered
in add
  direction=-(deriv+stochastic_injection*stochastic_deriv) + (momentum*previou
s_direction)
```

Average Performance	SGDM	SGD	CG	BFGS
	174.7	68.7	7	8

On average, I did not find a better result using SGDM compared to SGD, CG, or BFGS in finding the global minimum in terms of fewer steps.

You've reached the end of HW1

In [ ]: