Homework #1 Answers

Chongye Feng

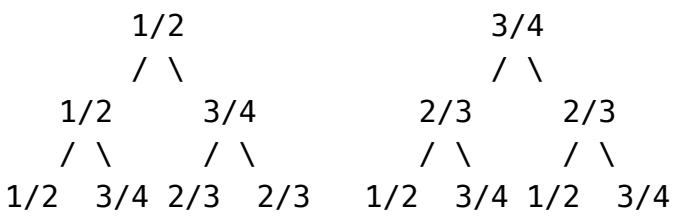
In [1]:
```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
%matplotlib inline
```

In [2]:
```
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
```

Q_1

- a: because it increases the chance of finding the minimum value of the function by evenly dividing the larger interval in half. This allows for a more thorough search of the potential minimum in comparison to only searching a portion of the interval. Additionally, bisecting the larger interval also reduces the overall size of the interval, making it easier to converge towards the minimum value.
- b:
  - If point e is placed in [a,b]:
    1. if f(e) > f(b), we could reduce the interval length from 0.75 to 0.5, a 1/3 reduction;
    2. if f(e) < f(b), we could reduce the interval length from 0.75 to 0.5, a 1/3 reduction;
    3. The average reduction would be 1/3.
  - If point e is placed in [b,d]:
    1. if f(e) > f(b), we could reduce the interval length from 0.75 to 0.625, a 1/6 reduction;
    2. if f(e) < f(b), we could reduce the interval length from 0.75 to 0.25, a 1/2 reduction;
    3. The average reduction would be 1/3.
- c: after step 2, the new interval would be, if e is placed in [a,b], [a,e,b] or [e,b,d] in the size of 0.5. And both of the oiginal bisection point is in the center of the interval, which is in a similar situation of step 1. Take [a,e,b] as an example, both [a,e] and [e,b] are in the same size and no difference without searching:
  - if f is in [a,e], f=0.125;
  - if f is in [e,b], f=0.375;
  - after getting f, the new reduced interval would be in size 0.25 or 0.375;

- d:



- e:

In [3]:
```
# step 1
(0.5 + 0.75) / 2
```

Out[3]: 0.625

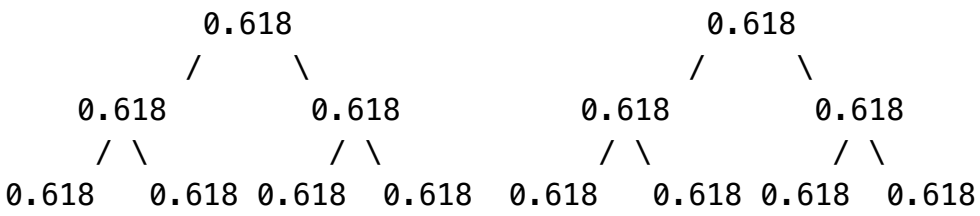
In [4]:
```
# step 2
(0.5 + 0.75 + 4/3) / 4
```

Out[4]: 0.6458333333333333

In [5]:
```
# step 3
((0.5 + 0.75) * 3 + 4/3) / 8
```

Out[5]: 0.6354166666666666

- f: For Golden Section, the ratio of size of interval of the previous step is 0.618, no change.



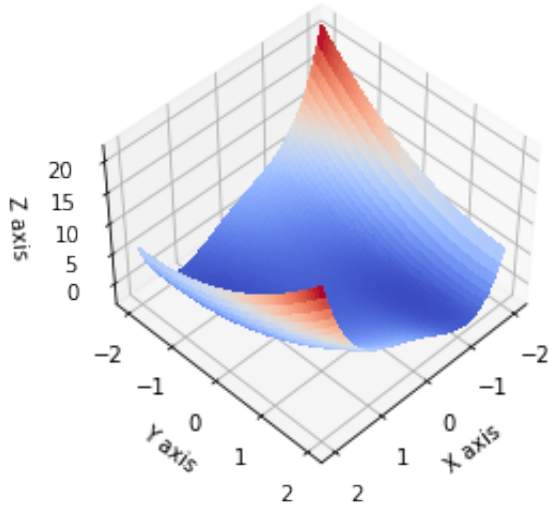
Comparing with bisection:

Method:	Bisection	Golden Section	Diff
1:	0.625	0.618	0.007
2:	0.646	0.618	0.028
3:	0.635	0.618	0.017

It shows that Golden Section is reducing more searching interval than Bisection method.

Q_2

In [6]:
```
def func_q2(coord):
    x = coord[0]
    y = coord[1]
    return x**4 - x**2 + y**2 + 2*x*y -2

x = np.linspace(-2, 2, 200)
y = np.linspace(-2, 2, 200)
X, Y = np.meshgrid(x, y)
Z = func_q2((X, Y))

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=Fa
ax.set_xlabel('X axis')
ax.set_ylabel('Y axis')
ax.set_zlabel('Z axis')
ax.view_init(45,45)
plt.show()
```



- a:

In [7]:
```
Z
```

Out[7]:
```
array([[22.        , 21.36601181, 20.75022147, ...,  5.07182951,
         5.52681583,  6.        ],
       [21.83960001, 21.20641988, 20.5914376 , ...,  5.07061742,
         5.5264118 ,  6.00040403],
       [21.68000808, 21.04763601, 20.43346179, ...,  5.07021339,
         5.52681583,  6.00161612],
       ...,
       [ 6.00161612,  5.52681583,  5.07021339, ..., 20.43346179,
        21.04763601, 21.68000808],
       [ 6.00040403,  5.5264118 ,  5.07061742, ..., 20.5914376 ,
        21.20641988, 21.83960001],
       [ 6.        ,  5.52681583,  5.07182951, ..., 20.75022147,
        21.36601181, 22.        ]])
```

In [8]:
```
# point f(1.5,1.5)
func_q2((1.5,1.5))
```

Out[8]:
```
7.5625
```

In [9]:
```
def first_derivate_q2(coord):
    x = coord[0]
    y = coord[1]
    dx = 4 * x**3 - 2 * x + 2 * y
    dy = 2 * y + 2 * x
    return (dx, dy)
```

In [10]:
```
first_derivate_q2((1.5,1.5))
```

Out[10]:
```
(13.5, 6.0)
```

So, the first step would be 0.1 * (13.5, 6), which is (1.35, 0.6).

And the x_0 (1.5, 1.5) with the limit is ([-2,2], [-2,2]).

So, the x_1 would be (1.5 - 1.35, 1.5 - 0.6), which is (0.15, 0.90).

To judge if it is a good step:

In [11]:
```
func_q2((0.15,0.9)) < func_q2((1.5,1.5))
```

Out[11]:
```
True
```

f(x_1) < f(x_0)

So that it is a good step.

Based on this outcome, the step size would be increased to (0.12 * first_derivative(x,y)). And, the start point would be updated to x_1 (0.15, 0.90).

- b:

In [12]:
```
from pylab import *
import numpy.linalg as LA

@timeit
def steepest_descent(func, first_derivate, starting_point, stepsize, t
    # evaluate the gradient at starting point
    deriv = first_derivate(starting_point)
    count = 0
    visited = []

    while LA.norm(deriv) > tol and count < 1e6:
        # calculate new point position
        new_x = starting_point[0] - stepsize * deriv[0]
        new_y = starting_point[1] - stepsize * deriv[1]

        new_point = (new_x,new_y)
        visited.append(new_point)

        if func(new_point) < func(starting_point):
            # the step makes function evaluation lower - it is a good
            starting_point = new_point
            deriv = first_derivate(starting_point)
            stepsize *= 1.2
            count += 1
        else:
            # the step makes function evaluation higher - it is a bad
            stepsize *= 0.5

    # return the results
    return {"x": starting_point, "evaluation": func(starting_point), "
```

In [13]:
```
steepest_descent(func_q2,first_derivate_q2,(1.5,1.5),0.1,1e-5)
```

```
func:'steepest_descent' took: 0.0007 sec
```

Out[13]:
```
{'x': (-0.999998515461034, 0.9999960662218339),
 'evaluation': -2.999999999985186,
 'n_steps': 41}
```

In my case, the method took 41 steps, from step 1, to converge to point (-0.999998515461034, 0.9999960662218339) with an evaluation of -2.999999999985186. It tooks 0.0007 second.

- c:

In [14]:
```
from scipy.optimize import minimize
```

In [15]:
```
@timeit
def timming_BFGS():
    res = minimize(func_q2, (1.5,1.5), method='BFGS', options={'disp':

    print("Minimum of the function:", res.fun)
    print("Location of the minimum:", res.x)

timming_BFGS()
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 7
         Function evaluations: 24
         Gradient evaluations: 8
Minimum of the function: -2.9999999999998255
Location of the minimum: [ 0.99999979 -0.9999998 ]
func:'timming_BFGS' took: 0.0030 sec
```

In [16]:
```
@timeit
def timming_CG():
    res = minimize(func_q2, (1.5,1.5), method='CG', options={'disp': T

    print("Minimum of the function:", res.fun)
    print("Location of the minimum:", res.x)

timming_CG()
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 9
         Function evaluations: 78
         Gradient evaluations: 26
Minimum of the function: -2.99999999999959
Location of the minimum: [-0.99999984  0.99999929]
func:'timming_CG' took: 0.0045 sec
```

In [17]:
```
steepest_descent(func_q2,first_derivate_q2,(1.5,1.5),0.1,1e-5)
```

```
func:'steepest_descent' took: 0.0006 sec
```

Out[17]:
```
{'x': (-0.999998515461034, 0.9999960662218339),
 'evaluation': -2.999999999985186,
 'n_steps': 41}
```

By comparing the three methods, BFGS and CG would take less steps than steepest descent, 7 and 9 steps compared with 41 steps regardly. So, in terms of step efficiency, BFGS method is better than CG method, steepest descent method is the worst in this case.

Q_3

- a:

In [18]:
```
def func_q3(coord):
    x = coord[0]
    y = coord[1]
    return ((1 - x)**2 + 10 * (y - x**2)**2)
```

In [19]:
```
def first_derivate_q3(coord):
    x = coord[0]
    y = coord[1]
    dx = 40 * x**3 + (2 - 40 * y) * x - 2
    dy = -20 * x**2 + 20 * y
    return (dx, dy)
```

In [20]:
```
steepest_descent(func_q3,first_derivate_q3,(-0.5,1.5),0.1,1e-5)
```

```
func:'steepest_descent' took: 0.0129 sec
```

Out[20]:
```
{'x': (0.9999908923749649, 0.9999815271830772),
 'evaluation': 8.361266798228901e-11,
 'n_steps': 1204}
```

It took steepest descent method 1204 steps to converge to point (1,1), (0.9999908923749649, 0.9999815271830772).

- b:

In [21]:
```
@timeit
def stochastic_gradient_descent(func,first_derivate,starting_point,ste
    '''stochastic_injection: controls the magnitude of stochasticity (
        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 LA.norm(deriv) > tol and count < 1e5:
        if stochastic_injection>0:
            # formulate a stochastic_deriv that is the same norm as yo
            stochastic_deriv = np.random.randn(len(starting_point))
            stochastic_deriv = stochastic_deriv / LA.norm(stochastic_d
        else:
            stochastic_deriv=np.zeros(len(starting_point))

        direction=-(deriv+stochastic_injection*stochastic_deriv)
        # calculate new point position
        new_x = starting_point[0] + stepsize * direction[0]
        new_y = starting_point[1] + stepsize * direction[1]

        new_point = (new_x,new_y)
        visited.append(new_point)

        if func(new_point) < func(starting_point):
            # the step makes function evaluation lower - it is a good
            starting_point = new_point
            deriv = first_derivate(starting_point)
            stepsize *= 1.2
            count+=1

        else:
            # the step makes function evaluation higher - it is a bad
            stepsize *= 0.5

    return {"x":starting_point,"evaluation":func(starting_point), "n_s
```

In [22]:
```
stochastic_gradient_descent(func_q3,first_derivate_q3,(-0.5,1.5),0.1)
```

```
func:'stochastic_gradient_descent' took: 0.0530 sec
```

Out[22]:
```
{'x': (0.9999892914635049, 0.9999781884938099),
 'evaluation': 1.1622943410450859e-10,
 'n_steps': 2219}
```

The SGD method took 2219 steps and 0.0977 second to converge. The final point is (1,1), (0.9999893578953681, 0.9999783136244784), with an evaluation at 1.1487267905535701e-10.

- c:

In [23]:
```
@timeit
def timming_BFGS_q3():
    res = minimize(func_q3, (-0.5,1.5), method='BFGS', options={'disp'

    print("Minimum of the function:", res.fun)
    print("Location of the minimum:", res.x)

timming_BFGS_q3()
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 22
         Function evaluations: 93
         Gradient evaluations: 31
Minimum of the function: 1.6856836004019217e-13
Location of the minimum: [0.99999959 0.99999917]
func:'timming_BFGS_q3' took: 0.0060 sec
```

In [24]:
```
@timeit
def timming_CG_q3():
    res = minimize(func_q3, (-0.5,1.5), method='CG', options={'disp':

    print("Minimum of the function:", res.fun)
    print("Location of the minimum:", res.x)

timming_CG_q3()
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 132
         Gradient evaluations: 44
Minimum of the function: 2.0711814827200667e-13
Location of the minimum: [0.99999955 0.99999908]
func:'timming_CG_q3' took: 0.0077 sec
```

In [25]:
```
stochastic_gradient_descent(func_q3,first_derivate_q3,(-0.5,1.5),0.1)
```

```
func:'stochastic_gradient_descent' took: 0.0499 sec
```

Out[25]:
```
{'x': (0.9999890227080755, 0.9999775843184513),
 'evaluation': 1.2262816028160373e-10,
 'n_steps': 2052}
```

By comparing the three methods, BFGS and CG would take less steps than SGD method, 22 and 20 steps compared with 2052 steps regardly. So, in terms of step efficiency, CG method is better than BFGS method, steepest descent method is the worst in this case.

- d:

No, it is not possible to draw a firm conclusion on the outcome of optimization algorithms such as Stochastic Gradient Descent (SGD), Conjugate Gradient (CG), or BFGS with just one run of each method.

To draw a robust conclusion on the outcome of different optimization algorithms, it is necessary to run multiple trials, with different initial conditions, such as random seed, tolerance, etc.

I think it is better to perform a systematic and thorough evaluation of different optimization algorithms on a diverse range of test functions and real-world datasets, in order to obtain a comprehensive understanding of their performance and limitations.

- e:

In [26]:
```
steepest_descent(func_q3,first_derivate_q3,(-0.5,-1),0.1,1e-5)
```

```
func:'steepest_descent' took: 0.0097 sec
```

Out[26]:
```
{'x': (0.9999895865363, 0.9999786856407166),
 'evaluation': 1.1081718190344482e-10,
 'n_steps': 1169}
```

In [27]:
```
stochastic_gradient_descent(func_q3,first_derivate_q3,(-0.5,1),0.1)
```

```
func:'stochastic_gradient_descent' took: 0.0541 sec
```

Out[27]:
```
{'x': (0.9999890657975701, 0.9999776672274304),
 'evaluation': 1.2171426698632107e-10,
 'n_steps': 2299}
```

Non-Stochastic (Steepest Descent) method:

n_steps	-1	-0.5
-1	1523	1479
-0.5	1479	1067
0	1182	1522
0.5	1724	1523
1	1541	1503

Stochastic Gradient Descent method:

n_steps	-1	-0.5
-1	3224	2157
-0.5	2235	2305
0	3831	2263
0.5	2226	2176
1	2298	2027

It shows that the non-stochastic (steepest descent) method would need less steps to converge, in general, than the Stochastic Gradient Descent method.

At starting point (-1,0) and (-0.5,-0.5), because it is no "hill climbing" between starting point and local minimum point, the path is shorter comparing with the other points with the same x_coordinate, the steps needed is the lest among the points with same x_coordinate. It is observed in the non-stochastic (steepest descent) method. On the other hand, the Stochastic Gradient Descent method does not have this observation.

Q_4

In [28]:
```
def func_q4(coord):
    x = coord[0]
    y = coord[1]
    return (2 * x**2 - 1.05 * x**4 + (1/6) * x**6 + x * y + y**2)
```

In [29]:
```
def first_derivate_q4(coord):
    x = coord[0]
    y = coord[1]
    dx = x**5 - 4.2 * x**3 + 4 * x + y
    dy = x + 2 * y
    return (dx, dy)
```

- a:

In [30]:
```
stochastic_gradient_descent(func_q4,first_derivate_q4,(-1.5,-1.5),0.1)
```

```
func:'stochastic_gradient_descent' took: 0.0021 sec
```

Out[30]:
```
{'x': (-2.1110997248668292e-06, 4.159456360117833e-06),
 'evaluation': 1.7433534130929103e-11,
 'n_steps': 64}
```

In [31]:
```
@timeit
def timming_BFGS_q4():
    res = minimize(func_q4, (-1.5,-1.5), method='BFGS', options={'disp

    print("Minimum of the function:", res.fun)
    print("Location of the minimum:", res.x)

timming_BFGS_q4()
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
Minimum of the function: 0.29863844223686
Location of the minimum: [-1.74755234  0.87377616]
func:'timming_BFGS_q4' took: 0.0028 sec
```

In [32]:
```
@timeit
def timming_CG_q4():
    res = minimize(func_q4, (-1.5,-1.5), method='CG', options={'disp':

    print("Minimum of the function:", res.fun)
    print("Location of the minimum:", res.x)

timming_CG_q4()
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
Minimum of the function: 0.2986384422397135
Location of the minimum: [-1.74755166  0.87377618]
func:'timming_CG_q4' took: 0.0044 sec
```

Comparing with 3e, the Stochastic Gradient Descent method required less steps in this case, 64 steps vs 2000+ steps. But the BFGS and CG methods have better performance than Stochastic Gradient Descent method, 8, 7 steps vs 69 steps.

Some important observation, the Stochastic Gradient Descent method did not converge into the global minimum every time, and BFGS and CG methods converge into the same global minimum every time. It is possible for the Stochastic Gradient Descent method be traped by the local minimum.

- b:

In [33]:
```
@timeit
def SGDM(func, first_derivate, starting_point, stepsize, momentum=0.9,
    count = 0
    visited = []
    deriv = first_derivate(starting_point)
    previous_direction = np.zeros(len(starting_point))
    while LA.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 yo
            stochastic_deriv = np.random.normal(0, LA.norm(deriv), len
        else:
            stochastic_deriv = np.zeros(len(starting_point))
        direction = (deriv + stochastic_injection * stochastic_deriv)
        # use previous direction to compute the current direction
        direction = momentum * previous_direction + direction
        # calculate new point position
        new_point = starting_point - stepsize * direction
        if func(new_point) < func(starting_point):
            # the step makes function evaluation lower - it is a good
            starting_point = new_point
            previous_direction = direction
            stepsize *= 1.2
            count += 1
        else:
            # the step makes function evaluation higher - it is a bad
            # if stepsize is too small, clear previous direction becau
            if stepsize < 1e-5:
                previous_direction = np.zeros(len(starting_point))
            else:
                # decrease stepsize by factor of 2 and clear previous
                stepsize *= 0.5
                previous_direction = np.zeros(len(starting_point))
        visited.append(starting_point)

    return {"x": starting_point, "evaluation": func(starting_point), "
```

In [34]:
```
SGDM(func_q4,first_derivate_q4,(-1.5,-1.5),0.1)
```

```
func:'SGDM' took: 0.0023 sec
```

Out[34]:
```
{'x': array([-1.74755258,  0.87377916]),
 'evaluation': 0.2986384422454149,
 'n_steps': 58}
```

I don't get a better result using SGDM compared to SGD, CG or BFGS in finding the global minimum in terms of fewer steps.

It took 0.0023 second and 58 steps to converge into the global minimum.

I did several runs of the SGDM method. Sometimes it would be trapped by the local minimum, similar with SGD method. And the steps it takes on average is less than the SGD method, around 58 steps vs around 64 steps.

In this Three-Hump Camel function, the rank of steps of methods to converge (from most to least) is SGD > SGDM > BFGS > CG.