In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import time
from scipy.optimize import minimize
```

In [2]:
```
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
```

Question 1: Bisection VS Golden Section

1. a. Placing e in the bisector of the larger interval [a,b] is better than placing it in [b,d] because it reduces the search space by a larger amount, thus making it a more effective reduction.

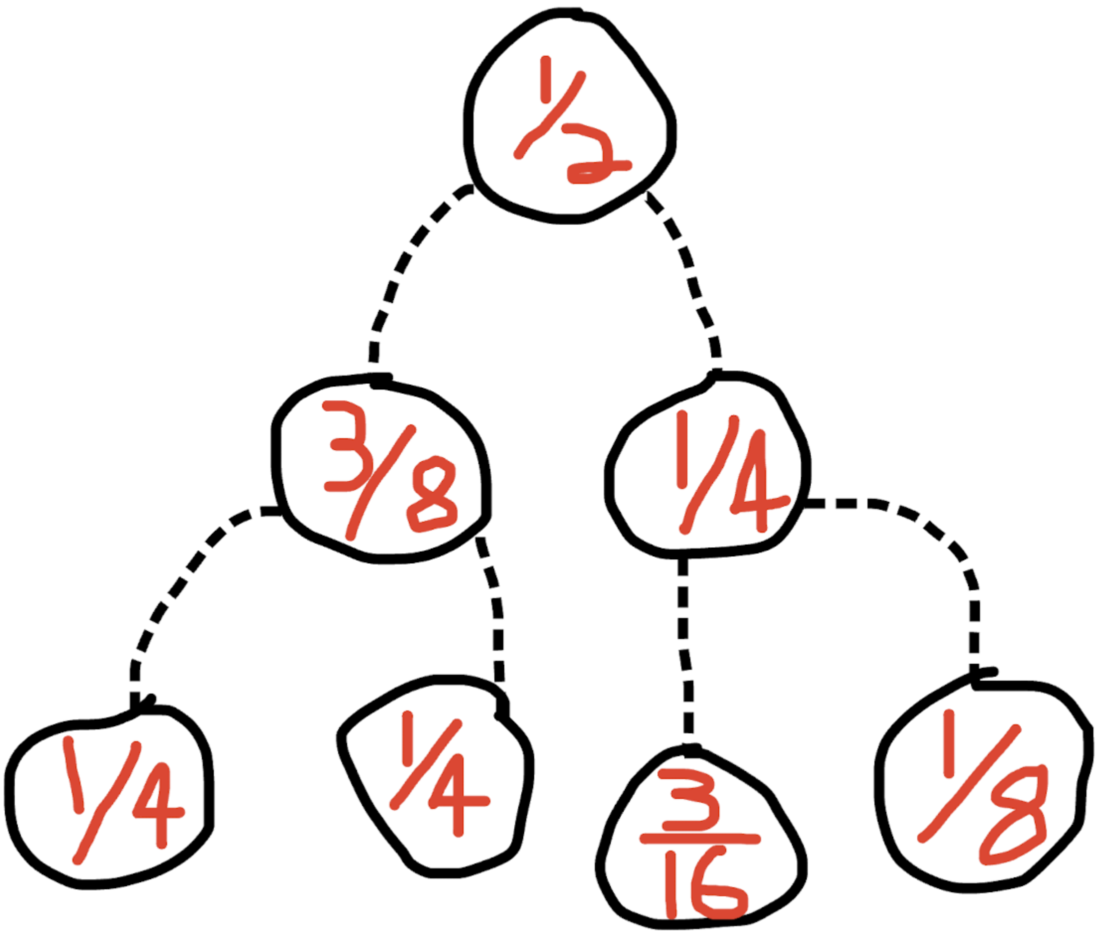
   b. The new interval is [a, e, b] and the search space is reduced by 0.25.

   e. Step 2: 13/32 or 0.4063 Step 3: 33/128 or 0.2578

1d

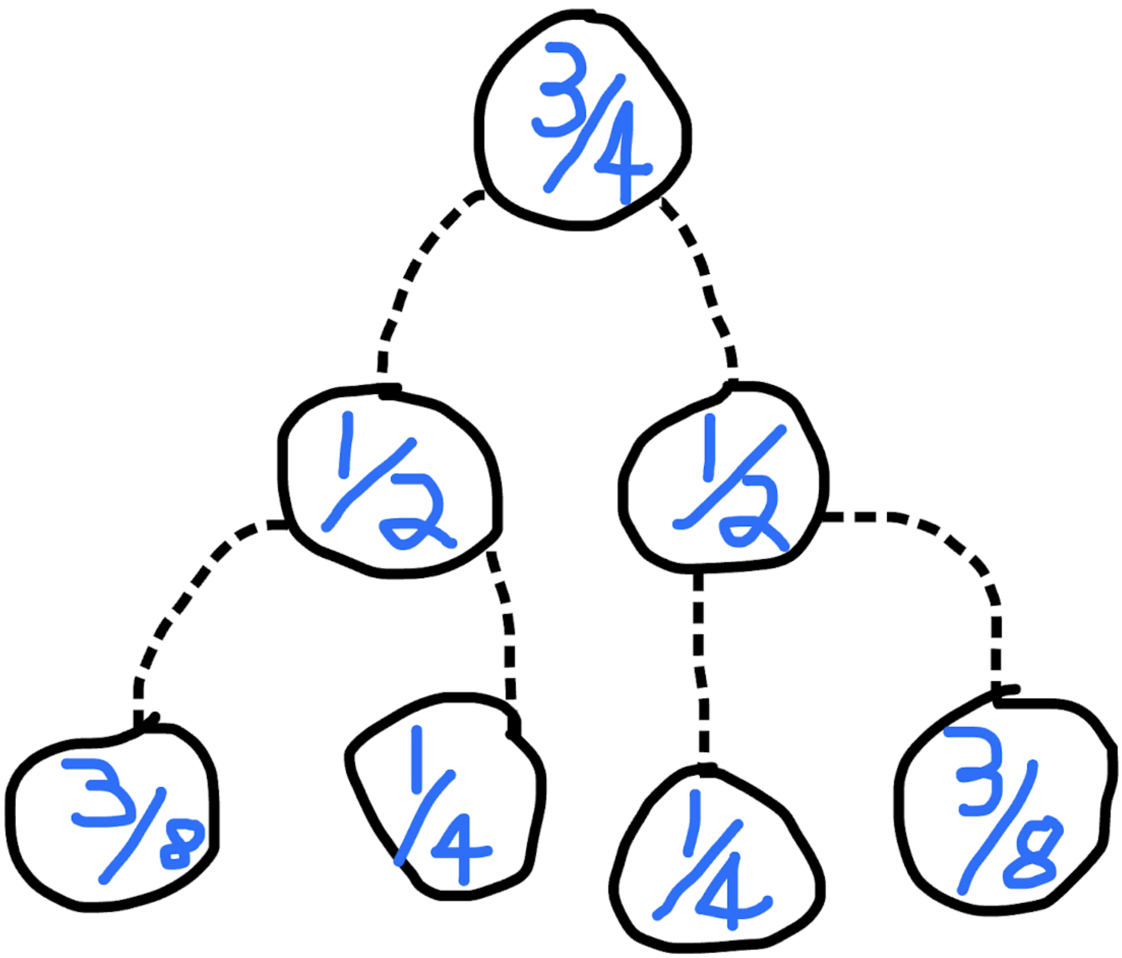
In [3]:
```
from IPython.display import Image
Image(filename='image1.png')
```

Out[3]:



In [4]:
```
from IPython.display import Image
Image(filename='image2.png')
```

Out[4]:



Are my trees ugly? Yes. But they were made with love <3

1f

Golden section converges to a minimum (or maximum) faster than bisection does. In theory this makes sense, but when I typed out a table to compare I saw that bisection (best-case scenario) does better than golden section. I guess golden section is still better because any other case of bisection does not converge as fast, and realistically you will probably rarely get best-case scenario when performing bisection.

In [5]:
```
from IPython.display import Image
Image(filename='image3.png')
```

Out[5]:

A	B	C	D	E
Number of steps	Golden section	Bisection (best case)	Bisection (ok case)	Bisection (worst case)
1	0.618	0.5	0.5	0.75
2	0.382	0.25	0.375	0.5
3	0.236	0.125	0.25	0.375

Question 2: Steepest Descent

In [6]:
```
def function3d(point):
    """
    This function calculates the value of a function of x and y  at a certai
    Parameters
    ----------
    x, y: the point to evaluate the function at

    Return
    ------
     the function value at the point
    """
    x, y = point

    return x**4 - x**2 + y**2 + 2*x*y - 2
```

In [7]:
```
def first_derivative(point):
    """
    This function calculates the first derivative of a function at a given p

    Parameters
    ----------
    point: list. Not really a list, more like comma separated variables
    A point, x,y

    Returns
    -------
    first_derivative: np.array
    The first derivative (gradient) at the point entered
    """
    x, y = point
    dfdx = (4 * x**3) - (2*x) + 2*y
    dfdy = (2*y) + 2*x

    return np.array([dfdx, dfdy])
```

2a

In [8]:
```
@timeit

def steepest_descent_one_step(func, gradient, x0, alpha=0.1):
    """
    This function performs a single step of the steepest descent algorithm.
    Parameters
    ----------
    func:
    function to optimize

    gradient: np.array
    gradient of a function

    x0: np.array
    starting point
```

```
    num_iter: int
    number of times the for loop will run

    alpha: float
    stepsize

    Returns
    -------
    x1 : new position
    value: funciton evaluated at that x1
    """
    point_searched = []
    # calculate the initial function value
    prev_value = func(x0)

    # print search progress and keep track of the points searched
    print(f"searching at {x0} with function value {prev_value}")
    point_searched.append(x0)

    # calculate the gradient at the current point
    g = gradient(x0)

    # calculate next point and its function value
    x1 = x0 - g * alpha
    #value = func(x1)

    return print(f"New position is {x1}")
```

In [9]:
```
steepest_descent_one_step(function3d, first_derivative, np.array([1.5, 1.5]))
```

```
searching at [1.5 1.5] with function value 7.5625
New position is [0.15 0.9 ]
func:'steepest_descent_one_step' took: 0.0006 sec
```

This is a good optimization step because it moves towards the minimum. We will increase the stepsize * 1.2 in the next step.

2b

In [10]:
```
from pylab import *
import numpy.linalg as LA

@timeit
def steepest_descent(func,func_gradient,x0, alpha,tol):
    """
    This function finds a local minimum using the steepest descent method.
    Parameters
    ----------
    func:
    The function whose minima we plan to find (inputted as a function)

    func_gradient:
    first_derivative of func
```

```
    x0: np.array
    position from which we start searching for the minima

    alpha: float
    step size

    tol: float
    tolerance value

    Returns
    -------
    Starting point: entered x0
    Evaluation: value of function at x0
    Path to minimum: lists all points evaluated on the way to minumum
    Steps to converge: counts number of steps until local mimimum is reachec
    """
    count= 0
    visited =[x0]
    deriv = func_gradient(x0)

    while LA.norm(deriv) > tol and count < 1e6:
        # calculate new point position
        x1 = x0 - deriv * alpha

        if func(x1) < func(x0):
            # Check if new value is less than previous value. If so, this is
            # and start the next search step from the current value
            alpha = alpha * 1.2
            x0 = x1
            deriv = func_gradient(x0)
            visited.append(x1)

        else:
            # If new_value > prev_value, we are moving in the wrong directic
            # and redo that step starting from the previous value.
            alpha = alpha * 0.5
            #visited.append(x0)

        count+=1
    return {"Starting point":x0,"Evaluation":func(x0),"Path to minimum":np.a
```

In [11]:

```
print(steepest_descent(function3d, first_derivative, np.array([-1.5, 1.5]),
```

```
func:'steepest_descent' took: 0.0005 sec
{'Starting point': array([-1.00000072,  1.0000014 ]), 'Evaluation': -2.99999
9999997453, 'Path to minimum': array([[-1.5       ,  1.5       ],
       [-0.75      ,  1.5       ],
       [-1.0875    ,  1.32      ],
       [-1.04004413,  1.25304   ],
       [-1.05492903,  1.17942863],
       [-1.00779561,  1.12779615],
       [-1.05181525,  1.0680762 ],
       [-0.98988976,  1.06322071],
       [-1.03043727,  1.03694491],
       [-1.00445425,  1.03554582],
       [-1.00784819,  1.02752454],
       [-1.00410623,  1.021433  ],
       [-1.00440363,  1.01499603],
       [-1.00122119,  1.01027389],
       [-1.00344611,  1.005431  ],
       [-0.99963579,  1.00479389],
       [-1.00218339,  1.00280712],
       [-1.00030255,  1.00266297],
       [-1.00062139,  1.00200836],
       [-1.00025502,  1.00154679],
       [-1.00036336,  1.00103092],
       [-0.99998637,  1.00071101],
       [-1.0002104 ,  1.00050266],
       [-1.00002076,  1.00040182],
       [-1.00014415,  1.00024404],
       [-1.00002569,  1.00021923],
       [-1.00005275,  1.00016153],
       [-1.00001618,  1.00012262],
       [-1.00003409,  1.00007692],
       [-0.99998591,  1.00005486],
       [-1.00002464,  1.00003355],
       [-0.99999138,  1.00003024],
       [-1.0000077 ,  1.0000216 ],
       [-1.00000319,  1.00001789],
       [-1.00000381,  1.00001317],
       [-1.00000155,  1.00000957],
       [-1.00000239,  1.00000587],
       [-0.99999901,  1.00000395],
       [-1.00000196,  1.00000231],
       [-0.99999897,  1.00000217],
       [-1.00000072,  1.0000014 ]]), 'Steps to converge': 41}
```

This function took 41 good steps to converge. The total number of steps taken was 51.

2c

In [12]:

```
# Conjugate gradient
x0 = np.array([-1.5, 1.5])

CG_method = minimize(function3d, x0, method='CG', options={'gtol': 1e-5, 'di
print(CG_method)
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 6
         Function evaluations: 39
         Gradient evaluations: 13
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: -2.9999999999997273
       x: [-1.000e+00  1.000e+00]
     nit: 6
     jac: [ 2.414e-06  5.364e-07]
    nfev: 39
    njev: 13
```

In [13]:
```
# BFGS
x0 = np.array([-1.5, 1.5])

BFGS_method = minimize(function3d, x0, method='BFGS', options={'gtol': 1e-5,
print(BFGS_method)
```

```
Optimization terminated successfully.
         Current function value: -3.000000
         Iterations: 7
         Function evaluations: 27
         Gradient evaluations: 9
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: -2.999999999999986
        x: [-1.000e+00  1.000e+00]
      nit: 7
      jac: [ 4.172e-07  2.384e-07]
 hess_inv: [[ 1.244e-01 -1.270e-01]
            [-1.270e-01  6.174e-01]]
     nfev: 27
     njev: 9
```

In terms of number of steps, both conjugate gradient (CG) and BFGS are more efficient than steepest descent. CG is only a bit more efficident (39 steps instead of 41) whie BFGS takes only 27 steps.

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

In [14]:
```
def rosenbrock_function3d(point):
    """
    This function calculates the value of a function of x and y  at a certai
    Parameters
    ----------
    x, y: the point to evaluate the function at

    Return
    ------
```

```
     the function value at the point
    """
    x, y = point

    return (1 - x)**2 + 10* (y- (x**2))**2
```

In [15]:
```
def rosenbrock_gradient(point):
    """
    This function calculates the first derivative of a function at a given p
    Parameters
    ----------
    point: list. Not really a list, more like comma separated variables
    A point, x,y

    Returns
    -------
    first_derivative: np.array
    The first derivative (gradient) at the point entered
    """
    x, y = point
    dfdx = -2 *(1-x) - 40 * x *(y-x**2)
    dfdy = 20 * (y-x**2)

    return np.array([dfdx, dfdy])
```

3a

In [16]:
```
steepest_descent(rosenbrock_function3d, rosenbrock_gradient,np.array([-0.5,
```

```
func:'steepest_descent' took: 0.0093 sec
```

Out[16]:
```
{'Starting point': array([0.99999089, 0.99998153]),
 'Evaluation': 8.361266796946337e-11,
 'Path to minimum': array([[-0.5       ,  1.5       ],
        [-1.05      ,  0.875     ],
        [-0.845175  ,  0.94325   ],
        ...,
        [ 0.99999068,  0.99998135],
        [ 0.99999093,  0.99998135],
        [ 0.99999089,  0.99998153]]),
 'Steps to converge': 1205}
```

3b

In [17]:
```
@timeit

def stochastic_gradient_descent(func,func_gradient,x0,alpha=0.1,tol=1e-5,sto
    '''
    This function finds a local minimum using the stochastic gradient methoc
    Parameters
    ----------
    func:
    The function whose minima we plan to find (inputted as a function)

    func_gradient:
```

```
    first_derivative of func

    x0: np.array
    position from which we start searching for the minima

    alpha: float
    step size

    tol: float
    tolerance value

    stochastic_injection: 0 or 1
    controls the magnitude of stochasticity (multiplied with stochastic_deri
    0 for no stochasticity, equivalent to SD.

    Returns
    -------
    Starting point: entered x0
    Evaluation: value of function at x0
    Path to minimum: lists all points evaluated on the way to minumum
    Steps to converge: counts number of steps until local mimimum is reachec
    '''
    # evaluate the gradient at starting point
    deriv = func_gradient(x0)

    count=0
    visited=[x0]
    while LA.norm(deriv) > tol and count < 1e5:
        if stochastic_injection>0:
            # formulate a stochastic_deriv that is the same norm as your gra
            #dim = deriv.shape
            stochastic_deriv= np.random.random(2) * 2 - 1
            stochastic_norm = LA.norm(stochastic_deriv)
            stochastic_deriv = stochastic_deriv / stochastic_norm * LA.norm(
        else:
            stochastic_deriv=np.zeros(len(x0))
        direction=-(deriv + stochastic_injection * stochastic_deriv)
        # calculate new point position
        x1 = x0 - deriv * alpha

        if func(x1) < func(x0):
            # Check if new value is less than previous value. If so, this is
            # and start the next search step from the current value
            alpha = alpha * 1.2
            x0 = x1
            visited.append(x1)
            deriv = func_gradient(x1)
            #print(f'good step {x1}')

        else:
            # If new_value > prev_value, we are moving in the wrong directic
            # and redo that step starting from the previous value.
            alpha = alpha * 0.5
            #print(f'bad step {x1}')
```

```
        count+=1
    return {"x":x0,"evaluation":func(x0),"path":np.asarray(visited), "Number
```

In [18]:
```
stochastic_gradient_descent(rosenbrock_function3d, rosenbrock_gradient, np.a
```

```
func:'stochastic_gradient_descent' took: 0.0183 sec
```

Out[18]:
```
{'x': array([0.99999089, 0.99998153]),
 'evaluation': 8.361266796946337e-11,
 'path': array([[-0.5       ,  1.5       ],
        [-1.05      ,  0.875     ],
        [-0.845175  ,  0.94325   ],
        ...,
        [ 0.99999068,  0.99998135],
        [ 0.99999093,  0.99998135],
        [ 0.99999089,  0.99998153]]),
 'Number of steps': 1205}
```

3c

In [19]:
```
x0 = np.array([-0.5, 1.5])

CG_rosenbrock = minimize(rosenbrock_function3d, x0, method='CG', options={'g
print(CG_rosenbrock)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 20
         Function evaluations: 132
         Gradient evaluations: 44
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 2.0711221375743512e-13
       x: [ 1.000e+00  1.000e+00]
     nit: 20
     jac: [ 4.992e-08 -2.474e-08]
    nfev: 132
    njev: 44
```

In [20]:
```
BFGS_rosenbrock = minimize(rosenbrock_function3d, x0, method='BFGS', options
print(BFGS_rosenbrock)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 22
         Function evaluations: 93
         Gradient evaluations: 31
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 1.6857105436734322e-13
        x: [ 1.000e+00  1.000e+00]
      nit: 22
      jac: [ 1.153e-07 -1.294e-08]
 hess_inv: [[ 5.099e-01  1.020e+00]
            [ 1.020e+00  2.089e+00]]
     nfev: 93
     njev: 31
```

In the case of the Rosenbrock Banana Function, CG and BFGS are far better than SGD. SGD found the minimum in ~1200 steps, while CG and BFGS took 132 and 93, respectively.

3d

No. Because of the stochasticity, number of steps with SGD will vary largely and so you need to take an average value after a few runs for comparison.

3e

I found the performance of SGD to be more consistent than that of steepest descent. For example, at point (-1.5, 1.5), SGD and steepest descent took ~1200 steps to converge. However, as the numbers got bigger the difference in step size grew. At point (-150, 150), SGD took ~79000 steps to converge while steepest descent took ~490000. At (-1500, 1500), SGD still took ~79000 steps while steepest descent took ~790000.

However when comparing CD and BFGS with SGD, their performances (in terms of step sizes) did not increase as quickly as the step size of steepest descent did.

Question 4: Stochastic Gradient Descent with Momentum (SGDM)

In [21]:
```
def momentum_function3d(point):
    """
    This function calculates the value of a function of x and y  at a certai
    Parameters
    ----------
    x, y: the point to evaluate the function at

    Return
    ------
     the function value at the point
```

```
    """
    x, y = point

    return 2*x**2 - 1.05*x**4 + x**6/6 + x*y + y**2
```

In [22]:
```
def momentum_gradient(point):
    """
    This function calculates the first derivative of a function at a given p
    Parameters
    ----------
    point: list. Not really a list, more like comma separated variables
    A point, x,y

    Returns
    -------
    first_derivative: np.array
    The first derivative (gradient) at the point entered
    """
    x, y = point
    dfdx = 4*x - 4.2*x**3 + x**5 + y
    dfdy = x + 2*y

    return np.array([dfdx, dfdy])
```

4a

In [23]:
```
stochastic_gradient_descent(momentum_function3d, momentum_gradient, np.array
```

```
func:'stochastic_gradient_descent' took: 0.0020 sec
```

Out[23]:
```
{'x': array([-1.74755328,  0.87377865]),
 'evaluation': 0.29863844224600855,
 'path': array([[-1.5       ,  1.5       ],
        [-1.708125  ,  1.35      ],
        [-1.81711465,  1.230975  ],
        [-1.72366291,  1.13811871],
        [-1.81648029,  1.04263383],
        [-1.73077839,  1.01476596],
        [-1.77260058,  0.97759624],
        [-1.73965321,  0.95033542],
        [-1.77022043,  0.92148765],
        [-1.74397071,  0.91366684],
        [-1.75467953,  0.90291347],
        [-1.74553402,  0.89499619],
        [-1.7540083 ,  0.88673796],
        [-1.74656205,  0.88456827],
        [-1.74961089,  0.88154912],
        [-1.74682193,  0.87938454],
        [-1.74960838,  0.87708366],
        [-1.74708789,  0.87655687],
        [-1.74825522,  0.8757213 ],
        [-1.74715493,  0.87519094],
        [-1.74777788,  0.87486877],
        [-1.74758021,  0.87463399],
        [-1.74765485,  0.87439135],
        [-1.74754601,  0.87419677],
        [-1.74764902,  0.87402131],
        [-1.74753349,  0.87397242],
        [-1.74759689,  0.87391111],
        [-1.74752418,  0.8738708 ],
        [-1.74757107,  0.87384747],
        [-1.74755094,  0.87383152],
        [-1.74756213,  0.87381419],
        [-1.74754713,  0.87380191],
        [-1.74755705,  0.8737956 ],
        [-1.74755201,  0.87379104],
        [-1.74755504,  0.87378622],
        [-1.74755067,  0.87378288],
        [-1.74755379,  0.87378114],
        [-1.74755205,  0.87377996],
        [-1.74755328,  0.87377865]]),
 'Number of steps': 39}
```

In [24]:
```
x0 = np.array([-1.5, -1.5])

CG_momentum = minimize(momentum_function3d, x0, method='CG', options={'gtol'
print(CG_momentum)
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 7
         Function evaluations: 63
         Gradient evaluations: 21
 message: Optimization terminated successfully.
 success: True
  status: 0
     fun: 0.29863844223965763
       x: [-1.748e+00  8.738e-01]
     nit: 7
     jac: [ 8.404e-06  7.227e-07]
    nfev: 63
    njev: 21
```

In [25]:
```
BFGS_momentum = minimize(momentum_function3d, x0, method='BFGS', options={'g
print(BFGS_momentum)
```

```
Optimization terminated successfully.
         Current function value: 0.298638
         Iterations: 8
         Function evaluations: 30
         Gradient evaluations: 10
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 0.29863844223686065
        x: [-1.748e+00  8.738e-01]
      nit: 8
      jac: [ 1.341e-07 -7.451e-09]
 hess_inv: [[ 8.569e-02 -4.290e-02]
            [-4.290e-02  5.109e-01]]
     nfev: 30
     njev: 10
```

On average, stochastic gradient descent did better than conjugate gradients (~40 steps VS ~60 steps). Stochastic gradient and BFGS had roughly the same performance (~40 steps and ~30 steps)

4b

In [26]:
```
@timeit

def SGDM(func,func_gradient,x0,alpha=0.1,gamma=0.9,tol=1e-5,stochastic_injec
    '''
    This function finds a local minimum using the stochastic gradient methoc
    Parameters
    ----------
    func:
    The function whose minima we plan to find (inputted as a function)

    func_gradient:
    first_derivative of func

    x0: np.array
```

```
    position from which we start searching for the minima

    alpha: float
    step size

    gamma: float
    momentum value

    tol: float
    tolerance value

    stochastic_injection: 0 or 1
    controls the magnitude of stochasticity (multiplied with stochastic_deri
    0 for no stochasticity, equivalent to SD.

    Returns
    -------
    Starting point: entered x0
    Evaluation: value of function at x0
    Path to minimum: lists all points evaluated on the way to minumum
    Steps to converge: counts number of steps until local mimimum is reached
    '''
    # evaluate the gradient at starting point
    deriv = func_gradient(x0)

    count=0
    visited=[x0]
    while LA.norm(deriv) > tol and count < 1e5:
        if stochastic_injection>0:
            # formulate a stochastic_deriv that is the same norm as your gra
            stochastic_deriv= np.random.random(2) * 2 - 1
            stochastic_norm = LA.norm(stochastic_deriv)
            stochastic_deriv = stochastic_deriv / stochastic_norm * LA.norm(
        else:
            stochastic_deriv=np.zeros(len(x0))
        if count == 0:
            previous_direction = -deriv
        direction=-(deriv+stochastic_injection*stochastic_deriv + gamma * pr
        x1 = x0 + alpha * direction

        if func(x1) < func(x0):
            # Check if new value is less than previous value. If so, this is
            # and start the next search step from the current value
            alpha = alpha * 1.2
            x0 = x1
            visited.append(x1)
            deriv = func_gradient(x0)

        else:
            # If new_value > prev_value, we are moving in the wrong directio
            # and redo that step starting from the previous value.
            if alpha<1e-5:
                previous_direction=previous_direction-previous_direction
            else:
                # do the same as SGD here
                alpha = alpha * 0.5
```

```
        count+=1
    return {"x":x0,"Evaluation":func(x0),"Path":np.asarray(visited), "Number
```

In [27]:

```
SGDM(momentum_function3d, momentum_gradient, np.array([-1.5, -1.5]))
```

```
func:'SGDM' took: 0.0030 sec
```

```
Out[27]: {'x': array([-1.74755242,  0.87377317]),
          'Evaluation': 0.2986384422461035,
          'Path': array([[-1.5       , -1.5       ],
                 [-1.42887291, -0.99991264],
                 [-1.52341299, -0.95558559],
                 [-1.51358025, -0.93314037],
                 [-1.50819735, -0.90514635],
                 [-1.51171835, -0.87149957],
                 [-1.51193333, -0.8702979 ],
                 [-1.5138259 , -0.86970886],
                 [-1.51209007, -0.86936447],
                 [-1.51013788, -0.86869735],
                 [-1.50914783, -0.86659327],
                 [-1.50780145, -0.86603561],
                 [-1.50810051, -0.86426464],
                 [-1.50881599, -0.86328878],
                 [-1.50874209, -0.86205446],
                 [-1.50871703, -0.86196794],
                 [-1.50859369, -0.86197152],
                 [-1.50861021, -0.86193535],
                 [-1.50862084, -0.86188384],
                 [-1.50860418, -0.86182104],
                 [-1.50856939, -0.86180588],
                 [-1.50857772, -0.86180438],
                 [-1.50862888, -0.86175653],
                 [-1.50857728, -0.86163747],
                 [-1.50853016, -0.8614841 ],
                 [-1.50859024, -0.86131583],
                 [-1.5086977 , -0.86120214],
                 [-1.50868289, -0.86091939],
                 [-1.50877758, -0.8606204 ],
                 [-1.50864942, -0.86058901],
                 [-1.50865986, -0.86010272],
                 [-1.5083494 , -0.85988323],
                 [-1.50825608, -0.85918581],
                 [-1.50857053, -0.85853138],
                 [-1.50877006, -0.85847055],
                 [-1.50849573, -0.85843156],
                 [-1.50899962, -0.85724743],
                 [-1.50935077, -0.85561289],
                 [-1.51029808, -0.8547077 ],
                 [-1.51118549, -0.85268783],
                 [-1.50958643, -0.85082981],
                 [-1.50861662, -0.8506436 ],
                 [-1.50822936, -0.85064252],
                 [-1.51060204, -0.84818799],
                 [-1.513403  , -0.84562414],
                 [-1.50988777, -0.83994568],
                 [-1.50739991, -0.8315602 ],
                 [-1.51145361, -0.82912219],
                 [-1.51563708, -0.82707445],
                 [-1.51740927, -0.82671246],
                 [-1.52416621, -0.81224243],
                 [-1.52789236, -0.79137261],
                 [-1.5139853 , -0.77658175],
                 [-1.50102341, -0.7515267 ],
```

```
       [-1.50348395, -0.75121575],
       [-1.51786004, -0.74459859],
       [-1.54127613, -0.72579468],
       [-1.52719251, -0.66603555],
       [-1.51817571, -0.66527007],
       [-1.53778436, -0.58585119],
       [-1.56126312, -0.49547162],
       [-1.52436188, -0.39937429],
       [-1.47632375, -0.36966609],
       [-1.50214622, -0.23476685],
       [-1.44060314, -0.16033728],
       [-1.4122792 , -0.14722699],
       [-1.37314195,  0.0151895 ],
       [-1.4798359 ,  0.15702739],
       [-1.6047083 ,  0.33006197],
       [-1.73508535,  0.49289655],
       [-1.77033475,  0.63790684],
       [-1.64515048,  0.78308844],
       [-1.76231657,  0.83695996],
       [-1.74609625,  0.86784473],
       [-1.74631626,  0.87044695],
       [-1.7476742 ,  0.87151245],
       [-1.74729029,  0.87140865],
       [-1.74700886,  0.87212268],
       [-1.74809577,  0.87290828],
       [-1.74702504,  0.87323309],
       [-1.74733609,  0.87281383],
       [-1.74728185,  0.87308691],
       [-1.7474133 ,  0.87351685],
       [-1.74759802,  0.87362149],
       [-1.74753497,  0.87372373],
       [-1.74754153,  0.87373789],
       [-1.74754202,  0.87374692],
       [-1.74754606,  0.87374306],
       [-1.74754335,  0.87374985],
       [-1.74755902,  0.8737628 ],
       [-1.74755931,  0.87377075],
       [-1.74754841,  0.87376586],
       [-1.74755381,  0.87377095],
       [-1.74755244,  0.87377015],
       [-1.74755277,  0.87377151],
       [-1.74755084,  0.87377256],
       [-1.74755109,  0.87377346],
       [-1.74755242,  0.87377317]]),
 'Number of step to converge': 98}
```

4b

No, I did not get a better result when using SGDM. When comparing number of steps, SGD took the fewest number of steps to find global minimum. This was unexpected, as I thought that SGD with momentum would perform better. Since the momentum takes previous good steps into consideration, I thought momentum would serve as a guiding force to move the search in the right direction.