Debugging outputs

1. (a) new (x,y) position: [0.15 0.9]

   (b) Converge after 41 accepted steps. Converge to point [-0.99999982 0.99999455] with value -2.9999999999721534. took: 0.0020 sec

2. (b) It is a stochastic method so your answer may vary. It takes ~1700 steps to converge and took ~0.1 sec

3. (b) takes ~250 steps to converge and took ~0.02 sec

Helper functions

A timing decorator. Put at the beginning of your function so that every time your function is called it'll print out the execution time

In [2]:
```
import time
import numpy as np

def timeit(f):

    def timed(*args, **kw):

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

        print(f'func:{f.__name__} took: {te-ts:.4f} sec')
        return result

    return timed
```

A function that help to visualize the optimization pathway:

In [3]:
```
%matplotlib notebook
def draw_path(func,path,x_min=-2,x_max=2,y_min=-2,y_max=2):
    a=np.linspace(x_min,x_max,100)
    b=np.linspace(y_min,y_max,100)
    x,y=np.meshgrid(a,b)
    z=func((x,y))
    fig,ax=plt.subplots()
    my_contour=ax.contour(x,y,z,50)
    plt.colorbar(my_contour)
    ax.plot(path[:,0],path[:,1])
```

Templates for algorithm you need to implement

1 a)

In [4]:
```
def func2d(X):
    '''
    A 2D function
    '''
    x,y = X
    return x**4-x**2+y**2+2*x*y-2

# def func2d_derivative(X):
#     x,y = X
#     return (4*x**3) - (2*x) + (2*y)

def func2d_derivative(X):
    x, y = X
    deriv_x = (4*x**3) - (2*x) + (2*y)
    deriv_y = (2*y) + (2*x)
    return np.array([deriv_x, deriv_y])

starting_point = np.array([1.5, 1.5])
stepsize = .1
new_point = starting_point - stepsize * func2d_derivative(starting_point)
display(new_point)
display(func2d(starting_point))
display(func2d(new_point))
```

```
array([0.15, 0.9 ])
7.5625
-0.9419937500000004
```

Answer

As shown above, the new (x,y) position will be (.15, .9). The value of the function at the starting point is 7.5625, while the value of the function at the new point is -0.9419937500000004. Since f(new_point) < f(starting_point), this is considered a good step. Since it is a goodstep, we will change the stepsize for the next step to be 1.2 * stepsize, or $1.2 * \lambda$

1 b)

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

```
    deriv = first_derivate(starting_point)

    count=0
    visited=[]
    while np.linalg.norm(deriv) > tol and count < 1e6:
        # calculate new point position
        deriv = first_derivate(starting_point)
        new_point = starting_point - stepsize * deriv
        if func(new_point) < func(starting_point):
            # the step makes function evaluation lower - it is a good step.
            stepsize  = 1.2 * stepsize
            visited.append(new_point)
            starting_point = new_point
        else:
            # the step makes function evaluation higher - it is a bad step.
            stepsize = 0.5 * stepsize
        count+=1
    # return the results
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.a
```

In [6]:

```
# result = steepest_descent(func2d,
#                  func2d_derivative,
#                  starting_point=np.array([1.5, 1.5]),
#                  stepsize=.1,
#                  tol=1e-5)
result = steepest_descent(func2d,
                          func2d_derivative,
                           starting_point = new_point,
                           stepsize = .1 * 1.2,
                           tol = 1E-5
                          )
print(result)
```

```
func:steepest_descent took: 0.0018 sec
{'x': array([-0.99999972,  0.99999691]), 'evaluation': -2.999999999991799,
'path': array([[-0.03162   ,  0.648     ],
       [-0.22733235,  0.47048256],
       [-0.4603766 ,  0.38644985],
       [-0.73063964,  0.41710875],
       [-0.91361447,  0.57314178],
       [-0.89067253,  0.77647099],
       [-0.98168777,  0.81739147],
       [-0.94167921,  0.88803587],
       [-1.0240441 ,  0.91571465],
       [-0.95964931,  0.94925202],
       [-1.01216804,  0.95311466],
       [-0.98795692,  0.96627781],
       [-0.99481157,  0.9720766 ],
       [-0.99412388,  0.97937406],
       [-0.99741632,  0.98505533],
       [-0.99646125,  0.99076871],
       [-1.00111334,  0.9939261 ],
       [-0.99723697,  0.99631795],
       [-1.00126535,  0.99668496],
       [-0.99895278,  0.99778247],
       [-0.99981882,  0.99811897],
       [-0.99948229,  0.99870549],
       [-1.00001741,  0.99902712],
       [-0.99975409,  0.99927314],
       [-0.99990384,  0.99941651],
       [-0.99986709,  0.99959085],
       [-0.99997668,  0.99970943],
       [-0.99988705,  0.9998471 ],
       [-1.00001432,  0.99985945],
       [-0.99993563,  0.99991689],
       [-0.99998875,  0.99992106],
       [-0.99998269,  0.99993914],
       [-0.99999092,  0.9999531 ],
       [-0.99999034,  0.99996764],
       [-0.9999977 ,  0.99997812],
       [-0.99999196,  0.99998896],
       [-1.00000165,  0.99998995],
       [-0.99999435,  0.99999462],
       [-0.99999982,  0.99999455],
       [-0.99999852,  0.99999607],
       [-0.99999972,  0.99999691]])}
```

In [7]:
```
print(len(result['path']))
```

```
41
```

Answer

As shown above, continuing steepest descents takes 41 accepted steps in order to converge to the local minimum with tolerance = $1 * 10^{-5}$

1 c)

In [9]:
```
from scipy.optimize import minimize

def save_step(*args):
    for arg in args:
        if type(arg) is np.ndarray:
            steps.append(arg)

@timeit
def minimize_function(x0, func, method):
    """
    Minimize a function

    Parameters
    ----------
    x0: np.ndarray
        Starting point
    func: function
        Scalar function to minimize
    method: str
        Method for minimization

    Returns
    -------
    res: OptimzizeResult
        Result object of scipy optimization
    """
    res = minimize(
        func,
        x0,
        method=method,
        options={"gtol": 1e-5, "disp":True},
        callback=save_step,
    )
    return res
```

In [10]:
```
# Conjugate Gradients minimization

steps = [starting_point]
res = minimize_function(starting_point, func2d, "CG")
print(res.x)
print(len(steps))
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 9
         Function evaluations: 78
         Gradient evaluations: 26
func:minimize_function took: 0.0153 sec
[-0.99999984  0.99999929]
10
```

In [11]:
```
# BFGS Minimization

steps = [starting_point]
res = minimize_function(starting_point, func2d, "BFGS")
print(res.x)
print(len(steps))
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 7
         Function evaluations: 24
         Gradient evaluations: 8
func:minimize_function took: 0.0067 sec
[ 0.99999979 -0.9999998 ]
8
```

Answer

In terms of steps, both conjugate gradient and BFGS minimization are more efficient than steepest descent. CG only took 10 steps, while BFGS only took 8 steps (including the initial starting step). Meanwhile, steepest descent took 41 steps, so both CG and BFGS are more efficient

2 a)

In [12]:
```
def rosenbrock(X):
    x, y = X
    return (1 - x)**2 + 10 * (y - x**2)**2

def rosenbrock_derivative(X):
    x, y = X
    partial_x = 2 * (x - 1) - 40 * x * (y - x**2)
    partial_y = 20 * (y - x ** 2)
    return np.array([partial_x, partial_y])

q2_start = np.array([-.5, 1.5])
result = steepest_descent(rosenbrock, rosenbrock_derivative, starting_point
print(result)
```

```
func:steepest_descent took: 0.0242 sec
{'x': array([0.99999105, 0.99998163]), 'evaluation': 8.230174326047824e-11,
'path': array([[-1.05      ,  0.875     ],
       [-0.845175  ,  0.94325   ],
       [-0.91805808,  0.86083548],
       ...,
       [ 0.99999093,  0.99998135],
       [ 0.99999089,  0.99998153],
       [ 0.99999105,  0.99998163]])}
```

In [13]:
```
print(len(result['path']))
# print(np.linalg.norm(rosenbrock_derivative(result['path'][-2])))
```

```
1205
```

Answer

Convergence to the minimum of the Rosenbrock function using my steepest descent algorithm took 1205 steps

2 b)

In [14]:
```
deriv = rosenbrock_derivative(q2_start)
random_array = rosenbrock_derivative(np.random.rand(deriv.shape[0]))
stochastic_deriv = random_array * (np.linalg.norm(deriv) / np.linalg.norm(ra
np.linalg.norm(deriv) == np.linalg.norm(stochastic_deriv)
```

Out[14]:
```
False
```

In [15]:
```
@timeit
def stochastic_gradient_descent(func,first_derivate,starting_point,stepsize,
    '''stochastic_injection: controls the magnitude of stochasticity (multip
        0 for no stochasticity, equivalent to SD.
        Use 1 in this homework to run SGD
    '''
    # evaluate the gradient at starting point
    deriv = first_derivate(starting_point)
    count=0
    visited=[]
    while np.linalg.norm(deriv) > tol and count < 1e5:
        deriv = first_derivate(starting_point)
        if stochastic_injection>0:
            stochastic_deriv = np.random.randn(deriv.shape[0])
            stochastic_deriv *= (np.linalg.norm(deriv) / np.linalg.norm(stoc
        else:
            stochastic_deriv=np.zeros(len(starting_point))
        direction=-(deriv+stochastic_injection*stochastic_deriv)
        # calculate new point position
        new_point = starting_point + (direction * stepsize)
        if func(new_point) < func(starting_point):
            # the step makes function evaluation lower - it is a good step.
            stepsize  = 1.2 * stepsize
            visited.append(new_point)
            starting_point = new_point
        else:
            # the step makes function evaluation higher - it is a bad step.
            stepsize = 0.5 * stepsize
        count+=1
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.a
```

In [16]:
```
result = stochastic_gradient_descent(rosenbrock, rosenbrock_derivative, np.a
print(result)
len(result['path'])
```

```
func:stochastic_gradient_descent took: 0.0440 sec
{'x': array([0.9999893 , 0.99997822]), 'evaluation': 1.1588020850588699e-10,
'path': array([[-0.87063551,  1.54699991],
       [-1.38606491,  1.61024359],
       [-1.31603513,  1.79393191],
       ...,
       [ 0.99998932,  0.99997803],
       [ 0.99998924,  0.99997803],
       [ 0.9999893 ,  0.99997822]]), 'count': 2258}
```

Out[16]: 1788

2 c)

In [17]:

```python
# Conjugate Gradients minimization Rosenbrock

steps = [q2_start]
res = minimize_function(q2_start, rosenbrock, "CG")
print(res.x)
print(len(steps))
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 132
         Gradient evaluations: 44
func:minimize_function took: 0.0179 sec
[0.99999955 0.99999908]
21
```

In [18]:

```python
# BFGS Minimization Rosenbrock

steps = [q2_start]
res = minimize_function(q2_start, rosenbrock, "BFGS")
print(res.x)
print(len(steps))
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 22
         Function evaluations: 93
         Gradient evaluations: 31
func:minimize_function took: 0.0133 sec
[0.99999959 0.99999917]
23
```

The SGD optimization algorithm is significantly worse than the Conjugate Gradients or BFGS algorithm, as CG and BFGS only take 20 and 22 iterations respectively, while the SGD takes ~1700 steps.

2 d)

You can draw a firm conclusion with just one run of CG and BFGS because each run of those algorithms will always have the exact same output; this is because these algorithms aren't stochastic, and will follow the same optimization path each time. However, for our stochastic gradient descent, because we are adding in randomization, we need to evaluate it multiple times; this is because each time we run it, the optimization path will be different due to the addition of randomness.

2 e)

In [345…

```
import random

averageCG, averageBFGS, averageSGD = 0, 0, 0
numiters = 10
for i in range(numiters):
    start = random.sample(range(-20, 20), 2)
    steps = []
    minimize_function(start, rosenbrock, "CG")
    averageCG += len(steps)

    steps = []
    minimize_function(start, rosenbrock, "BFGS")
    averageBFGS += len(steps)

    SGD = stochastic_gradient_descent(rosenbrock,
                                      rosenbrock_derivative,
                                      q2_start,
                                      stepsize = .1,
                                      tol=1e-5,
                                      stochastic_injection=1)
    averageSGD = len(SGD['path'])

averageCG /= numiters
averageBFGS /= numiters
averageSGD /= numiters
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 13
         Function evaluations: 113
         Gradient evaluations: 37
func:minimize_function took: 0.0149 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 48
         Function evaluations: 177
         Gradient evaluations: 59
func:minimize_function took: 0.0124 sec
func:stochastic_gradient_descent took: 0.0289 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 14
         Function evaluations: 102
         Gradient evaluations: 34
func:minimize_function took: 0.0026 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 24
         Function evaluations: 90
         Gradient evaluations: 30
func:minimize_function took: 0.0033 sec
func:stochastic_gradient_descent took: 0.0229 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 21
         Function evaluations: 171
         Gradient evaluations: 57
func:minimize_function took: 0.0040 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 61
         Function evaluations: 240
         Gradient evaluations: 80
func:minimize_function took: 0.0063 sec
func:stochastic_gradient_descent took: 0.0193 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 12
         Function evaluations: 96
         Gradient evaluations: 32
func:minimize_function took: 0.0022 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 53
         Function evaluations: 192
         Gradient evaluations: 64
func:minimize_function took: 0.0047 sec
func:stochastic_gradient_descent took: 0.0218 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 9
         Function evaluations: 72
```

```
         Gradient evaluations: 24
func:minimize_function took: 0.0018 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 30
         Function evaluations: 117
         Gradient evaluations: 39
func:minimize_function took: 0.0032 sec
func:stochastic_gradient_descent took: 0.0204 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 17
         Function evaluations: 141
         Gradient evaluations: 47
func:minimize_function took: 0.0032 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 22
         Function evaluations: 84
         Gradient evaluations: 28
func:minimize_function took: 0.0021 sec
func:stochastic_gradient_descent took: 0.0203 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 16
         Function evaluations: 123
         Gradient evaluations: 41
func:minimize_function took: 0.0029 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 61
         Function evaluations: 231
         Gradient evaluations: 77
func:minimize_function took: 0.0054 sec
func:stochastic_gradient_descent took: 0.0221 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 15
         Function evaluations: 117
         Gradient evaluations: 39
func:minimize_function took: 0.0027 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 55
         Function evaluations: 222
         Gradient evaluations: 74
func:minimize_function took: 0.0052 sec
func:stochastic_gradient_descent took: 0.0220 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 18
         Function evaluations: 147
         Gradient evaluations: 49
func:minimize_function took: 0.0038 sec
Optimization terminated successfully.
         Current function value: 0.000000
```

```
         Iterations: 59
         Function evaluations: 207
         Gradient evaluations: 69
func:minimize_function took: 0.0051 sec
func:stochastic_gradient_descent took: 0.0202 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 15
         Function evaluations: 102
         Gradient evaluations: 34
func:minimize_function took: 0.0022 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 46
         Function evaluations: 171
         Gradient evaluations: 57
func:minimize_function took: 0.0041 sec
func:stochastic_gradient_descent took: 0.0190 sec
```

In [301…

```
print(averageCG, averageBFGS, averageSGD)
```

```
17.0 48.2 171.2
```

Answer

For optimization of the Rosenbrock Banana Function, the CG method works best, followed by BFGS, and finally SGD. On average, CG takes the least number of steps, and BFGS follows closely behind. However, SGD takes significantly more steps to converge. Therefore, the non-stochastic methods of CG and BFGS are superior performers for this Rosenbrock Banana Function

3 a)

In [19]:

```
def three_hump_camel(X):
    x, y = X
    return 2 * x**2 - 1.05 * x**4 + (x**6) / 6 + x * y + y**2

def three_hump_camel_derivative(X):
    x, y = X
    partial_x = 4 * x**3 - 4.2 * x**3 + x**5 + y
    partial_y = x + 2 * y
    return np.array([partial_x, partial_y])
```

In [20]:

```
SGD = stochastic_gradient_descent(three_hump_camel,
                                  three_hump_camel_derivative,
                                  np.array([.5, .5]),
                                  stepsize = .1,
                                  tol=1e-5,
                                  stochastic_injection=1)
print(SGD["x"])
```

```
print(len(SGD['path']))
print(SGD['count'])
```

```
func:stochastic_gradient_descent took: 0.8998 sec
[ 0.28104285 -0.09157593]
20
100000
```

In [134…

```
averageCG, averageBFGS, averageSGD = 0, 0, 0
numFoundGlobalCG, numFoundGlobalBFGS, numFoundGlobalSGD = 0, 0, 0
global_min = np.array([0, 0])
numiters = 10
for i in range(numiters):
    random_floats = np.random.uniform(-2, 2, size=2)
    start = np.array(random_floats)
    steps = []
    result = minimize_function(start, three_hump_camel, "CG")
    averageCG += len(steps)
    if np.isclose(a=result.x[1], b=0):
        numFoundGlobalCG += 1

    steps = []
    result = minimize_function(start, three_hump_camel, "BFGS")
    averageBFGS += len(steps)
    if np.isclose(result.x[1], b=0):
        numFoundGlobalBFGS += 1

    SGD = stochastic_gradient_descent(three_hump_camel,
                                      three_hump_camel_derivative,
                                      start,
                                      stepsize = .1,
                                      tol=1e-5,
                                      stochastic_injection=1)
    averageSGD += len(SGD['path'])
    print(f"Stochastic gradient current func value: {SGD['evaluation']}")
    if np.isclose(SGD["evaluation"], 0):
        numFoundGlobalSGD += 1

averageCG /= numiters
averageBFGS /= numiters
averageSGD /= numiters
;
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 8
         Function evaluations: 60
         Gradient evaluations: 20
func:minimize_function took: 0.0061 sec
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
func:minimize_function took: 0.0024 sec
func:stochastic_gradient_descent took: 0.8918 sec
Stochastic gradient current func value: 0.05595688781942402
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 7
         Function evaluations: 57
         Gradient evaluations: 19
func:minimize_function took: 0.0017 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 8
         Function evaluations: 48
         Gradient evaluations: 16
func:minimize_function took: 0.0015 sec
func:stochastic_gradient_descent took: 0.8807 sec
Stochastic gradient current func value: 0.7553429642993508
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 45
         Gradient evaluations: 15
func:minimize_function took: 0.0013 sec
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 33
         Gradient evaluations: 11
func:minimize_function took: 0.0010 sec
func:stochastic_gradient_descent took: 0.8743 sec
Stochastic gradient current func value: 0.42200959091050694
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 6
         Function evaluations: 39
         Gradient evaluations: 13
func:minimize_function took: 0.0011 sec
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 30
         Gradient evaluations: 10
func:minimize_function took: 0.0008 sec
func:stochastic_gradient_descent took: 0.8838 sec
Stochastic gradient current func value: 0.721553306867756
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 5
         Function evaluations: 33
         Gradient evaluations: 11
func:minimize_function took: 0.0010 sec
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 5
         Function evaluations: 24
         Gradient evaluations: 8
func:minimize_function took: 0.0007 sec
func:stochastic_gradient_descent took: 0.8882 sec
Stochastic gradient current func value: 0.347297151489511
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 6
         Function evaluations: 39
         Gradient evaluations: 13
func:minimize_function took: 0.0013 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 6
         Function evaluations: 27
         Gradient evaluations: 9
func:minimize_function took: 0.0008 sec
func:stochastic_gradient_descent took: 0.8776 sec
Stochastic gradient current func value: 0.6733961130555794
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 39
         Gradient evaluations: 13
func:minimize_function took: 0.0012 sec
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 27
         Gradient evaluations: 9
func:minimize_function took: 0.0009 sec
func:stochastic_gradient_descent took: 0.8751 sec
Stochastic gradient current func value: 0.9243989916566333
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 5
         Function evaluations: 36
         Gradient evaluations: 12
func:minimize_function took: 0.0011 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 6
         Function evaluations: 27
         Gradient evaluations: 9
func:minimize_function took: 0.0008 sec
func:stochastic_gradient_descent took: 0.8892 sec
Stochastic gradient current func value: 0.6740915373917729
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 51
         Gradient evaluations: 17
func:minimize_function took: 0.0014 sec
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 6
         Function evaluations: 24
         Gradient evaluations: 8
func:minimize_function took: 0.0007 sec
func:stochastic_gradient_descent took: 0.8858 sec
Stochastic gradient current func value: 0.8314656463620618
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 4
         Function evaluations: 27
         Gradient evaluations: 9
func:minimize_function took: 0.0011 sec
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 5
         Function evaluations: 21
         Gradient evaluations: 7
func:minimize_function took: 0.0006 sec
func:stochastic_gradient_descent took: 0.8689 sec
Stochastic gradient current func value: 0.10406560929362686
```

Out[134… ''

In [135…

```
print(averageCG, averageBFGS, averageSGD)
print(numFoundGlobalCG, numFoundGlobalBFGS, numFoundGlobalSGD)
```

```
6.3 6.6 59.9
2 1 0
```

Answer

On average, using the stochastic gradient descent did not take fewer steps when converging to the global minimum, and often times it converged less times than the CG or BFGS algorithms. Likely, the SGD algorithm is getting stuck at a local minimum and thus doesn't converge to the global minimum. Thus, CG and BFGS still outperform SGD

In [121…

```
@timeit
def SGDM(func,first_derivate,starting_point,stepsize,momentum=0.9,tol=1e-5,s
    # evaluate the gradient at starting point
    deriv = first_derivate(starting_point)
    count=0
    visited=[]
    previous_direction = np.zeros(len(starting_point))
    # previous_point = starting_point
    while np.linalg.norm(deriv) > tol and count < 1e5:
        deriv = first_derivate(starting_point)
```

```
        if stochastic_injection>0:
            # formulate a stochastic_deriv that is the same norm as your gra
            stochastic_deriv = np.random.randn(deriv.shape[0])
            stochastic_deriv *= (np.linalg.norm(deriv) / np.linalg.norm(stoc
        else:
            stochastic_deriv=np.zeros(len(starting_point))
        direction=-(deriv+stochastic_injection*stochastic_deriv)
        # direction = momentum * previous_direction + (1 - momentum) * direc
        # calculate new point position
        new_point = starting_point * momentum + (1 - momentum) * direction
        if func(new_point) < func(starting_point):
            # the step makes function evaluation lower - it is a good step.
            stepsize = 1.2 * stepsize
            visited.append(new_point)
            previous_direction = direction
            starting_point = new_point
        else:
            # the step makes function evaluation higher - it is a bad step.
            # if stepsize is too small, clear previous direction because we
            if stepsize<1e-5:
                previous_direction=previous_direction-previous_direction
            else:
                # do the same as SGD here
                # previous_direction = (direction * stepsize) + (previous_di
                stepsize = 0.5 * stepsize

        count+=1
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.a
```

In [120…

```
result = SGDM(three_hump_camel,
              three_hump_camel_derivative,
              starting_point=np.array([0, .1]),
              stepsize = .1,
              momentum=.9,
              tol=1e-5,
              stochastic_injection=1)
print(result)
print(len(result["path"]))
```

```
func:SGDM took: 0.0091 sec
{'x': array([-1.94946476e-05,  8.37689943e-06]), 'evaluation': 6.66950310875
3549e-10, 'path': array([[ 1.09881499e-02,  6.22865336e-02],
       [ 5.37132694e-03,  5.73219479e-02],
       [-5.22009119e-03,  5.21665480e-02],
       [-1.08389313e-02,  2.58764574e-02],
       [-7.90453544e-03,  1.72639722e-02],
       [-5.79673464e-03,  1.19785179e-02],
       [-8.27435071e-03,  7.83533117e-03],
       [-8.27735652e-03,  5.23569261e-03],
       [-7.85529525e-03,  5.04803335e-03],
       [-7.39438044e-03,  3.79706540e-03],
       [-7.37741788e-03,  3.23277158e-03],
       [-7.00711273e-03,  2.66769674e-03],
       [-6.31361723e-03,  2.38992946e-03],
       [-5.67314834e-03,  2.44247174e-03],
       [-5.60234068e-03,  2.32435116e-03],
       [-5.47871314e-03,  2.04086454e-03],
       [-5.18702333e-03,  2.21825059e-03],
       [-4.80336155e-03,  1.85397282e-03],
       [-4.30089045e-03,  1.72065590e-03],
       [-4.21552117e-03,  1.54977227e-03],
       [-3.76316912e-03,  1.46213514e-03],
       [-3.67886686e-03,  1.48441975e-03],
       [-3.56670423e-03,  1.28221084e-03],
       [-3.50056025e-03,  1.26621168e-03],
       [-3.43612521e-03,  1.24760057e-03],
       [-3.06114815e-03,  1.21033667e-03],
       [-2.84006632e-03,  1.28546890e-03],
       [-2.64122991e-03,  1.05987057e-03],
       [-2.56687265e-03,  1.08930325e-03],
       [-2.49755570e-03,  9.34225328e-04],
       [-2.39896103e-03,  1.00041840e-03],
       [-2.20624037e-03,  8.46388297e-04],
       [-2.03284528e-03,  7.21440496e-04],
       [-1.83962936e-03,  7.77805356e-04],
       [-1.66176708e-03,  7.69881171e-04],
       [-1.64523508e-03,  7.33325930e-04],
       [-1.59703407e-03,  7.39887757e-04],
       [-1.48245548e-03,  6.08496598e-04],
       [-1.45344429e-03,  6.05794201e-04],
       [-1.32426123e-03,  5.21631680e-04],
       [-1.29763667e-03,  5.22738259e-04],
       [-1.16551997e-03,  5.15284178e-04],
       [-1.12677061e-03,  4.30915539e-04],
       [-1.07868313e-03,  3.68528746e-04],
       [-1.00286760e-03,  4.15860189e-04],
       [-9.07550962e-04,  3.65281665e-04],
       [-8.33881153e-04,  3.10821823e-04],
       [-8.17920207e-04,  3.10740567e-04],
       [-7.42296798e-04,  2.72269725e-04],
       [-7.01657961e-04,  2.31774738e-04],
       [-6.24991425e-04,  2.17463036e-04],
       [-6.12250902e-04,  2.21756099e-04],
       [-5.47196510e-04,  2.26465838e-04],
       [-5.39648780e-04,  2.13729989e-04],
```

```
       [-4.82977076e-04,  2.01893506e-04],
       [-4.75972360e-04,  1.84627460e-04],
       [-4.27977039e-04,  1.66885168e-04],
       [-4.19621469e-04,  1.66832917e-04],
       [-3.76665407e-04,  1.52441833e-04],
       [-3.45807462e-04,  1.29789797e-04],
       [-3.23398030e-04,  1.09872304e-04],
       [-3.13411216e-04,  1.19199240e-04],
       [-3.07700986e-04,  1.11561413e-04],
       [-2.85425739e-04,  9.51185726e-05],
       [-2.55065584e-04,  8.78641730e-05],
       [-2.27104187e-04,  8.32994379e-05],
       [-2.15268410e-04,  9.09960669e-05],
       [-2.09470150e-04,  7.81577839e-05],
       [-1.86918484e-04,  7.64294764e-05],
       [-1.83315451e-04,  6.83748614e-05],
       [-1.78826715e-04,  7.05939874e-05],
       [-1.60004401e-04,  6.74296423e-05],
       [-1.51254500e-04,  5.60225614e-05],
       [-1.38441403e-04,  6.03358712e-05],
       [-1.34563893e-04,  6.09877080e-05],
       [-1.28930818e-04,  6.21315675e-05],
       [-1.23622145e-04,  6.24630780e-05],
       [-1.12700601e-04,  6.00787638e-05],
       [-1.08984482e-04,  5.91784266e-05],
       [-9.86678557e-05,  5.50484451e-05],
       [-9.21375497e-05,  4.32134306e-05],
       [-9.12591934e-05,  4.11624158e-05],
       [-9.03681469e-05,  3.70572554e-05],
       [-8.53027129e-05,  3.09391174e-05],
       [-8.05418600e-05,  2.63662607e-05],
       [-7.27759479e-05,  2.95388737e-05],
       [-6.85952025e-05,  2.47018970e-05],
       [-6.57727389e-05,  2.14434754e-05],
       [-6.00646387e-05,  2.44529730e-05],
       [-5.78464442e-05,  2.07952260e-05],
       [-5.16506515e-05,  1.94674916e-05],
       [-4.67361791e-05,  2.03828337e-05],
       [-4.33183475e-05,  2.09161312e-05],
       [-4.23589654e-05,  2.06333615e-05],
       [-4.21694051e-05,  1.80987356e-05],
       [-3.97338287e-05,  1.87916969e-05],
       [-3.84009422e-05,  1.88590163e-05],
       [-3.46337485e-05,  1.75651569e-05],
       [-3.20175065e-05,  1.72625923e-05],
       [-3.01351400e-05,  1.69818283e-05],
       [-2.82359375e-05,  1.65407617e-05],
       [-2.80299143e-05,  1.58312678e-05],
       [-2.68998171e-05,  1.22630923e-05],
       [-2.54579722e-05,  1.25230255e-05],
       [-2.40208864e-05,  1.00671902e-05],
       [-2.15539421e-05,  9.57663666e-06],
       [-1.94946476e-05,  8.37689943e-06]]), 'count': 110}
107
```

In [140…

```
averageSGDMSteps = 0
numFoundGlobalSGDM = 0
global_min = np.array([0, 0])
numiters = 10
for i in range(numiters):
    random_floats = np.random.uniform(-2, 2, size=2)
    start = np.array(random_floats)

    SGDM_results = SGDM(three_hump_camel,
                    three_hump_camel_derivative,
                    start,
                    stepsize = .1,
                    momentum = .9,
                    tol=1e-5,
                    stochastic_injection=1)

    averageSGDMSteps += len(SGDM_results['path'])
    print(f"Stochastic gradient momentum current func value: {SGDM_results['
    if np.isclose(SGDM_results["evaluation"], 0):
        numFoundGlobalSGDM += 1

averageSGDMSteps /= numiters
```

```
func:SGDM took: 0.0110 sec
Stochastic gradient momentum current func value: 6.802615626489246e-10
func:SGDM took: 0.0033 sec
Stochastic gradient momentum current func value: 6.965944793996393e-10
func:SGDM took: 0.0030 sec
Stochastic gradient momentum current func value: 6.311801876416017e-10
func:SGDM took: 0.0031 sec
Stochastic gradient momentum current func value: 7.958172354327641e-10
func:SGDM took: 0.0022 sec
Stochastic gradient momentum current func value: 6.745740366872636e-10
func:SGDM took: 0.0027 sec
Stochastic gradient momentum current func value: 5.432246923185867e-10
func:SGDM took: 0.0028 sec
Stochastic gradient momentum current func value: 5.345762873676946e-10
func:SGDM took: 0.0025 sec
Stochastic gradient momentum current func value: 7.005739930152044e-10
func:SGDM took: 0.0022 sec
Stochastic gradient momentum current func value: 6.95394778384492e-10
func:SGDM took: 0.0022 sec
Stochastic gradient momentum current func value: 8.172952479675688e-10
```

In [141…

```
print(averageSGDMSteps)
print(numFoundGlobalSGDM)
```

```
161.1
10
```

3 b)

SGDM still takes more steps on average to converge compared to CG or BFGS, but seems to properly find the global minimum more often. While SGD would often get stuck at a local minima, SGDM seems to more accurately find the true global minimum. However, it does often take more steps than CG or BFGS.

```
In [ ]:
```