Chem277B: Machine Learning Algorithms

Homework assignment #1: Local Optimization Methods

1. Bisection vs. Golden Section.

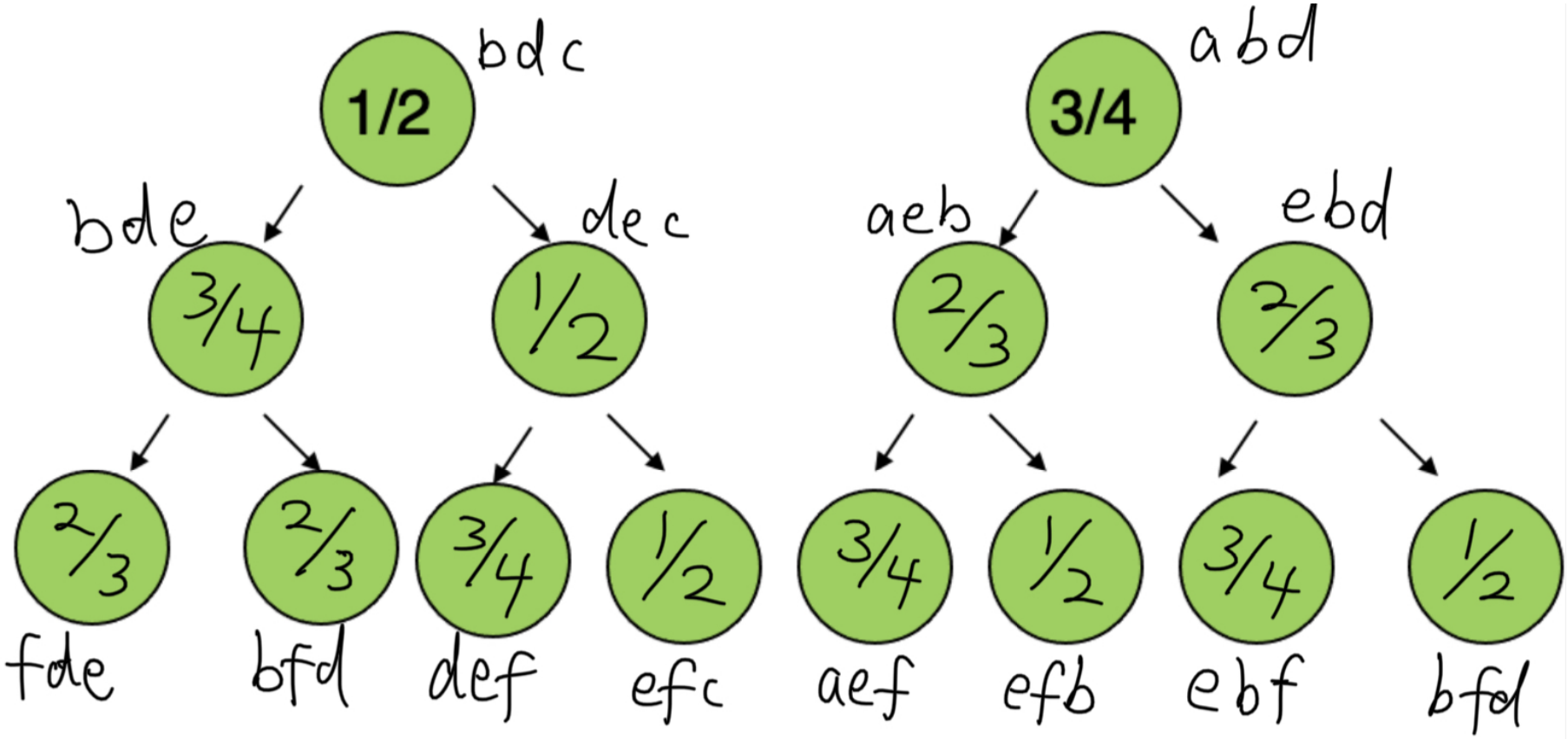
(a) Placing the point e at [a, b] provides higher probability of including a minimum than placing the point at [b, d]. By comparison, it has 100% higher probability. Note it's a compromise between rapid converging and covering as much search space as possible. We prioritize convergence to fast reduction of search space.

(b) In the provided case of f(x), the new search space will be [a, e, b] that covers 1/2 of the original [a, c] space. The reduction is from a previous step of [a, b, d] of 3/4 to 1/2, by 1/4 of the original [a, c] space.

```
Note what's listed above is only one specific scenario based upon the provided f(x), there can be as
many as 4 scenarios, [b, d, e], [d, e, c], [a, e, b], [e, b, d] generated.
```

(c) In the provided case of f(x), the new search space will be [a, e, f], with f at the bisection point of [e, b]. In this case the search space is reduced to 3/8. Again, there will be another 7 different scenarios.

(d)



(e) the average size of search space at step 2 is: 13/32 = 0.40625 ~ 0.406;

```
the average size of search space at step 2 is: 33/128 = 0.2578125 ~ 0.258;
```

(f) Bisection reduces search space by a ratio of (1/2 and 3/4), whereas golden section reduces search space by a ratio of (τ - 1 =) 0.618.

Search Space Size (avg)	Golden Section	Bisection
Step 1	0.618	0.625
Step 2	0.382	0.406
Step 3	0.236	0.258
Step 4	0.146	0.166

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

(a) The new (x, y) point after the first step of steepest descent using a stepsize of 0.1 sits at (0.15, 0.9). It's considered an ideal optimization step because after 1 step the new function value f(xn+1) is lower than the original function value f(xn). But because the gradient has not reached tolerance, I will change the stepsize to 1.2*stepsize in the next step.

The codes that support the above observation is as follows.

In [244…

```
def func_2(starting_point):
    f_2 = starting_point[0]**4 - starting_point[0]**2 + starting_point[1]**2 + 2*starting_point[0]*starting_point[1] -
    return f_2

def dfdx(starting_point):
    return 4 * starting_point[0]**3 - 2*starting_point[0] + 2*starting_point[1]
def dfdy(starting_point):
    return 2* starting_point[1] + 2 * starting_point[0]

starting_point = [1.5, 1.5]
first_derivative = [dfdx(starting_point), dfdy(starting_point)]

print(f"The original function value is {func_2(starting_point)}")
print(f"The first derivatives are {first_derivative}")
print(f"The normalized gradiant value is {LA.norm(first_derivative)}")

stepsize = 0.1
x1 = x0 - stepsize * dfdx(starting_point)
y1 = y0 - stepsize * dfdy(starting_point)
new_point = [x1, y1]

print(new_point)
print(f"The new function value is {func_2(new_point)}")
```

```
The original function value is 7.5625
The first derivatives are [13.5, 6.0]
The normalized gradiant value is 14.773286702694158
[0.1499999999999999, 0.8999999999999999]
The new function value is -0.9419937500000004
```

In [245…

```
import time

from pylab import *
import numpy.linalg as LA
import numpy as np

# to show figures inline
from matplotlib import pyplot as plt
%matplotlib inline

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

@timeit
def steepest_descent(func, first_derivative, starting_point, stepsize, tol):
    # evaluate the gradient at starting point
    if LA.norm(first_derivative) <= tol:
        return {"x":starting_point}

    count=0
    visited=[]
    while LA.norm(first_derivative) > tol and count < 1e6:
        # calculate new point position
        new_point = np.array(starting_point) - stepsize * np.array(first_derivative)

        if func(new_point) < func(starting_point):
            # the step makes function evaluation lower - it is a good step. what do you do?
            starting_point = new_point
            first_derivative = [dfdx(starting_point), dfdy(starting_point)]
            stepsize = stepsize * 1.2
            visited.append(starting_point)
        else:
            # the step makes function evaluation higher - it is a bad step. what do you do?
            stepsize = stepsize * 0.5

        count+=1
    # return the results
    return {"x":starting_point,"evaluation":func(starting_point),"path":np.asarray(visited), "steps": count}
```

In [246…

```
tol = 1e-5
steepest_descent(func_2, first_derivative, starting_point, stepsize, tol)
```

```
func:'steepest_descent' took: 0.0053 sec
```

Out[246]:
```
{'x': array([-0.99999852,  0.99999607]),
 'evaluation': -2.999999999985186,
 'path': array([[ 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],
        [-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]]),
 'steps': 51}
```

(c) Both conjugate gradients (CG) and BFGS perform much better than the steepest descent method. Steepest descent takes 51 steps to converge whereas CG and BFGS each takes 9 and 7 steps to converge.

In [247…
```
from scipy.optimize import minimize

starting_point = [1.5, 1.5]
res_CG = minimize(func_2, starting_point, method='CG', options={'disp': True, 'gtol': 1e-5})
res_BFGS = minimize(func_2, starting_point, method='BFGS', options={'disp': True, 'gtol': 1e-5})
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 9
         Function evaluations: 78
         Gradient evaluations: 26
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 7
         Function evaluations: 24
         Gradient evaluations: 8
```

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

(a) Given the steepest descent algorithm developed above and given the Rosenbrock Banana Function, the loss function converged successfully at a stepsize of 0.1.

The final coordinate is ~[1.0, 1.0]. The function took: 0.0304 sec and 1523 steps to converge.

The codes that support the above claims is as follows.

In [248…
```
def func_3(starting_point):
    f_3 = (1 - starting_point[0])**2 + 10 * (starting_point[1] - starting_point[0]**2)**2
    return f_3

def dfdx(starting_point):
    return 40 * starting_point[0]**3 + (2 - 40 * starting_point[1]) * starting_point[0] - 2
def dfdy(starting_point):
    return 20 * (starting_point[1] - starting_point[0]**2)

starting_point = np.array([-0.5, 1.5])
first_derivative = np.array([dfdx(starting_point), dfdy(starting_point)])
stepsize = 0.1
tol = 1e-5

steepest_descent(func_3, first_derivative, starting_point, stepsize, tol)
```

```
func:'steepest_descent' took: 0.0558 sec
```

Out[248]:
```
{'x': array([0.99999089, 0.99998153]),
 'evaluation': 8.36126679831272e-11,
 'path': array([[-1.05      ,  0.875     ],
        [-0.845175  ,  0.94325   ],
        [-0.91805808,  0.86083548],
        ...,
        [ 0.99999068,  0.99998135],
        [ 0.99999093,  0.99998135],
        [ 0.99999089,  0.99998153]]),
 'steps': 1523}
```

(b) Per the instruction, I created a `random_vec` array by using `np.random.normal` function. I then create a new stochastic derivative from the `random_vec` and the first/existing derivative in every while loop. From the new derivative the gradient descent direction can be confirmed. I'm able to use the new SGD loss function to converge the Rosenbeck Banana Function with a step size of 0.01 in ~2000 steps. The function takes around 0.1 second. The final minimum found is at ~[1.0, 1.0].

Note with the original step size of 0.1, the function doesn't converge and it ends with overweighted scaler error after just a few steps. Result image is as follows. I don't really understand the cause.

```
{'x': array([nan, nan]),
 'evaluation': nan,
 'path': array([[-5.00000000e-01,  1.50000000e+00],
        [-6.02210418e+00, -1.23156819e+00],
        [ 8.83498138e+02,  1.58203747e+02],
        [-2.54342781e+08,  5.35222149e+08],
        [ 1.64421718e+25, -4.31540665e+23],
        [-4.97842345e+74, -9.26618762e+74],
        [             inf,             -inf],
        [             nan,              nan]]),
 'steps': 7,
 'stepsize': 0.00078125}
```

In [249…
```
@timeit
def stochastic_gradient_descent(func,first_derivative,starting_point,stepsize,tol=1e-5,stochastic_injection=1):
    '''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
    if LA.norm(first_derivative) <= tol:
        return {"x":starting_point}

    count=0
    visited=[]
    visited.append(starting_point)
    new_point = starting_point
    deriv = np.array([dfdx(new_point), dfdy(new_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_vec = np.random.normal(0, 1, len(deriv))
            stochastic_deriv = random_vec / LA.norm(random_vec) * LA.norm(deriv)
        else:
            stochastic_deriv = np.zeros(len(new_point))

        direction = -(deriv + stochastic_injection * stochastic_deriv)

        # calculate new point position
        old_point = new_point
        new_point = old_point + stepsize * direction

        deriv = np.array([dfdx(new_point), dfdy(new_point)])
        visited.append(new_point)

        if func(new_point) < func(old_point):
            # the step makes function evaluation lower - it is a good step. what do you do?
            stepsize = stepsize * 1.2
        else:
            # the step makes function evaluation higher - it is a bad step. what do you do?
            stepsize = stepsize * 0.5

        count+=1
    return {"x":new_point,"evaluation":func(new_point),"path":np.asarray(visited),
            "steps": count, "stepsize": stepsize}
```

In [250…

```
starting_point = np.array([-0.5, 1.5])
first_derivative = np.array([dfdx(starting_point), dfdy(starting_point)])
stepsize = 0.01

stochastic_gradient_descent(func_3, first_derivative, starting_point, stepsize, tol=1e-5, stochastic_injection=1)
```

```
func:'stochastic_gradient_descent' took: 0.1187 sec
```

Out[250]:
```
{'x': array([0.99999621, 0.9999921 ]),
 'evaluation': 1.5386440155417994e-11,
 'path': array([[-0.5       ,  1.5       ],
        [-0.48745066,  1.48837115],
        [-0.40493396,  1.39041451],
        ...,
        [ 0.99999517,  0.99999193],
        [ 0.99999597,  0.99999213],
        [ 0.99999621,  0.9999921 ]]),
 'steps': 2122,
 'stepsize': 0.0093161243444614 1}
```

(c) Comparing to my Stochastic Gradient Descent function, it seems both CG and BFGS does much better to find the global minimum. CG takes 20 and BFGS takes 22 iterations.

The data that supports the above claim is as follows.

In [251…
```
res_CD = minimize(func_3, starting_point, method='CG', options={'disp': True, 'gtol': 1e-5})
res_BFGS = minimize(func_3, starting_point, method='BFGS', options={'disp': True, 'gtol': 1e-5})
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 132
         Gradient evaluations: 44
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 22
         Function evaluations: 93
         Gradient evaluations: 31
```

(d) At the given starting point, stepsize, tolerance and taken into account the SGD function is development by myself, with one run we can confirm CG and BFGS are more efficient than the SGD function. I think with the large difference it's reasonable to claim that the result is statistically significant.

(e) From the table below, it seems with the Rosenbrock Banana Function, steepest descent consistently takes ~1500 steps to converge. SGD takes more than 2100 steps. CG and BFGS are both ~20 steps.

Likely with the one minimum situation, adding stochastic vector in the function doesn't help much.

Algorithm	Steps at [-0.5, 1.5]	Steps at [2.0, -1.5]	Steps at [0.0, 0.0]	Average Steps
GD	1523	1501	1473	1499
SGD	1992	2125	2221	2113
CG	20	16	13	16

Algorithm	Steps at [-0.5, 1.5]	Steps at [2.0, -1.5]	Steps at [0.0, 0.0]	Average Steps
BFGS	22	24	11	19

In [252…
```
starting_point = np.array([2.0, -1.5])
first_derivative = np.array([dfdx(starting_point), dfdy(starting_point)])
stepsize = 0.01
tol=1e-5

res_CD = minimize(func_3, starting_point, method='CG', options={'disp': True, 'gtol': 1e-5})
res_BFGS = minimize(func_3, starting_point, method='BFGS', options={'disp': True, 'gtol': 1e-5})
steepest_descent(func_3, first_derivative, starting_point, stepsize, tol)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 16
         Function evaluations: 99
         Gradient evaluations: 33
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 24
         Function evaluations: 99
         Gradient evaluations: 33
func:'steepest_descent' took: 0.0306 sec
```

Out[252]:
```
 {'x': array([0.99998991, 0.99997947]),
  'evaluation': 1.030463618436142e-10,
  'path': array([[-0.21      , -0.95      ],
         [-0.14537736, -0.830708  ],
         [-0.0932184 , -0.70804267],
         ...,
         [ 0.9999896 ,  0.99997936],
         [ 0.99998992,  0.99997932],
         [ 0.99998991,  0.99997947]]),
  'steps': 1501}
```

In [253…
```
stochastic_gradient_descent(func_3, first_derivative, starting_point, stepsize, tol=1e-5, stochastic_injection=1)
```

```
func:'stochastic_gradient_descent' took: 0.1343 sec
```

Out[253]:
```
 {'x': array([0.99998915, 0.9999779 ]),
  'evaluation': 1.19341011157042e-10,
  'path': array([[ 2.        , -1.5       ],
         [ 0.85449064, -3.56608766],
         [ 1.04232869, -3.1409623 ],
         ...,
         [ 0.99998945,  0.9999779 ],
         [ 0.99998945,  0.99997789],
         [ 0.99998915,  0.9999779 ]]),
  'steps': 2026,
  'stepsize': 0.009376926032291498}
```

In [254…
```
starting_point = np.array([0.0, 0.0])
first_derivative = np.array([dfdx(starting_point), dfdy(starting_point)])
stepsize = 0.01
tol=1e-5

res_CD = minimize(func_3, starting_point, method='CG', options={'disp': True, 'gtol': 1e-5})
res_BFGS = minimize(func_3, starting_point, method='BFGS', options={'disp': True, 'gtol': 1e-5})
steepest_descent(func_3, first_derivative, starting_point, stepsize, tol)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 13
         Function evaluations: 90
         Gradient evaluations: 30
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 11
         Function evaluations: 42
         Gradient evaluations: 14
func:'steepest_descent' took: 0.0342 sec
```

Out[254]:
```
 {'x': array([0.99998997, 0.99997962]),
  'evaluation': 1.0163275456779964e-10,
  'path': array([[2.00000000e-02, 0.00000000e+00],
         [4.35161600e-02, 9.60000000e-05],
         [7.10178358e-02, 6.13724980e-04],
         ...,
         [9.99989742e-01, 9.99979433e-01],
         [9.99989992e-01, 9.99979447e-01],
         [9.99989968e-01, 9.99979621e-01]]),
  'steps': 1473}
```

In [255…
```
stochastic_gradient_descent(func_3, first_derivative, starting_point, stepsize, tol=1e-5, stochastic_injection=1)
```

```
func:'stochastic_gradient_descent' took: 0.1232 sec
```

Out[255]:
```
 {'x': array([0.99998916, 0.99997786]),
  'evaluation': 1.1964211830125417e-10,
  'path': array([[ 0.        ,  0.        ],
         [ 0.02397254, -0.0196015 ],
         [ 0.03442043, -0.03473206],
         ...,
         [ 0.99998912,  0.99997788],
         [ 0.99998918,  0.99997785],
         [ 0.99998916,  0.99997786]]),
  'steps': 2120,
  'stepsize': 0.006469530794764869}
```

4. Stochastic Gradient Descent with Momentum (SGDM)

(a) SGD, CG and BFGS algorithms were all able to find global minimum very rapidly. However there's certain chance that SGD converges to a local minimum at [0, 0]. When SGD algorithm finds the global minimum for the Three-Hump Camel function, it was completed with ~ 50 steps and in ~0.002 second, compared to >2000 steps for the Rosenbrock Banana function. CG and BFGS were also completed very fast and all three algorithms got the same global minimum result at [-1.74755205, 0.8737715] with a minimum function value of 0.2986384422578563. From the steps, CG and BFGS are still faster than the SGD algorithm I developed. The result seems significant.

When SGD algorithm doesn't find the global minimum, but the local minimum, it was able to do so with ~50 steps as well.

In [256…

```python
def func_4(starting_point):
    x = starting_point[0]
    y = starting_point[1]
    f_4 = 2 * x**2 - 1.05 * x**4 + x**6 / 6 + x * y + y**2
    return f_4

def derivative(starting_point):
    x = starting_point[0]
    y = starting_point[1]
    dfdx = x**5 - 21 * x**3 / 5 + 4 * x + y
    dfdy = 2 * y + x
    return np.array([dfdx, dfdy])

@timeit
def stochastic_gradient_descent(func, first_derivative, starting_point, stepsize, tol=1e-5, stochastic_injection=1):
    '''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
    if LA.norm(first_derivative) <= tol:
        return {"x":starting_point}

    count=0
    visited=[]
    visited.append(starting_point)
    new_point = starting_point
    deriv = derivative(new_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_vec = np.random.normal(0, 1, len(deriv))
            stochastic_deriv = random_vec / LA.norm(random_vec) * LA.norm(deriv)
        else:
            stochastic_deriv = np.zeros(len(new_point))

        direction = -(deriv + stochastic_injection * stochastic_deriv)

        # calculate new point position
```

```
        old_point = new_point
        new_point = old_point + stepsize * direction

        deriv = derivative(new_point)       # gradient
        visited.append(new_point)

        if func(new_point) < func(old_point):
            # the step makes function evaluation lower - it is a good step. what do you do?
            stepsize = stepsize * 1.2
        else:
            # the step makes function evaluation higher - it is a bad step. what do you do?
            stepsize = stepsize * 0.5

        count+=1
    return {"x":new_point,"evaluation":func(new_point),"path":np.asarray(visited),
            "steps": count, "stepsize": stepsize}

starting_point = np.array([-1.5, -1.5])
first_derivative = derivative(starting_point)
stepsize = 0.1
tol = 1e-5

res_CD = minimize(func_4, starting_point, method='CG', options={'disp': True, 'gtol': 1e-5})
res_BFGS = minimize(func_4, starting_point, method='BFGS', options={'disp': True, 'gtol': 1e-5})
stochastic_gradient_descent(func_4, first_derivative, starting_point, stepsize, tol=1e-5, stochastic_injection=1)
```

```
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: 8
         Function evaluations: 30
         Gradient evaluations: 10
func:'stochastic_gradient_descent' took: 0.0047 sec
```

Out[256]:

```
{'x': array([-3.93067072e-07,  3.50397768e-06]),
 'evaluation': 1.1209564776414913e-11,
 'path': array([[-1.50000000e+00, -1.50000000e+00],
        [-1.26665250e+00, -6.13048575e-01],
        [-1.27833064e+00, -6.10916478e-01],
        [-9.07518124e-01, -7.47846196e-02],
        [-7.56865300e-01,  3.76530013e-01],
        [-6.92017865e-01,  2.19854124e-01],
        [-6.02445951e-01,  4.91013348e-02],
        [-2.67690929e-01, -2.63818583e-01],
        [ 4.55106073e-01,  4.78380569e-01],
        [ 4.65959167e-01,  4.62895988e-01],
        [ 5.59753004e-01,  1.07595133e-01],
        [-2.54205257e-01,  1.84606727e-01],
        [-1.94883034e-01, -1.16883392e-02],
        [ 1.71745219e-01, -2.12209087e-01],
        [ 1.51448135e-01, -2.42695837e-01],
        [-3.31567816e-02, -1.80498297e-01],
        [ 1.79945367e-01, -3.78880207e-02],
        [ 1.51513753e-01,  1.44134726e-02],
        [ 1.29750449e-01,  5.32709698e-02],
        [ 1.08819174e-02, -1.09387438e-01],
        [ 5.94732993e-02, -9.91114612e-02],
        [-5.87039383e-03, -1.36356099e-02],
        [ 5.09405601e-03,  1.38948908e-02],
        [ 9.97066638e-03, -3.13473558e-03],
        [-1.12069907e-03,  1.15454168e-02],
        [ 6.51927732e-03, -7.34946060e-03],
        [-2.03187257e-02,  5.80404491e-04],
        [ 3.51373803e-02, -2.29797246e-04],
        [ 8.72986336e-03, -3.10178827e-02],
        [ 1.85702003e-02, -1.69713301e-02],
        [ 1.86704963e-02, -9.72897608e-03],
        [ 1.67888212e-02, -1.78221780e-02],
        [ 9.68514840e-03, -2.67507494e-02],
        [ 1.53355094e-02, -2.12089504e-02],
        [ 1.61167230e-02, -1.98374218e-02],
        [-3.72174134e-03, -3.81120170e-03],
        [ 7.99514014e-03, -3.53568921e-03],
        [-7.11565832e-04, -4.54787668e-03],
        [ 1.90763007e-03, -8.23627013e-04],
        [ 7.84021906e-04,  5.82993261e-04],
        [ 8.20164841e-04, -3.82813057e-04],
        [ 6.51283246e-04, -9.34560008e-04],
        [-3.14592058e-04,  2.59980536e-04],
        [ 3.14213889e-04,  6.04512314e-04],
        [-2.74892257e-04, -2.78438227e-04],
        [ 1.75157344e-04,  3.92213639e-04],
        [-1.42889091e-04,  5.54492189e-04],
        [ 1.49032976e-04, -8.97659050e-05],
```

```
          [-3.92932035e-05, -3.11809543e-04],
          [-3.86388093e-06,  2.52181457e-05],
          [-7.12368762e-06,  2.55010650e-05],
          [-4.27251227e-07,  1.99627267e-05],
          [-4.52983787e-06,  2.05925257e-05],
          [ 4.44878288e-07,  7.76621547e-06],
          [ 2.30975220e-06,  1.96084269e-06],
          [-4.75622095e-06,  8.93058723e-07],
          [-3.93067072e-07,  3.50397768e-06]]),
  'steps': 56,
  'stepsize': 0.17857724550824372}
```

(b) I tested the algorithms 10 times and summarized the results as follow:

Algorithm	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10	Percentage Global Min
SGDM	No Converge	No Converge	No Converge	No Converge	No Converge	No Converge	No Converge	No Converge	No Converge	No Converge	>50%(?)
SGD	50 steps GM	44 steps GM	51 steps LM	62 steps GM	56 steps LM	48 steps LM	40 steps GM	55 steps LM	51 steps GM	52 steps LM	50%
CG	8 steps LM	8 steps LM	8 steps LM	8 steps LM	8 steps LM	8 steps LM	8 steps LM	8 steps LM	8 steps LM	8 steps LM	0%
BFGS	7 steps LM	7 steps LM	7 steps LM	7 steps LM	7 steps LM	7 steps LM	7 steps LM	7 steps LM	7 steps LM	7 steps LM	0%

From the repeated tests, it seems that SGD might converge to local minimum instead of global minimum. CG and BFGS consistently converge to global minimum with less than 10 iterations each test. SGDM is supposed to converge to global minimum with higher probability. But I wasn't able to implement the algorithm successfully.

In [257…

```python
@timeit
def SGDM(func, first_derivate, starting_point, stepsize, momentum=0.9, tol=1e-5, stochastic_injection=1):
    # evaluate the gradient at starting point
    if LA.norm(first_derivative) <= tol:
        return {"x":starting_point}

    count=0
    visited=[]
    visited.append(starting_point)
    new_point = starting_point
    deriv = derivative(new_point)
    previous_direction = np.zeros(len(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_vec = np.random.normal(0, 1, len(deriv))
            stochastic_deriv = random_vec / LA.norm(random_vec) * LA.norm(deriv)
```

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

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

        # calculate new point position
        old_point = new_point
        new_point = old_point + stepsize * direction

        deriv = derivative(new_point)
        visited.append(new_point)

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

        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 useful direction
            if stepsize<1e-5:
                previous_direction = previous_direction - previous_direction
            else:
                # do the same as SGD here
                stepsize = stepsize * 0.5
        count+=1
    return {"x":new_point,"evaluation":func(new_point),"path":np.asarray(visited),
            "steps": count, "stepsize": stepsize}
```

In [258…

```
starting_point = np.array([-1.5, -1.5])
first_derivative = derivative(starting_point)
stepsize = 0.1
tol = 1e-5

res_CD = minimize(func_4, starting_point, method='CG', options={'disp': True, 'gtol': 1e-5})
res_BFGS = minimize(func_4, starting_point, method='BFGS', options={'disp': True, 'gtol': 1e-5})
stochastic_gradient_descent(func_4, first_derivative, starting_point, stepsize, tol=1e-5, stochastic_injection=1)
```

```
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: 8
         Function evaluations: 30
         Gradient evaluations: 10
func:'stochastic_gradient_descent' took: 0.0052 sec
```

Out[258]:

```
{'x': array([-1.74755166,  0.87377488]),
 'evaluation': 0.2986384422405405,
 'path': array([[-1.5       , -1.5       ],
        [-1.41956151, -1.50914074],
        [-0.95056173, -1.38787343],
        [-1.24447016, -0.80791743],
        [-1.60609545, -0.1432247 ],
        [-1.7881944 ,  0.61235064],
        [-1.71466328,  0.5431495 ],
        [-1.79533275,  0.60319456],
        [-1.78992161,  0.59521725],
        [-1.4543175 ,  0.73916193],
        [-1.47283464,  0.67305082],
        [-1.59891242,  0.82345867],
        [-1.69034062,  0.68049893],
        [-1.67272765,  0.7406713 ],
        [-1.89859909,  0.81568335],
        [-1.65948022,  0.58839114],
        [-1.67929537,  0.71269865],
        [-1.7699313 ,  0.67048829],
        [-1.73359648,  0.74264516],
        [-1.71612122,  0.76337565],
        [-1.72904674,  0.80929521],
        [-1.75624145,  0.80295026],
        [-1.75442136,  0.82211224],
        [-1.74648139,  0.81767937],
        [-1.7404239 ,  0.8365287 ],
        [-1.7517354 ,  0.84827945],
        [-1.74289523,  0.84355829],
        [-1.74391612,  0.84325737],
        [-1.7491039 ,  0.8473448 ],
        [-1.73976969,  0.84921837],
        [-1.74340544,  0.8476352 ],
        [-1.74326315,  0.85254241],
        [-1.74478087,  0.8519078 ],
        [-1.74715343,  0.85775105],
        [-1.74810083,  0.86286658],
        [-1.74492755,  0.86249761],
        [-1.74732377,  0.86437317],
        [-1.74801061,  0.86618024],
        [-1.74559541,  0.86812819],
        [-1.74612557,  0.86749748],
        [-1.74561223,  0.86861742],
        [-1.74633545,  0.87180068],
        [-1.74921574,  0.87415712],
        [-1.74606332,  0.87386895],
        [-1.74612312,  0.87322764],
        [-1.74951962,  0.87445315],
        [-1.74899677,  0.87342246],
        [-1.74708686,  0.87463286],
```

```
          [-1.747483  ,  0.87390048],
          [-1.74750807,  0.87394095],
          [-1.74768724,  0.87389153],
          [-1.74751707,  0.8738399 ],
          [-1.74752479,  0.87385318],
          [-1.74756862,  0.87387539],
          [-1.7475382 ,  0.87386355],
          [-1.74755555,  0.87386989],
          [-1.74756914,  0.87386463],
          [-1.74757353,  0.87384586],
          [-1.74757527,  0.87384239],
          [-1.74754202,  0.87380731],
          [-1.74756158,  0.87382017],
          [-1.74756206,  0.87381971],
          [-1.74756093,  0.87380686],
          [-1.74755715,  0.87379415],
          [-1.747547  ,  0.8737906 ],
          [-1.7475518 ,  0.87379348],
          [-1.74755119,  0.87379292],
          [-1.74755103,  0.87378753],
          [-1.74755227,  0.87378843],
          [-1.74755595,  0.87378798],
          [-1.74755542,  0.87378856],
          [-1.74755559,  0.87378834],
          [-1.74755071,  0.87378543],
          [-1.74755462,  0.87378006],
          [-1.74755373,  0.87377723],
          [-1.74755166,  0.87377488]]),
  'steps': 75,
  'stepsize': 0.1719589671422323}
```

In [259…

```
SGDM(func_4, first_derivative, starting_point, stepsize, momentum=0.9, tol=1e-5, stochastic_injection=1)
```

```
func:'SGDM' took: 2.1719 sec
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_19311/362927481.py:10: RuntimeWarning: invalid value encount
ered in double_scalars
  dfdx = x**5 - 21 * x**3 / 5 + 4 * x + y
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_19311/362927481.py:11: RuntimeWarning: invalid value encount
ered in double_scalars
  dfdy = 2 * y + x
/var/folders/pk/syhjl001491bwlx124c6hv_h0000gn/T/ipykernel_19311/362927481.py:4: RuntimeWarning: invalid value encounte
red in double_scalars
  f_4 = 2 * x**2 - 1.05 * x**4 + x**6 / 6 + x * y + y**2
```

Out[259]:
```
{'x': array([ inf, -inf]),
 'evaluation': nan,
 'path': array([[-1.50000000e+00, -1.50000000e+00],
        [-1.07899217e+00, -1.37033201e+00],
        [-1.20077386e-01, -3.44751137e-01],
        ...,
        [ 5.08052730e+06,  7.26254494e+07],
        [-2.99010328e+31,  1.65325576e+31],
        [            inf,            -inf]]),
 'steps': 30577,
 'stepsize': 0.0014417784565480128}
```