Forum Code Workbook

Optimization Algorithms LBA

QUESTIONS?

We'll use the Slack channel, #formal-analyses-class, for Q&A so please check this channel as you work on the assignment. Post your questions before the due date and come to office hours for longer discussions.

OVERVIEW

This assignment is a hands-on algorithms experience! You'll begin by exploring a straightforward simplified optimization scenario, for which you'll define the optimization problem and write code to solve it. Next, you'll take your algorithmic thinking skills to the real world by writing a natural language algorithm to solve the optimization problem in a more practical scenario. Lastly, you'll get a chance to test out your algorithm in the city!

You'll notice some "*Optional Challenge*" problems throughout the assignment to challenge yourself. These will only be scored (4 or 5) if they are completed correctly with thorough explanation. If you attempt an optional challenge but do not succeed, you will not be penalized with a low score. Remember that you *must* include an explanation and interpretation for optional problems to be scored.

This is an individual assignment. We will be checking for similarities among submissions and will take plagiarism seriously.

FORMAT

Complete the assignment in the Forum Code Workbook. It is pre-formatted for you and doesn't require installing any additional programming software or downloading any extra resources. You can code directly within the workbook! Read this article from the Forum Help Center for help with how to open and submit the workbook. However, note that an internet connection is required to save your progress and submit your work. If you are having technical difficulties working within or submitting the Forum Code Workbook, we suggest reaching out to helpdesk@minerva.edu (and copy your instructor) for technical assistance.

- Run all code cells, in order, before submitting your work.
- Stay within the word limits for each question. You may be scored down if you consistently exceed the suggested word count.
- Any external resources from outside of class need to be cited following APA conventions.
- For HC guidelines, scroll to the end of these instructions.

PART 1: Hill Climbing in Python

The code below is adapted from an in-class activity in FA session 7. It is a simple "hill-climbing" algorithm (except that it is descending rather than climbing) with the goal of reaching the lowest point on an error surface. Focus only on the `optimize` and `step` functions toward the end; the rest of the code contains helper functions to make plots, displayed only for reference.

Code Cell 1 of 8 - Read Only

In [3]

```
# MOST OF THIS CODE IS PROVIDED ONLY FOR REFERENCE, NOT TO BE FOCUSED ON IN THIS ASSIGNMENT. SCROLL DOWN TO THE 'OPTIMIZE' AND 'STEP' FUNCTIONS NEAR THE END.
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from matplotlib.ticker import MultipleLocator, AutoMinorLocator
plt.rcParams.update({'font.size': 12})

def error(a, b):
   # defines the function from here https://www.youtube.com/watch?v=1i8muvzZkPw
    func = 3*(1 - a)**2*np.exp(-a**2-(b+1)**2) - 10*(a/5-a**3-b**5)*np.exp(-a**2-b**2) - (1/3)*np.exp(-b**2-(a+1)**2)
    return -func + 7.8

def get_xyz(f, a_limits=(-3, 3), b_limits=(-3, 3)):
    # sets up an xy grid with associated z values for 3d plotting
    a = np.linspace(a_limits[0], a_limits[1], num=100)
    b = np.linspace(b_limits[0], b_limits[1], num=100)

    mesh = np.meshgrid(a, b)
    z = np.vectorize(f)(mesh[0], mesh[1])

```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Ready | |

```
23
24 def fmt(x):
25     return '{0:.{1}f}'.format(x, 1)
26
27 def cmap_map(function, cmap):
28     """ Applies function (which should operate on vectors of shape 3: [r, g,
   b]), on colormap cmap.
29     This routine will break any discontinuous points in a colormap.
30     """
31     cdict = cmap._segmentdata
32     step_dict = {}
33     # Firt get the list of points where the segments start or end
34     for key in ('red', 'green', 'blue'):
35         step_dict[key] = list(map(lambda x: x[0], cdict[key]))
36     step_list = sum(step_dict.values(), [])
37     step_list = np.array(list(set(step_list)))
38     # Then compute the LUT, and apply the function to the LUT
39     reduced_cmap = lambda step : np.array(cmap(step)[0:3])
40     old_LUT = np.array(list(map(reduced_cmap, step_list)))
41     new_LUT = np.array(list(map(function, old_LUT)))
42     # Now try to make a minimal segment definition of the new LUT
43     cdict = {}
44     for i, key in enumerate(['red','green','blue']):
45         this_cdict = {}
46         for j, step in enumerate(step_list):
47             if step in step_dict[key]:
48                 this_cdict[step] = new_LUT[j, i]
49             elif new_LUT[j,i] != old_LUT[j, i]:
50                 this_cdict[step] = new_LUT[j, i]
51         colorvector = list(map(lambda x: x + (x[1], ), this_cdict.items()))
52         colorvector.sort()
53         cdict[key] = colorvector
54
55     return matplotlib.colors.LinearSegmentedColormap('colormap',cdict,1024)
56
57 dark_jet = cmap_map(lambda x: x*0.75, matplotlib.cm.jet)
58
59 def _plot2d(f, a_limits=(-3, 3), b_limits=(-3, 3), max=(0, 1.6)):
60     # creates a 2d contour plot
61     a, b, z = get_xyz(f, a_limits, b_limits)
```

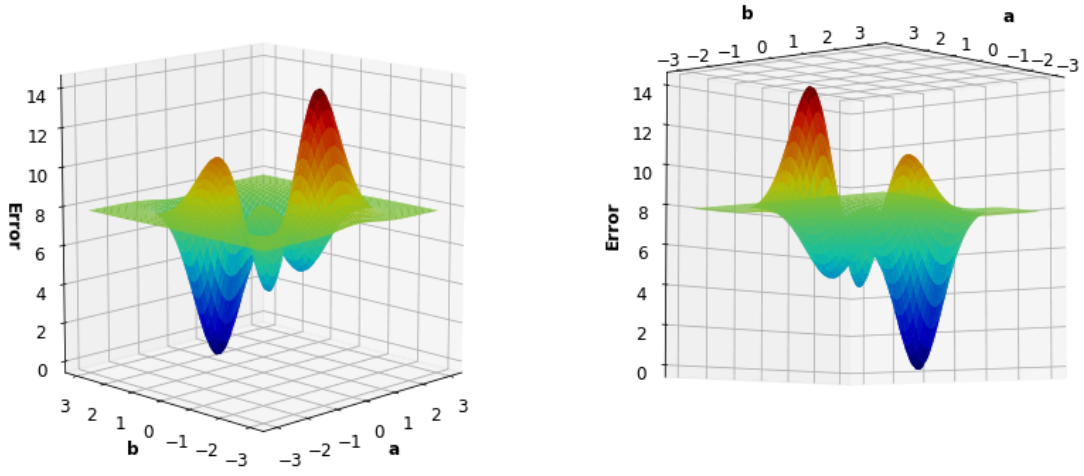
```
    fig = plt.figure(figsize=(8,8))
    ax = plt.axes()

    z = ax.contour(a, b, z, levels=22, cmap=dark_jet)
    ax.clabel(z, inline=True, fontsize=12,  fmt=fmt)

    ax.grid(True)
    ax.grid(True, which="minor", linestyle="dotted")

    ax.xaxis.set_major_locator(MultipleLocator(1))
    ax.xaxis.set_minor_locator(AutoMinorLocator(5))

    ax.yaxis.set_major_locator(MultipleLocator(1))
    ax.yaxis.set_minor_locator(AutoMinorLocator(5))

    ax.plot(*max, marker="*", markersize=18)

    return fig, ax

def plot2d(f, a_limits=(-3, 3), b_limits=(-3, 3), max=(0, 1.6)):
    # plots the optimization landscape
    _, ax = _plot2d(f, a_limits, b_limits, max)
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    plt.show()

def plot3d(f, a_limits=(-3, 3), b_limits=(-3, 3)):
   # same, but plots in 3d with the same colorscheme
    a, b, z = get_xyz(f, a_limits, b_limits)

    fig = plt.figure(figsize=(14, 7))
    ax = fig.add_subplot(1, 2, 1, projection='3d')
    #ax = plt.axes(projection='3d')
    ax.view_init(12,-135)
    ax.plot_surface(a, b, z, cmap=dark_jet, edgecolor='none')
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    ax.set_zlabel('Error',fontweight='bold')

    ax = fig.add_subplot(1, 2, 2, projection='3d')
```

```
    ax.plot_surface(a, b, z, cmap=dark_jet, edgecolor='none')
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    ax.set_zlabel('Error',fontweight='bold')

    plt.show()

def plot_optimization(move, start_a, start_b, stepsize, max_steps=1500):
    # given the starting points, creates a 2d contour plot, and error plot, and prints the end results
    a, b, errors = optimize(move, start_a, start_b, stepsize=stepsize, max_steps=1500)

    fig, ax = _plot2d(error)
    ax.plot(a, b, marker='o',  markersize=3, color='gray')
    ax.plot(start_a, start_b, marker='o',  markersize=9, color='green')
    ax.plot(a[-1], b[-1], marker='v',  markersize=9, color='black')
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    plt.show()
    print("Final location (a,b) =", (round(a[-1],2),round(b[-1],2)))
    print("Final value of Error(a,b) =", round(errors[-1],3))

    plt.xlabel('Steps')
    plt.ylabel('Error')
    plt.plot(errors, marker='.')
    plt.show()

### THE FUNCTIONS BELOW SHOULD BE THE FOCAL POINT OF YOUR ANALYSIS

def step(a, b, stepsize):
    # how to take a step
    current_error = error(a, b)

    if error(a + stepsize, b) < current_error:
        a = a + stepsize
    elif error(a, b + stepsize) < current_error:
        b = b + stepsize
    elif error(a - stepsize, b) < current_error:
        a = a - stepsize
    elif error(a, b - stepsize) < current_error:
```

```
143         b = b - stepsize
144
145     return a, b
146
147 def optimize(move, start_a, start_b, stepsize, max_steps=150):
148     # takes defined "move" function, and implements an optimization run
149     all_a = [start_a]
150     all_b = [start_b]
151     all_error = [error(start_a,start_b)]
152
153     for i in range(max_steps):
154         a, b = move(start_a, start_b, stepsize)
155
156         if a == start_a and b == start_b:
157             break
158
159         start_a, start_b = a, b
160
161         all_a.append(start_a)
162         all_b.append(start_b)
163         all_error.append(error(start_a,start_b))
164
165     return all_a, all_b, all_error
166
167 plot3d(error)
```

Run Code

Out [3]



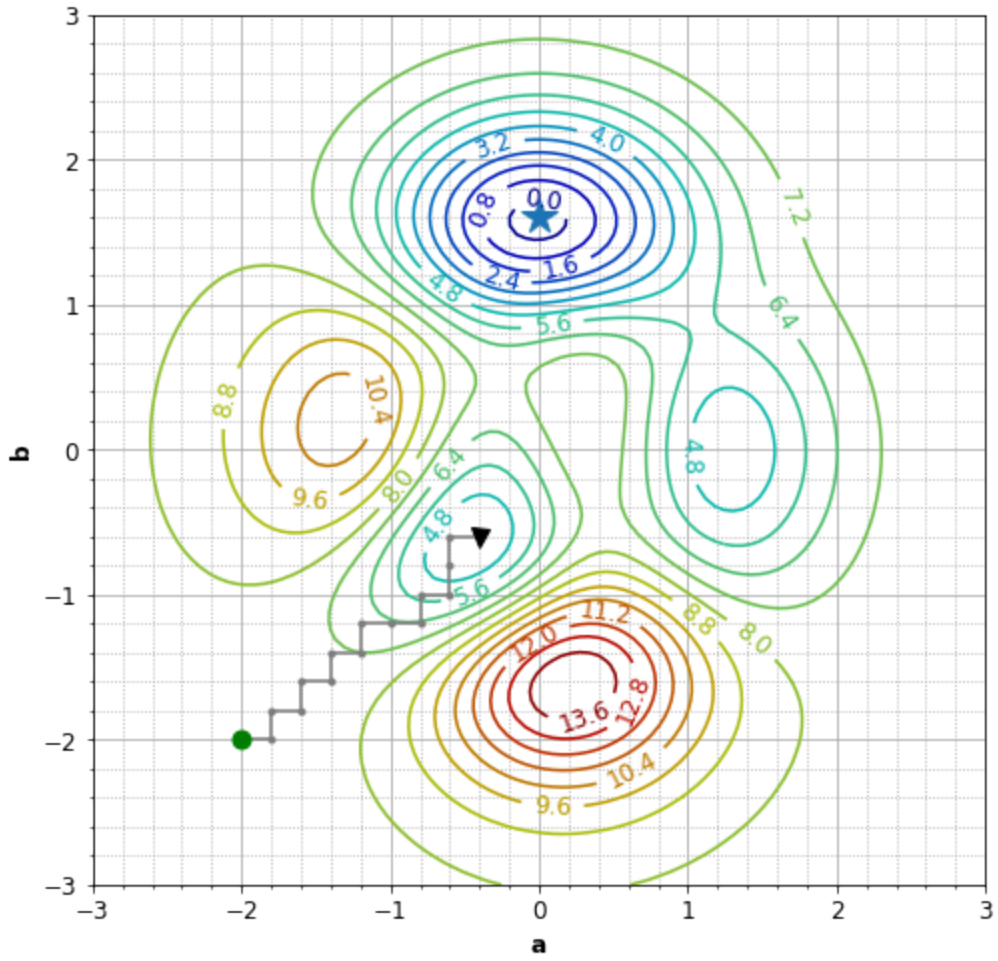
Code Cell 2 of 8

In [4]

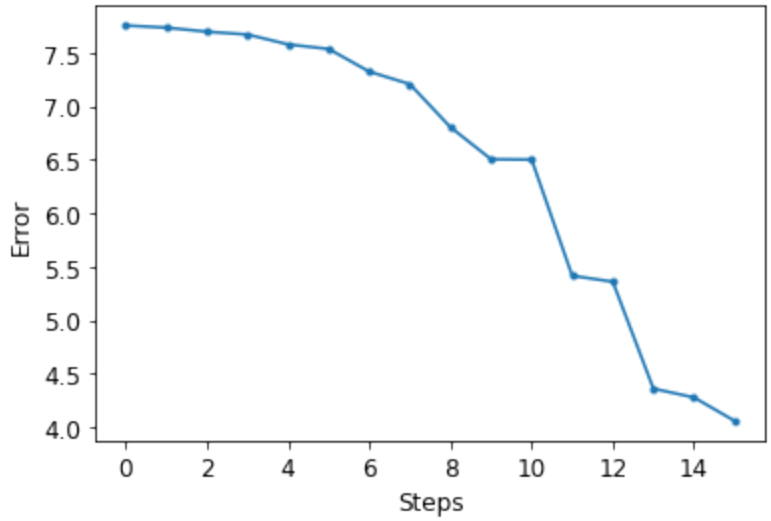
```
# call the optimization functions and visualize the process
# you can play around with different starting points and step sizes!
START_A = -2
START_B = -2
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)

# uncomment the next line if you want to print the resulting lists for a, b, and Error
# optimize(step, START_A, START_B, stepsize)
```

Run Code

Out [4]




```
Final location (a,b) = (-0.4, -0.6)
Final value of Error(a,b) = 4.06
```



Question 1 of 15

Focusing mostly on the `optimize` and `step` functions and the displayed output, define the optimization problem that this algorithm is trying to solve: What is the objective function? What

are the decision variables? Are there any constraints? Provide justification for your identifications (<150 words) [#optimization].

Normal

The objective function is trying to find the minimum value of the error function, error(a,b), by adjusting the values of two decision variables, 'a' and 'b.' The error function calculates the value of a mathematical expression. The objective value is the numerical value the algorithm is trying to minimize that represents how good or bad the solution is, so the returned value when calling error(a,b).
The decision variables 'a' and 'b' represent the coordinates on the error surface. The algorithm iteratively adjusts these variables in the 'step' function to see the error surface and move towards lower error values to find the optimal solution. The search space is constrained (direct traditional; constraint) by the limits set for the values of 'a' and 'b' in get_xyz, plot2d (e.g. line 59), and plot3d (a_limits and b_limits set to (-3, 3)). In line 147, max_steps=150 is a constraint on the optimization process itself, rather than directly on the decision variables, so it is an indirect (operational) constraint.

Question 2 of 15

Again, focusing on the `optimize` and `step` functions, as well as the visual output, describe how the algorithm works holistically. You can ignore all of the other functions, which are mostly just for plotting the visualizations, included here only for reference. Identify the inputs, outputs, key steps (e.g., how decision variables are being changed), and the termination condition. Explain at least one advantage and one limitation of this algorithm, being sure to justify whether or not this algorithm would be guaranteed to lead to the global optimum (<200 words) [#algorithms].

Normal

The **inputs** are the initial guesses for 'a' and 'b' (lines 149-151) which determine how big of steps to take (stepsize), and set a limit on how many steps to try (max_steps). The **outputs** are the final 'a' and 'b' values and the minimum error.
In the **optimize** function, the algorithm starts by creating lists to keep track of the 'a,' 'b,' and error values (lines 113-115). Then, it goes into a loop, repeatedly applying the **step** function to change 'a' and 'b' according to the chosen step size (lines 113-164). It compares the error at the current position to the errors at potential next positions (forward, backward, up, down) and moves to the position where the error is minimized. This loop keeps going until it hits the maximum number of allowed steps or if there's no more room for improvement. The

algorithm stops (termination) if it reaches the maximum number of steps or if the step function does not lead to a change in 'a' or 'b' (indicating a local minimum).
This algorithm is fast in simple situations where the problem is not too complicated. It can quickly find a good solution if the landscape of the problem is smooth and straightforward which is a **strength**. However, it has the risk of getting stuck at a local minimum. The algorithm's success depends on the initial starting point and the chosen step size which could lead to suboptimal solutions, especially in scenarios with multiple lows.

Question 3 of 15

Identify how this algorithm can be modified to improve its ability to reach the global optimum. There are many ways this algorithm can be improved, but for the purpose of this task, just pick ONE specific alteration, retaining the overall hill-climbing structure of the code (major changes can be explored below). You can suggest modifications to the `optimize` and/or `step` functions. This need not be anything completely novel or sophisticated. Start with a simple and straightforward modification that you understand and can explain well.

Here, write a brief summary to explain your proposed modification. Describe how you would go about implementing this modification in the code. Justify why this would be an improvement. With this modification, revisit the advantages and limitations, including the algorithm's ability to reach the global optimum (<200 words) [#algorithms].

Normal

The original method moves the algorithm in four directions: up, down, left, and right, choosing the direction that improves the situation the most. I suggest we also allow it to move diagonally, adding four more directions. This way, the algorithm can explore better and potentially find the best solution faster.

To implement, I need to consider the four diagonal directions (upper right, upper left, lower right, lower left) in addition to the original four. Then, calculate the error for all eight possible directions plus the current position to determine which move would result in the lowest error. Out of all evaluated moves, we select the one with the lowest error. If the current position has the lowest error, do not move.

By considering diagonal moves, the algorithm can navigate more freely, taking shortcuts that may lead to quicker and more efficient solutions, especially when the best path is not straight. This could mean reaching a local or global optimum in fewer steps, saving both time and resources.

However, checking eight directions at each step means more calculations, which might slow down the process. Also, even with this added flexibility, finding the global optimum in complex scenarios isn't guaranteed, but if the algorithm found it, it would be faster.

Question 4 of 15

Optional challenge: Implement your modification directly in the code cells below and add in-line comments to explain how your added code works. Use the following code cells as needed to test out your implementation and demonstrate sufficient evidence of your testing. Here, describe how you tested your algorithm and explain why this approach was effective. Also, be sure to cite any external materials used to develop the idea for the implementation (<200 words) [#algorithms].

Normal

I started by running the algorithm with the original four directions to see how many steps it took and what the final error value was. Then, I implemented the algorithm to add diagonal directions and tested it under the same conditions to compare its performance. I tested the algorithm from different starting points to see how well it worked in different situations. This helped us understand how the extra directions affected the algorithm's speed and error reduction (for example, in the first test, the initial algorithm found the local minimum in 12 steps, whereas my modified one found it in 7).
The modified algorithm might find solutions faster by exploring more effectively since diagonal moves could offer a shorter path to the minimum in some cases (advantage). However, it takes more time and computational cost to calculate errors for eight directions at each step. The algorithm still doesn't guarantee finding the global minimum, risking getting stuck at a local minimum. The algorithm only explores its nearby surroundings, so it might perceive the local minimum as the best solution and stop searching further since it has nowhere lower to go. This can happen when the function has multiple valleys and peaks.

Code Cell 3 of 8

In [5]:
```
#original code!

import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from matplotlib.ticker import MultipleLocator, AutoMinorLocator
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Ready |

```
plt.rcParams.update({'font.size': 12})

def error(a, b):
   # defines the function from here https://www.youtube.com/watch?v=1i8muvzZkPw
    func = 3*(1 - a)**2*np.exp(-a**2-(b+1)**2) - 10*(a/5-a**3-b**5)*np.exp(-a**2-b**2) - (1/3)*np.exp(-b**2-(a+1)**2)
    return -func + 7.8

def get_xyz(f, a_limits=(-3, 3), b_limits=(-3, 3)):
    # sets up an xy grid with associated z values for 3d plotting
    a = np.linspace(a_limits[0], a_limits[1], num=100)
    b = np.linspace(b_limits[0], b_limits[1], num=100)

    mesh = np.meshgrid(a, b)
    z = np.vectorize(f)(mesh[0], mesh[1])

    return mesh[0], mesh[1], z

def fmt(x):
    return '{0:.{1}f}'.format(x, 1)

def cmap_map(function, cmap):
    """ Applies function (which should operate on vectors of shape 3: [r, g, b]), on colormap cmap.
    This routine will break any discontinuous points in a colormap.
    """
    cdict = cmap._segmentdata
    step_dict = {}
    # Firt get the list of points where the segments start or end
    for key in ('red', 'green', 'blue'):
        step_dict[key] = list(map(lambda x: x[0], cdict[key]))
    step_list = sum(step_dict.values(), [])
    step_list = np.array(list(set(step_list)))
    # Then compute the LUT, and apply the function to the LUT
    reduced_cmap = lambda step : np.array(cmap(step)[0:3])
    old_LUT = np.array(list(map(reduced_cmap, step_list)))
    new_LUT = np.array(list(map(function, old_LUT)))
    # Now try to make a minimal segment definition of the new LUT
    cdict = {}
    for i, key in enumerate(['red','green','blue']):
        this_cdict = {}
```

```
        for j, step in enumerate(step_list):
            if step in step_dict[key]:
                this_cdict[step] = new_LUT[j, i]
            elif new_LUT[j,i] != old_LUT[j, i]:
                this_cdict[step] = new_LUT[j, i]
        colorvector = list(map(lambda x: x + (x[1], ), this_cdict.items()))
        colorvector.sort()
        cdict[key] = colorvector

    return matplotlib.colors.LinearSegmentedColormap('colormap',cdict,1024)

dark_jet = cmap_map(lambda x: x*0.75, matplotlib.cm.jet)

def _plot2d(f, a_limits=(-3, 3), b_limits=(-3, 3), max=(0, 1.6)):
    # creates a 2d contour plot
    a, b, z = get_xyz(f, a_limits, b_limits)

    fig = plt.figure(figsize=(8,8))
    ax = plt.axes()

    z = ax.contour(a, b, z, levels=22, cmap=dark_jet)
    ax.clabel(z, inline=True, fontsize=12,  fmt=fmt)

    ax.grid(True)
    ax.grid(True, which="minor", linestyle="dotted")

    ax.xaxis.set_major_locator(MultipleLocator(1))
    ax.xaxis.set_minor_locator(AutoMinorLocator(5))

    ax.yaxis.set_major_locator(MultipleLocator(1))
    ax.yaxis.set_minor_locator(AutoMinorLocator(5))

    ax.plot(*max, marker="*", markersize=18)

    return fig, ax

def plot2d(f, a_limits=(-3, 3), b_limits=(-3, 3), max=(0, 1.6)):
    # plots the optimization landscape
    _, ax = _plot2d(f, a_limits, b_limits, max)
    ax.set_xlabel('a',fontweight='bold')
```

```
    plt.show()

def plot3d(f, a_limits=(-3, 3), b_limits=(-3, 3)):
   # same, but plots in 3d with the same colorscheme
    a, b, z = get_xyz(f, a_limits, b_limits)

    fig = plt.figure(figsize=(14, 7))
    ax = fig.add_subplot(1, 2, 1, projection='3d')
    #ax = plt.axes(projection='3d')
    ax.view_init(12,-135)
    ax.plot_surface(a, b, z, cmap=dark_jet, edgecolor='none')
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    ax.set_zlabel('Error',fontweight='bold')

    ax = fig.add_subplot(1, 2, 2, projection='3d')
    ax.view_init(-5,40)
    ax.plot_surface(a, b, z, cmap=dark_jet, edgecolor='none')
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    ax.set_zlabel('Error',fontweight='bold')

    plt.show()

def plot_optimization(move, start_a, start_b, stepsize, max_steps=1500):
    # given the starting points, creates a 2d contour plot, and error plot, and prints the end results
    a, b, errors = optimize(move, start_a, start_b, stepsize=stepsize, max_steps=1500)

    fig, ax = _plot2d(error)
    ax.plot(a, b, marker='o',  markersize=3, color='gray')
    ax.plot(start_a, start_b, marker='o',  markersize=9, color='green')
    ax.plot(a[-1], b[-1], marker='v',  markersize=9, color='black')
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    plt.show()
    print("Final location (a,b) =", (round(a[-1],2),round(b[-1],2)))
    print("Final value of Error(a,b) =", round(errors[-1],3))

    plt.xlabel('Steps')
```

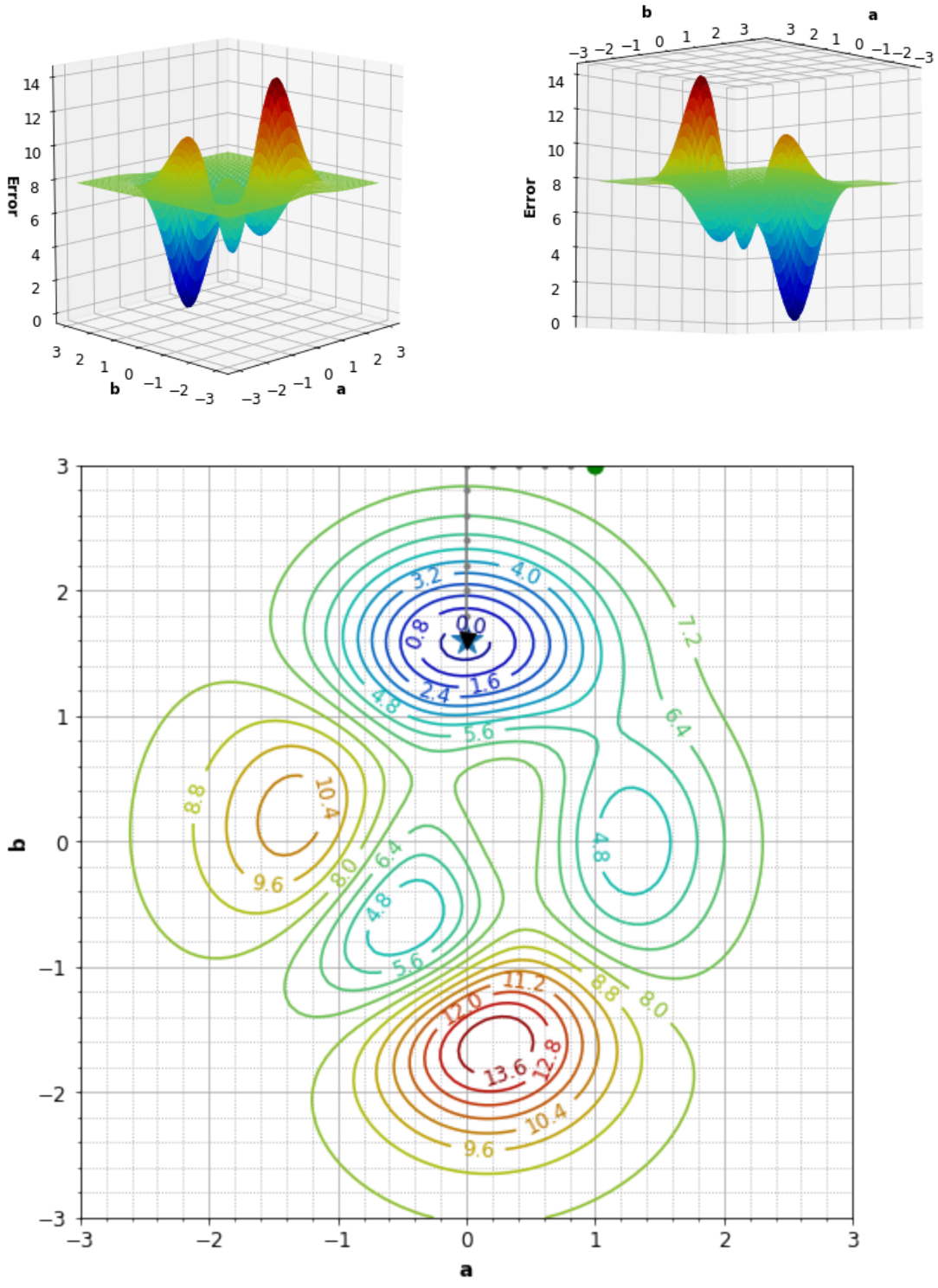
```
127     plt.ylabel('Error')
128     plt.plot(errors, marker='.')
129     plt.show()
130 
131 ### THE FUNCTIONS BELOW SHOULD BE THE FOCAL POINT OF YOUR ANALYSIS
132 
133 def step(a, b, stepsize):
134     # how to take a step
135     current_error = error(a, b)
136 
137     if error(a + stepsize, b) < current_error:
138         a = a + stepsize
139     elif error(a, b + stepsize) < current_error:
140         b = b + stepsize
141     elif error(a - stepsize, b) < current_error:
142         a = a - stepsize
143     elif error(a, b - stepsize) < current_error:
144         b = b - stepsize
145 
146     return a, b
147 
148 def optimize(move, start_a, start_b, stepsize, max_steps=150):
149     # takes defined "move" function, and implements an optimization run
150     all_a = [start_a]
151     all_b = [start_b]
152     all_error = [error(start_a,start_b)]
153 
154     for i in range(max_steps):
155         a, b = move(start_a, start_b, stepsize)
156 
157         if a == start_a and b == start_b:
158             break
159 
160         start_a, start_b = a, b
161 
162         all_a.append(start_a)
163         all_b.append(start_b)
164         all_error.append(error(start_a,start_b))
165 
166     return all_a, all_b, all_error
```

```
plot3d(error)
# call the optimization functions and visualize the process
# you can play around with different starting points and step sizes!
START_A = 1
START_B = 3
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)

# optional implementation and testing of hill-climbing modification here

# TESTING
START_A = 0
START_B = 0.5
stepsize = 0.5
plot_optimization(step, START_A, START_B, stepsize)
START_A = -2
START_B = -3
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)
START_A = -0.1
START_B = -1.2
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)
START_A = -0.9
START_B = -1.4
stepsize = 0.3
plot_optimization(step, START_A, START_B, stepsize)
START_A = 1
START_B = 2
stepsize = 0.5
plot_optimization(step, START_A, START_B, stepsize)
START_A = 1
START_B = 2
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)
START_A = 2
START_B = 2
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)

```

```
208 # uncomment the next line if you want to print the resulting lists for a, b,
    and Error
209 # optimize(step, START_A, START_B, stepsize)
```

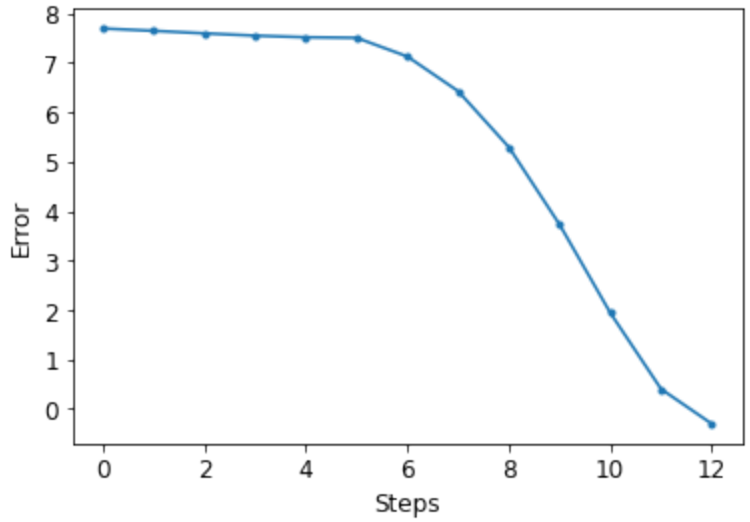
Run Code

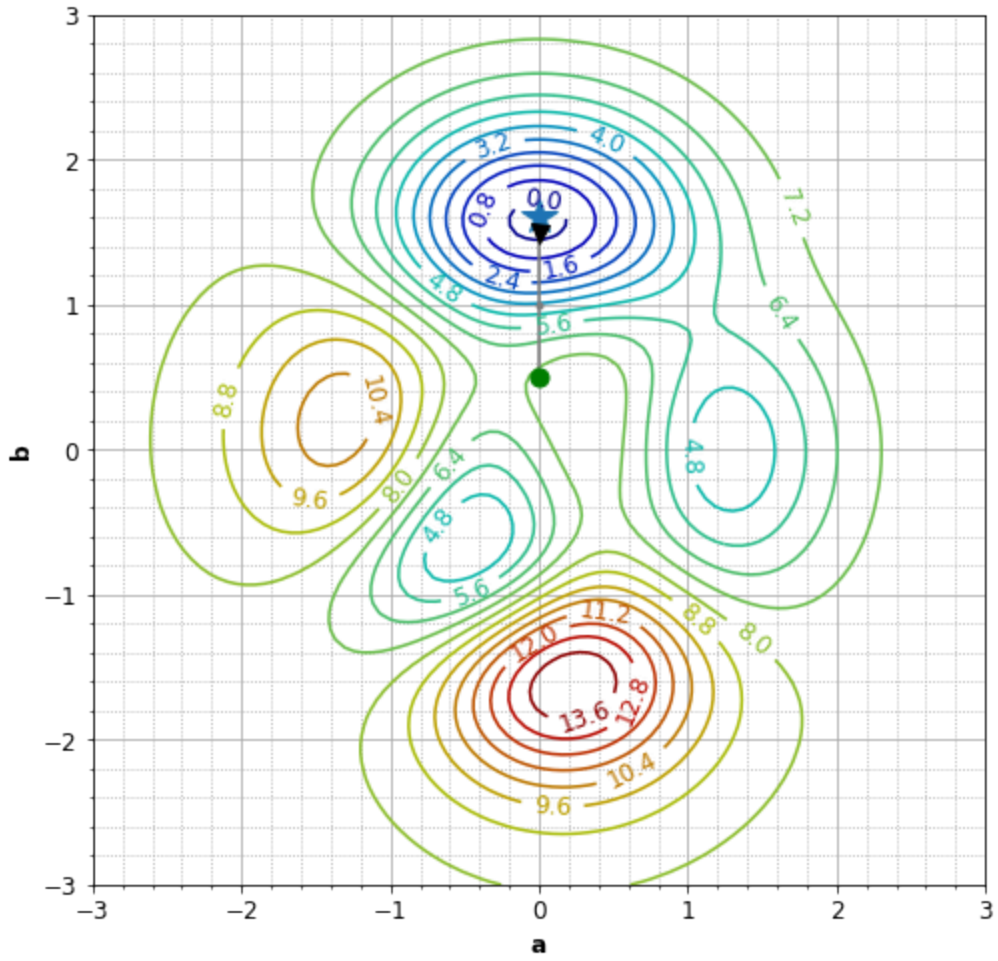
Out [5]



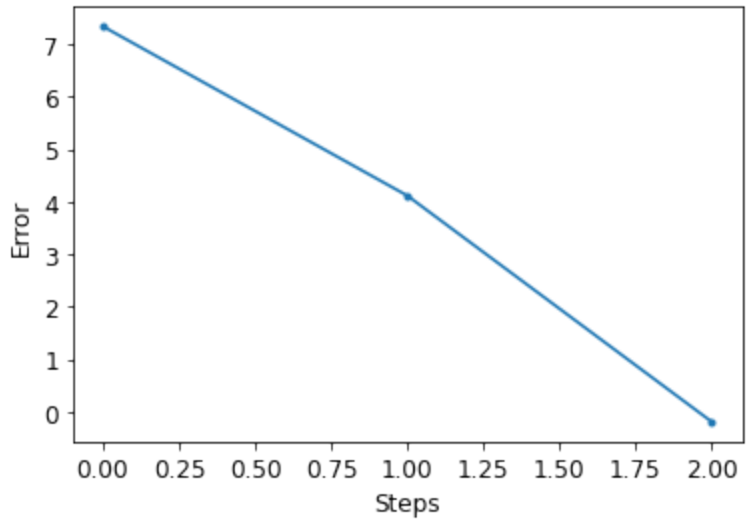
Python 3 (384MB RAM) | Edit Run All Cells Kernel Ready | |

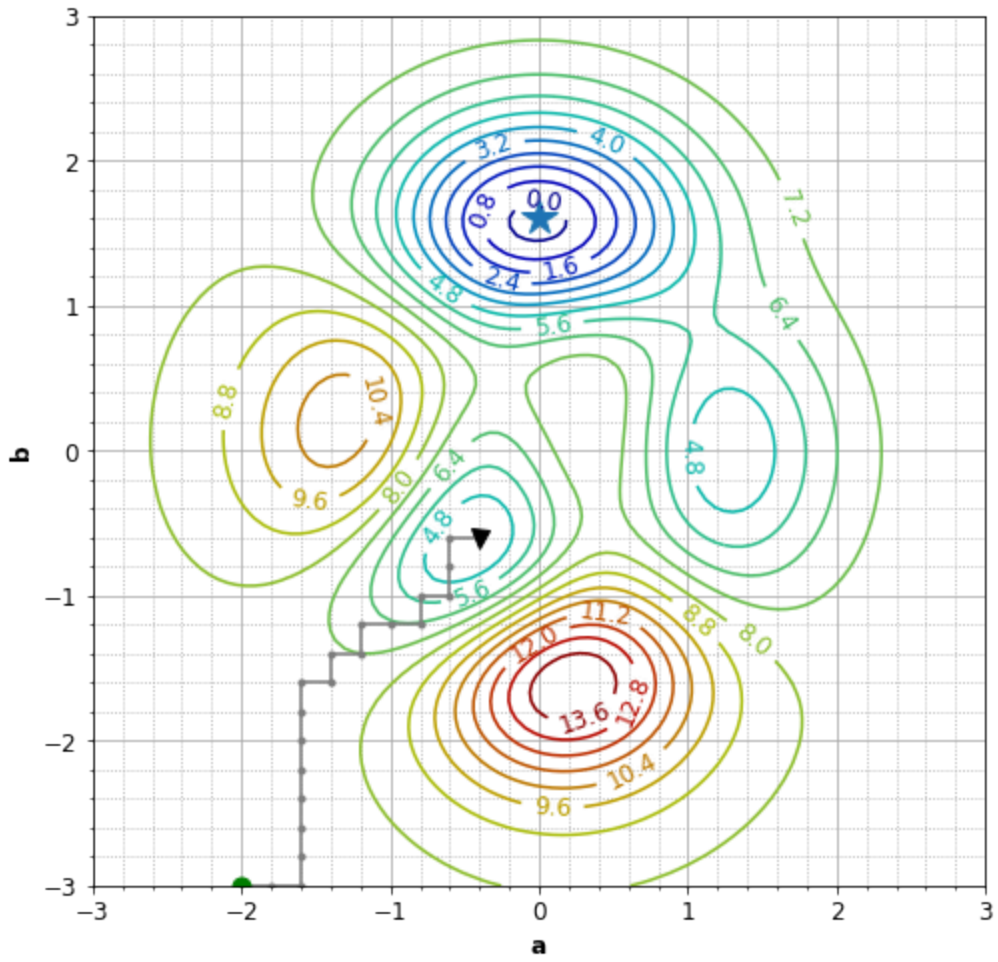
```
Final location (a,b) = (0.0, 1.6)
Final value of Error(a,b) = -0.3
```



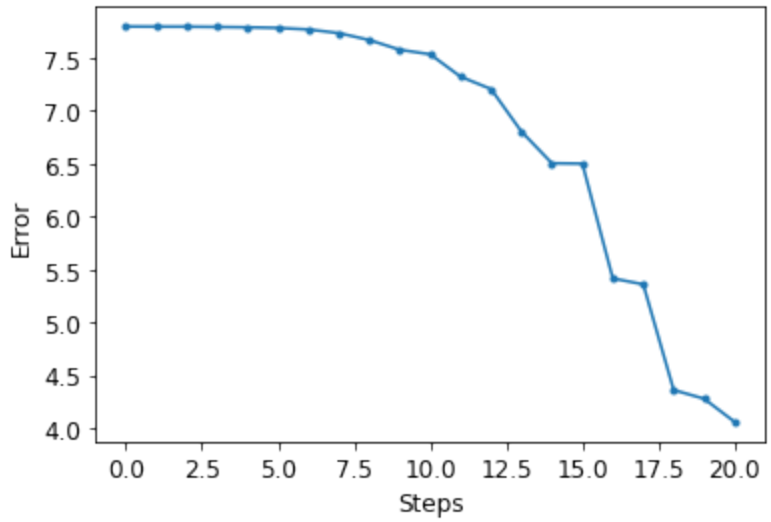


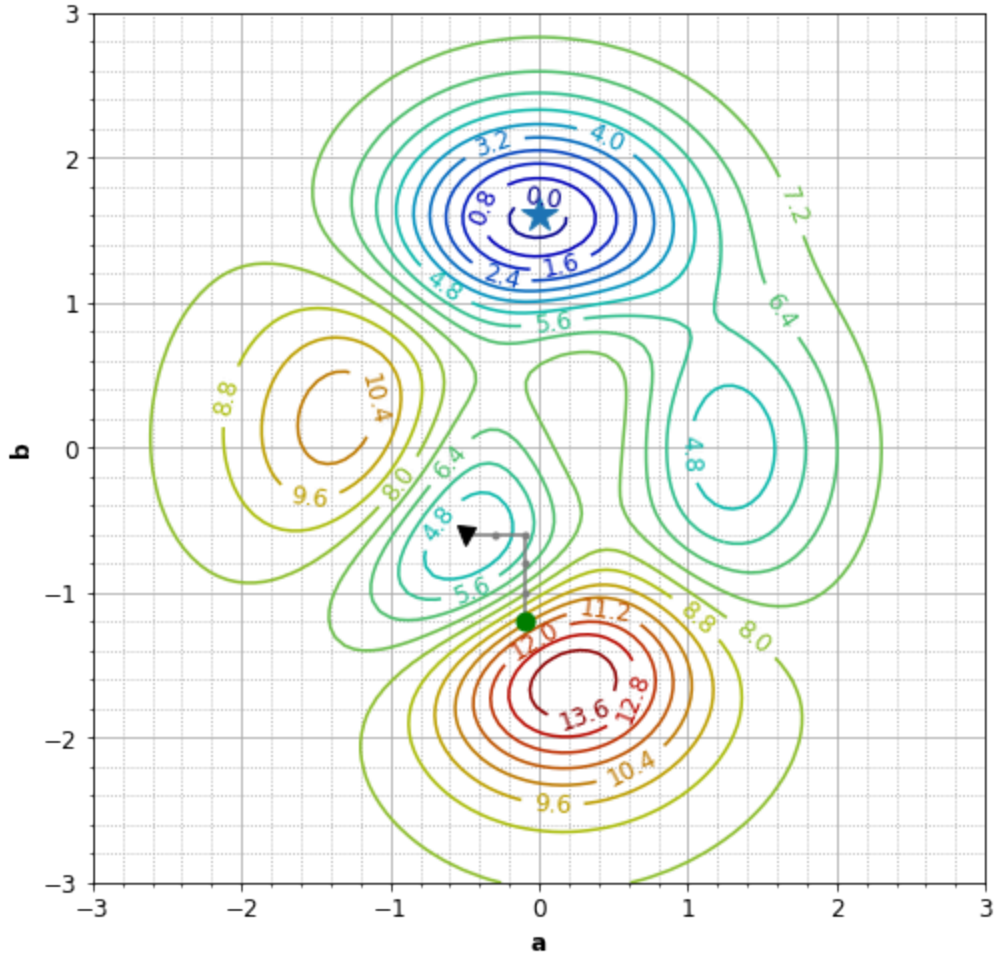

```
Final location (a,b) = (0, 1.5)
Final value of Error(a,b) = -0.197
```



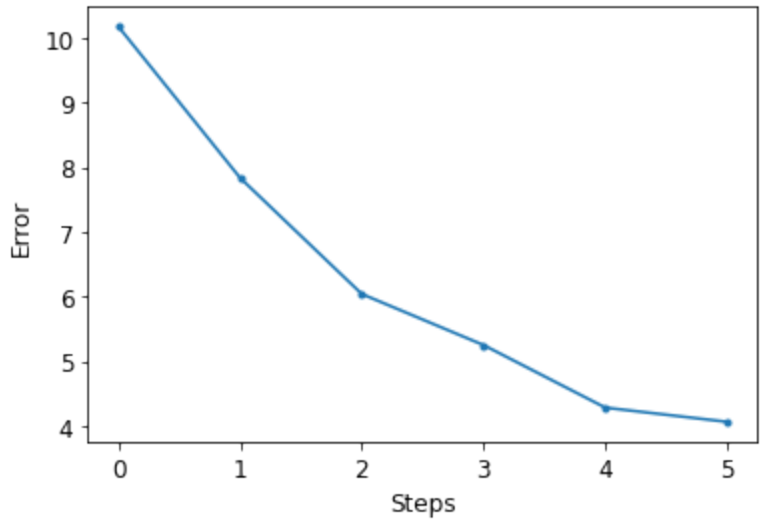


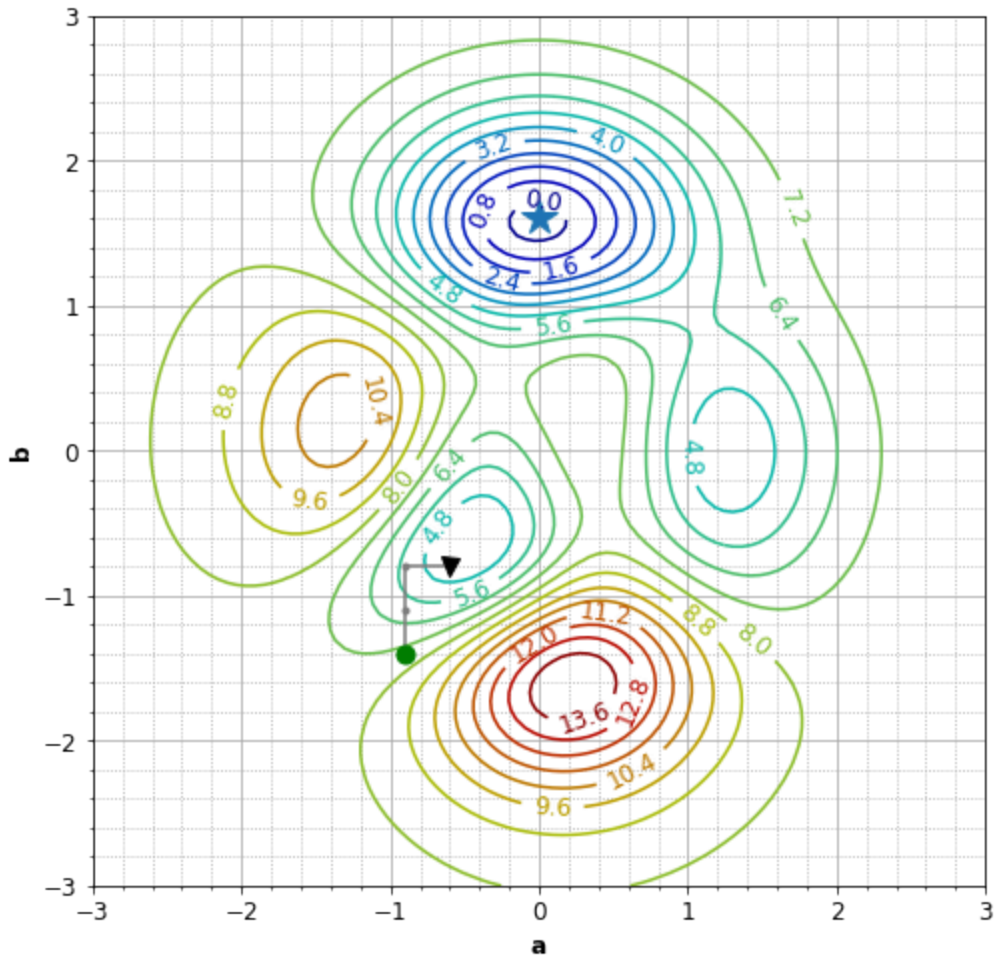

```
Final location (a,b) = (-0.4, -0.6)
Final value of Error(a,b) = 4.06
```



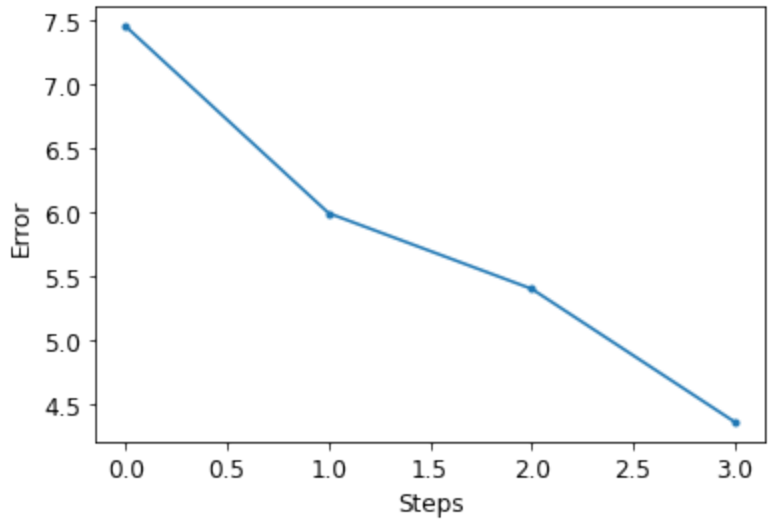


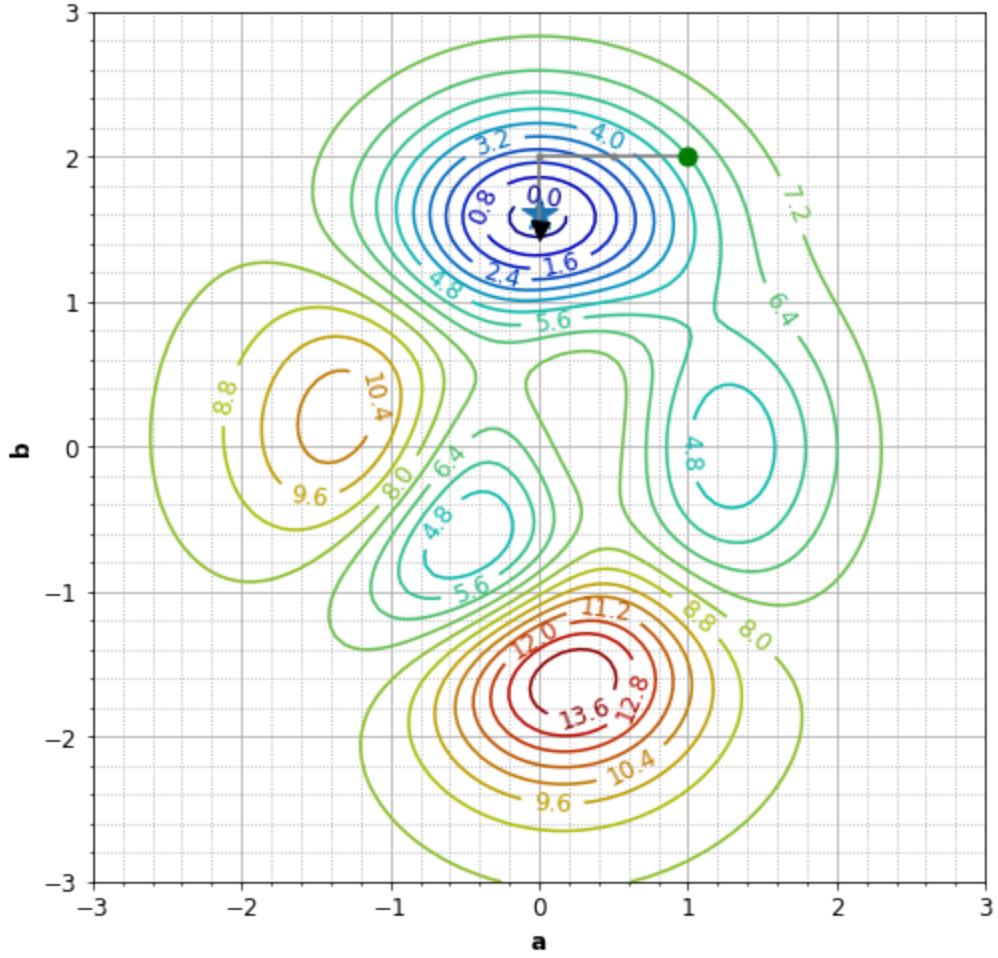

```
Final location (a,b) = (-0.5, -0.6)
Final value of Error(a,b) = 4.06
```



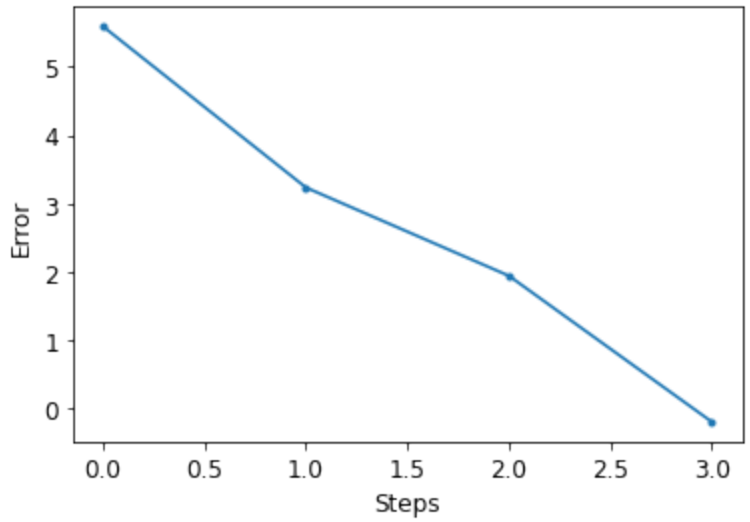


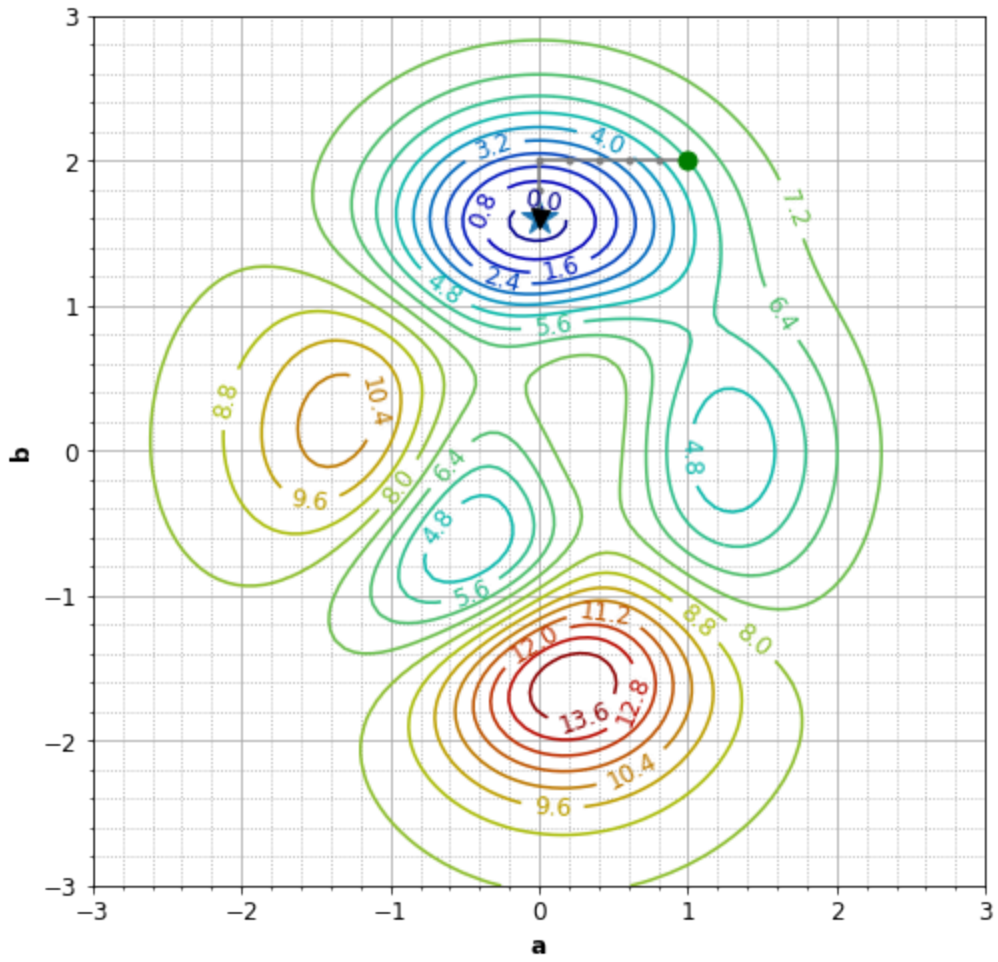

```
Final location (a,b) = (-0.6, -0.8)
Final value of Error(a,b) = 4.36
```



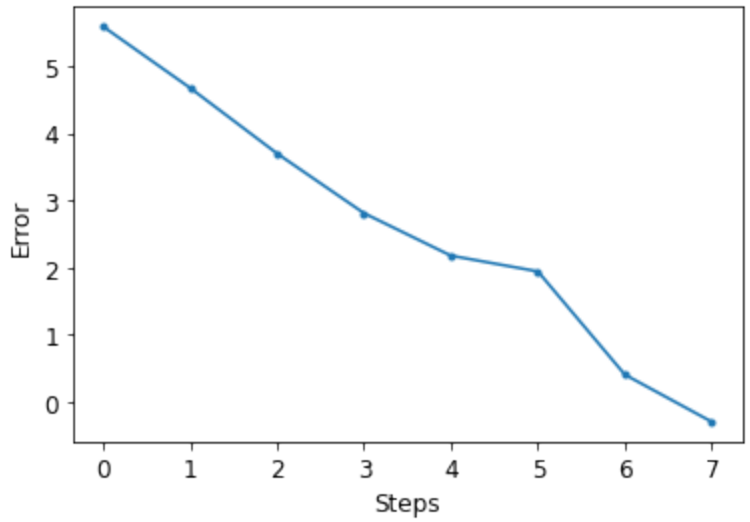


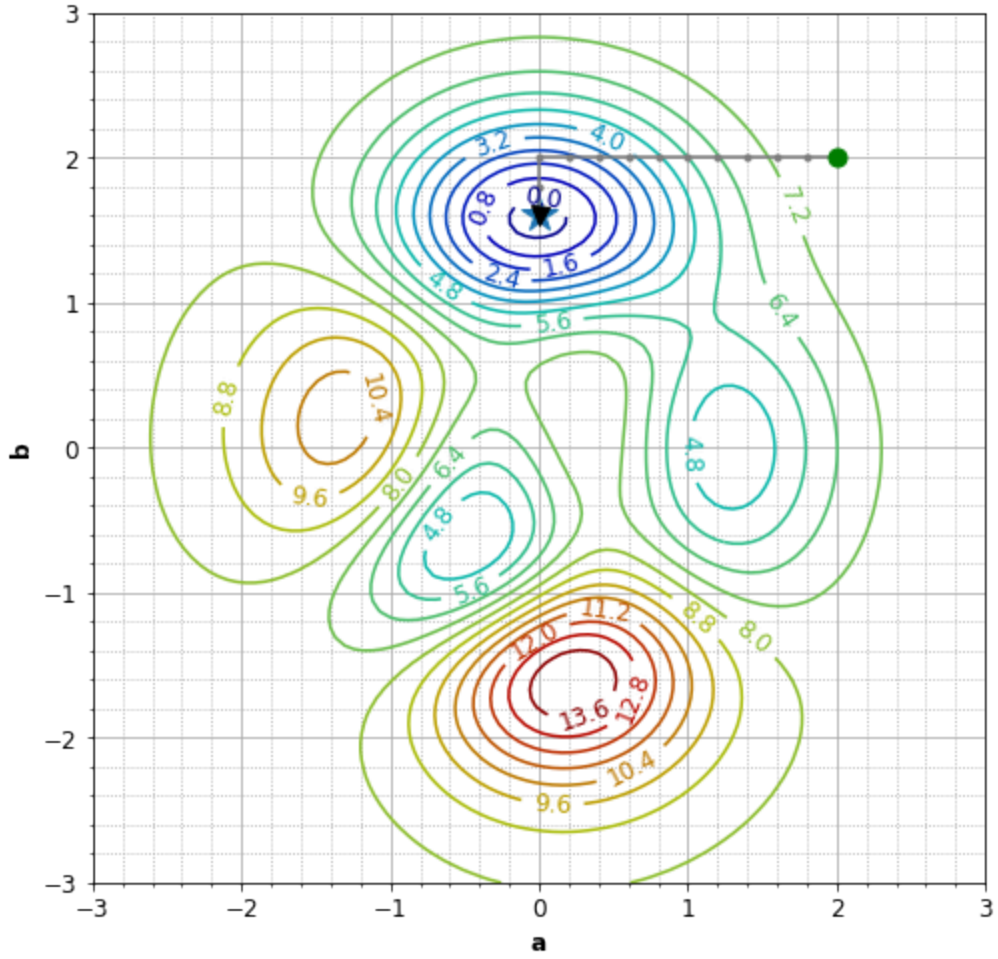

```
Final location (a,b) = (0.0, 1.5)
Final value of Error(a,b) = -0.197
```



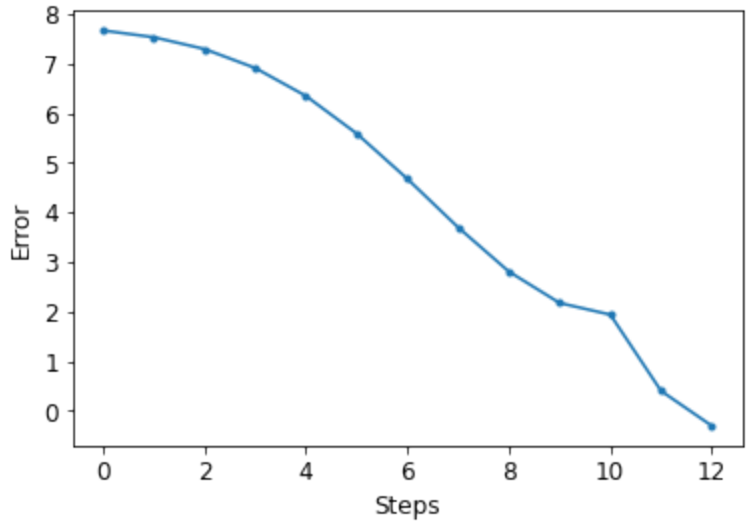



```
Final location (a,b) = (0.0, 1.6)
Final value of Error(a,b) = -0.3
```






```
Final location (a,b) = (0.0, 1.6)
Final value of Error(a,b) = -0.3
```



Code Cell 4 of 8

In [6]

```
# modified code with diagonals
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
from matplotlib.ticker import MultipleLocator, AutoMinorLocator
plt.rcParams.update({'font.size': 12})

def error(a, b):
   # defines the function from here https://www.youtube.com/watch?v=1i8muvzZkPw
    func = 3*(1 - a)**2*np.exp(-a**2-(b+1)**2) - 10*(a/5-a**3-b**5)*np.exp(-a**2-b**2) - (1/3)*np.exp(-b**2-(a+1)**2)
    return -func + 7.8

def get_xyz(f, a_limits=(-3, 3), b_limits=(-3, 3)):
    # sets up an xy grid with associated z values for 3d plotting
    a = np.linspace(a_limits[0], a_limits[1], num=100)
    b = np.linspace(b_limits[0], b_limits[1], num=100)

    mesh = np.meshgrid(a, b)
    z = np.vectorize(f)(mesh[0], mesh[1])

    return mesh[0], mesh[1], z

```

```
    return '{0:.{1}f}'.format(x, 1)

def cmap_map(function, cmap):
    """ Applies function (which should operate on vectors of shape 3: [r, g, b]), on colormap cmap.
    This routine will break any discontinuous points in a colormap.
    """
    cdict = cmap._segmentdata
    step_dict = {}
    # Firt get the list of points where the segments start or end
    for key in ('red', 'green', 'blue'):
        step_dict[key] = list(map(lambda x: x[0], cdict[key]))
    step_list = sum(step_dict.values(), [])
    step_list = np.array(list(set(step_list)))
    # Then compute the LUT, and apply the function to the LUT
    reduced_cmap = lambda step : np.array(cmap(step)[0:3])
    old_LUT = np.array(list(map(reduced_cmap, step_list)))
    new_LUT = np.array(list(map(function, old_LUT)))
    # Now try to make a minimal segment definition of the new LUT
    cdict = {}
    for i, key in enumerate(['red','green','blue']):
        this_cdict = {}
        for j, step in enumerate(step_list):
            if step in step_dict[key]:
                this_cdict[step] = new_LUT[j, i]
            elif new_LUT[j,i] != old_LUT[j, i]:
                this_cdict[step] = new_LUT[j, i]
        colorvector = list(map(lambda x: x + (x[1], ), this_cdict.items()))
        colorvector.sort()
        cdict[key] = colorvector

    return matplotlib.colors.LinearSegmentedColormap('colormap',cdict,1024)

dark_jet = cmap_map(lambda x: x*0.75, matplotlib.cm.jet)

def _plot2d(f, a_limits=(-3, 3), b_limits=(-3, 3), max=(0, 1.6)):
    # creates a 2d contour plot
    a, b, z = get_xyz(f, a_limits, b_limits)

    fig = plt.figure(figsize=(8,8))
```

```
65
66      z = ax.contour(a, b, z, levels=22, cmap=dark_jet)
67      ax.clabel(z, inline=True, fontsize=12,  fmt=fmt)
68
69      ax.grid(True)
70      ax.grid(True, which="minor", linestyle="dotted")
71
72      ax.xaxis.set_major_locator(MultipleLocator(1))
73      ax.xaxis.set_minor_locator(AutoMinorLocator(5))
74
75      ax.yaxis.set_major_locator(MultipleLocator(1))
76      ax.yaxis.set_minor_locator(AutoMinorLocator(5))
77
78      ax.plot(*max, marker="*", markersize=18)
79
80      return fig, ax
81
82  def plot2d(f, a_limits=(-3, 3), b_limits=(-3, 3), max=(0, 1.6)):
83      # plots the optimization landscape
84      _, ax = _plot2d(f, a_limits, b_limits, max)
85      ax.set_xlabel('a',fontweight='bold')
86      ax.set_ylabel('b',fontweight='bold')
87      plt.show()
88
89  def plot3d(f, a_limits=(-3, 3), b_limits=(-3, 3)):
90     # same, but plots in 3d with the same colorscheme
91      a, b, z = get_xyz(f, a_limits, b_limits)
92
93      fig = plt.figure(figsize=(14, 7))
94      ax = fig.add_subplot(1, 2, 1, projection='3d')
95      #ax = plt.axes(projection='3d')
96      ax.view_init(12,-135)
97      ax.plot_surface(a, b, z, cmap=dark_jet, edgecolor='none')
98      ax.set_xlabel('a',fontweight='bold')
99      ax.set_ylabel('b',fontweight='bold')
100     ax.set_zlabel('Error',fontweight='bold')
101
102     ax = fig.add_subplot(1, 2, 2, projection='3d')
103     ax.view_init(-5,40)
104     ax.plot_surface(a, b, z, cmap=dark_jet, edgecolor='none')
```

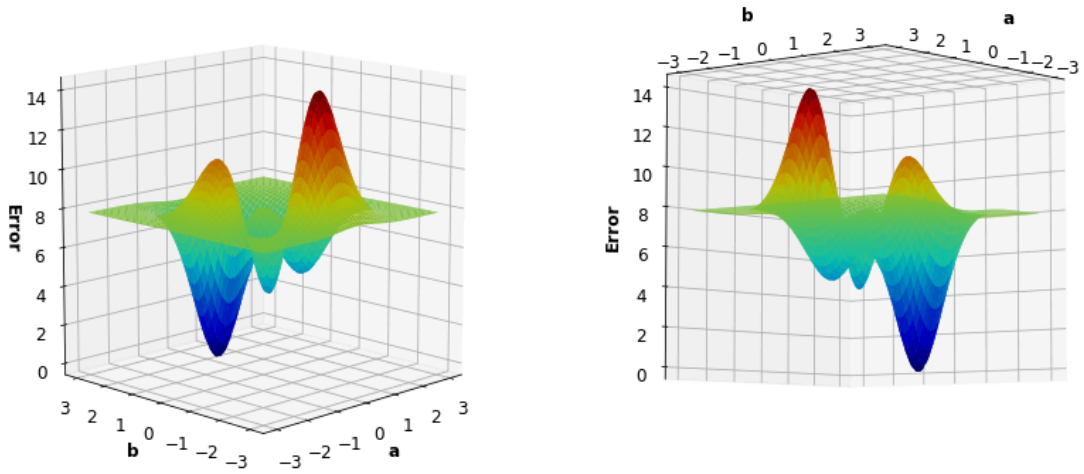
```
    ax.set_ylabel('b',fontweight='bold')
    ax.set_zlabel('Error',fontweight='bold')

    plt.show()

def plot_optimization(move, start_a, start_b, stepsize, max_steps=1500):
    # given the starting points, creates a 2d contour plot, and error plot, and prints the end results
    a, b, errors = optimize(move, start_a, start_b, stepsize=stepsize, max_steps=1500)

    fig, ax = _plot2d(error)
    ax.plot(a, b, marker='o',  markersize=3, color='gray')
    ax.plot(start_a, start_b, marker='o',  markersize=9, color='green')
    ax.plot(a[-1], b[-1], marker='v',  markersize=9, color='black')
    ax.set_xlabel('a',fontweight='bold')
    ax.set_ylabel('b',fontweight='bold')
    plt.show()
    print("Final location (a,b) =", (round(a[-1],2),round(b[-1],2)))
    print("Final value of Error(a,b) =", round(errors[-1],3))

    plt.xlabel('Steps')
    plt.ylabel('Error')
    plt.plot(errors, marker='.')
    plt.show()

### THE FUNCTIONS BELOW SHOULD BE THE FOCAL POINT OF YOUR ANALYSIS

def step(a, b, stepsize):
    current_error = error(a, b)  # Calculate current error at starting position by calling the error function with the current coordinates (a, b)
    best_a, best_b = a, b  # Initialize the best move at the current position. These variables will store the coordinates of the best next step, starting with the assumption that staying in place is the best move.
    best_error = current_error  # Record the error of the current position as the best error. This variable tracks the lowest error found among all possible moves, starting with the current position's error.

    # Define all possible moves, including diagonal movements
    moves = [(stepsize, 0), (-stepsize, 0), (0, stepsize), (0, -stepsize),
             (stepsize, stepsize), (-stepsize, stepsize), (stepsize, -stepsize), (-stepsize, -stepsize)] #represent all possible moves from the current position. It includes moves in the four cardinal directions (up, down,
```

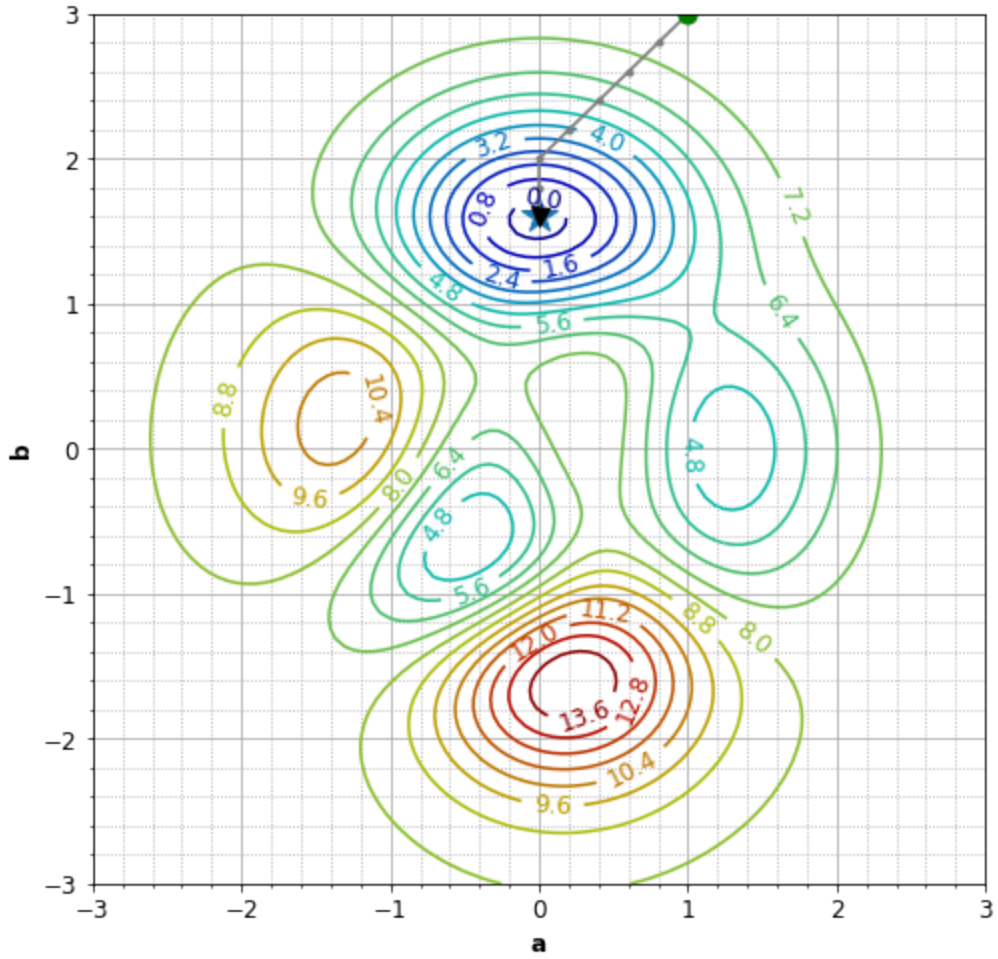
```
change in a (first element) and a change in b (second element) corresponding to a specific direction.

    for da, db in moves:  # Iterate through each possible move
        new_a, new_b = a + da, b + db  # Calculate new position after the move. Inside the loop, this line calculates the new position (new_a, new_b) that results from applying a move to the current position (a, b)
        new_error = error(new_a, new_b)  # Calculate error at the new position. Once the new position is calculated, this line computes the error at that new position using the error function.
        if new_error < best_error:  # If the new error is lower than the best error
            best_a, best_b = new_a, new_b  # Update the best move to this new position
            best_error = new_error  # Update the best error to this new error

    return best_a, best_b  # Return the coordinates of the best move


def optimize(move, start_a, start_b, stepsize, max_steps=150):
    # takes defined "move" function, and implements an optimization run
    all_a = [start_a]
    all_b = [start_b]
    all_error = [error(start_a,start_b)]

    for i in range(max_steps):
        a, b = move(start_a, start_b, stepsize)

        if a == start_a and b == start_b:
            break

        start_a, start_b = a, b

        all_a.append(start_a)
        all_b.append(start_b)
        all_error.append(error(start_a,start_b))

    return all_a, all_b, all_error

plot3d(error)

# call the optimization functions and visualize the process
# you can play around with different starting points and step sizes!
```

```
START_A = 1
START_B = 3
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)


# uncomment the next line if you want to print the resulting lists for a, b, and Error
# optimize(step, START_A, START_B, stepsize)
```

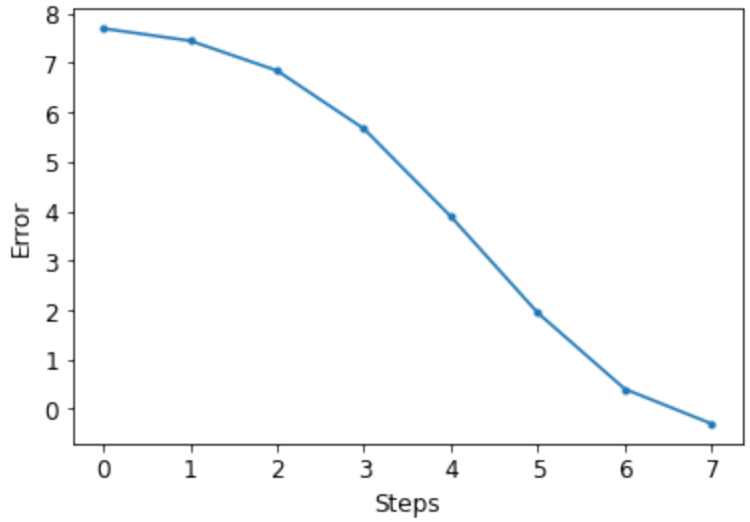
Run Code

Out [6]






```
Final location (a,b) = (0.0, 1.6)
Final value of Error(a,b) = -0.3
```



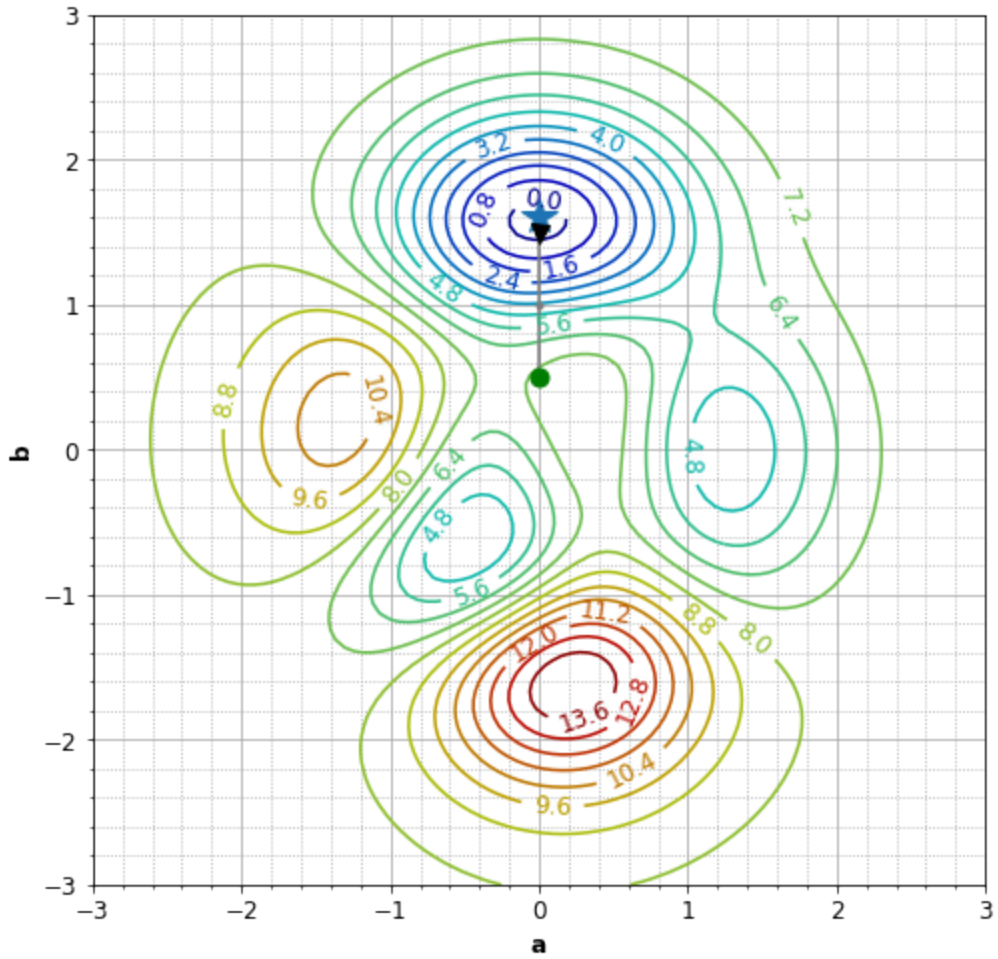
Code Cell 5 of 8

```
In [7]    1  # optional implementation and testing of hill-climbing modification here
          2
```

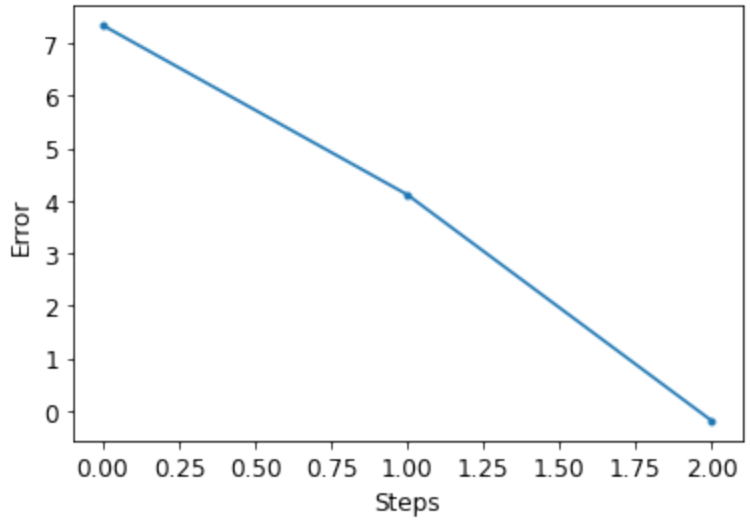
```
# TESTING
START_A = 0
START_B = 0.5
stepsize = 0.5
plot_optimization(step, START_A, START_B, stepsize)
START_A = -2
START_B = -3
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)
START_A = -0.1
START_B = -1.2
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)
START_A = -0.9
START_B = -1.4
stepsize = 0.3
plot_optimization(step, START_A, START_B, stepsize)
START_A = 1
START_B = 2
stepsize = 0.5
plot_optimization(step, START_A, START_B, stepsize)
START_A = 1
START_B = 2
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)
START_A = 2
START_B = 2
stepsize = 0.2
plot_optimization(step, START_A, START_B, stepsize)

```

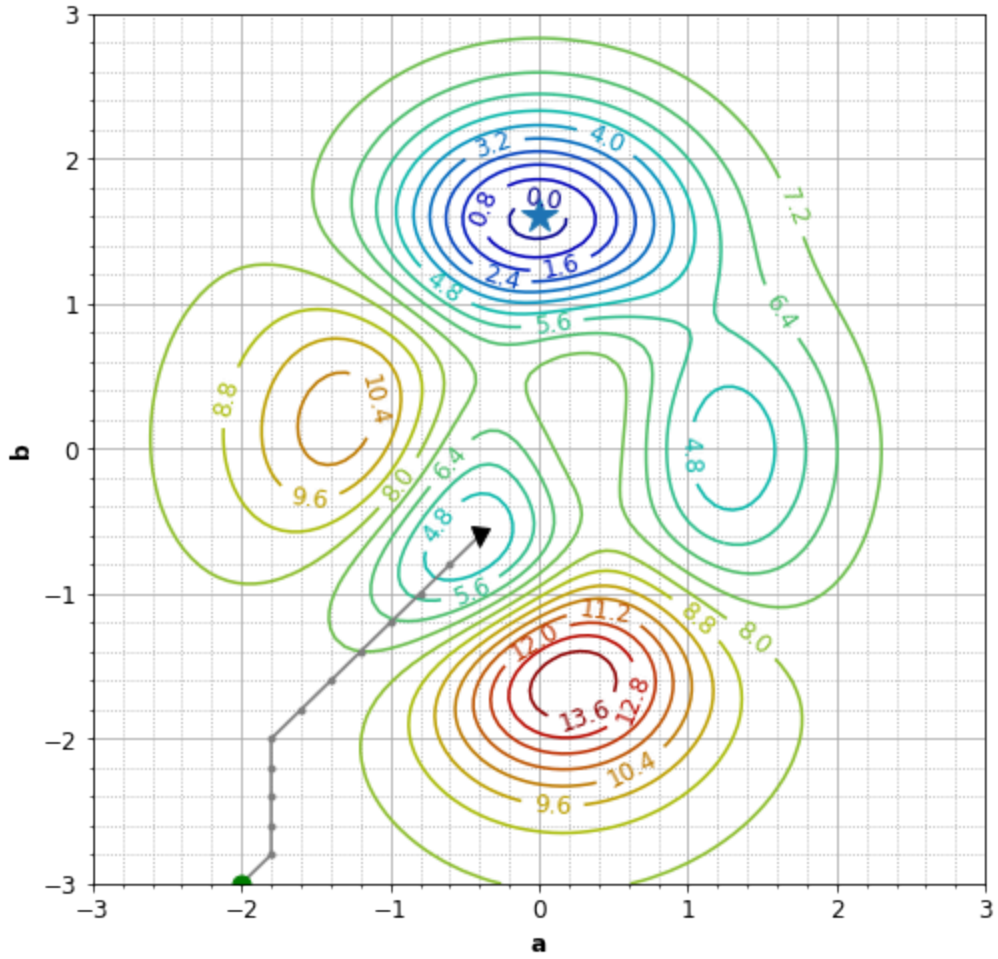
Run Code

Out [7]

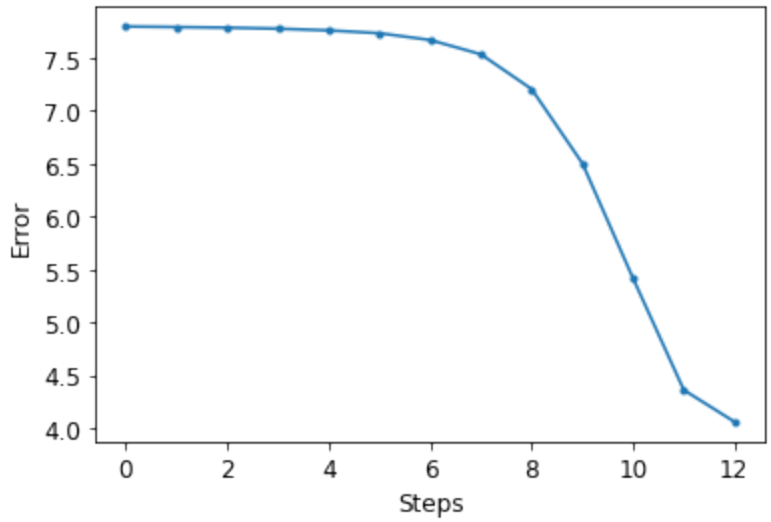


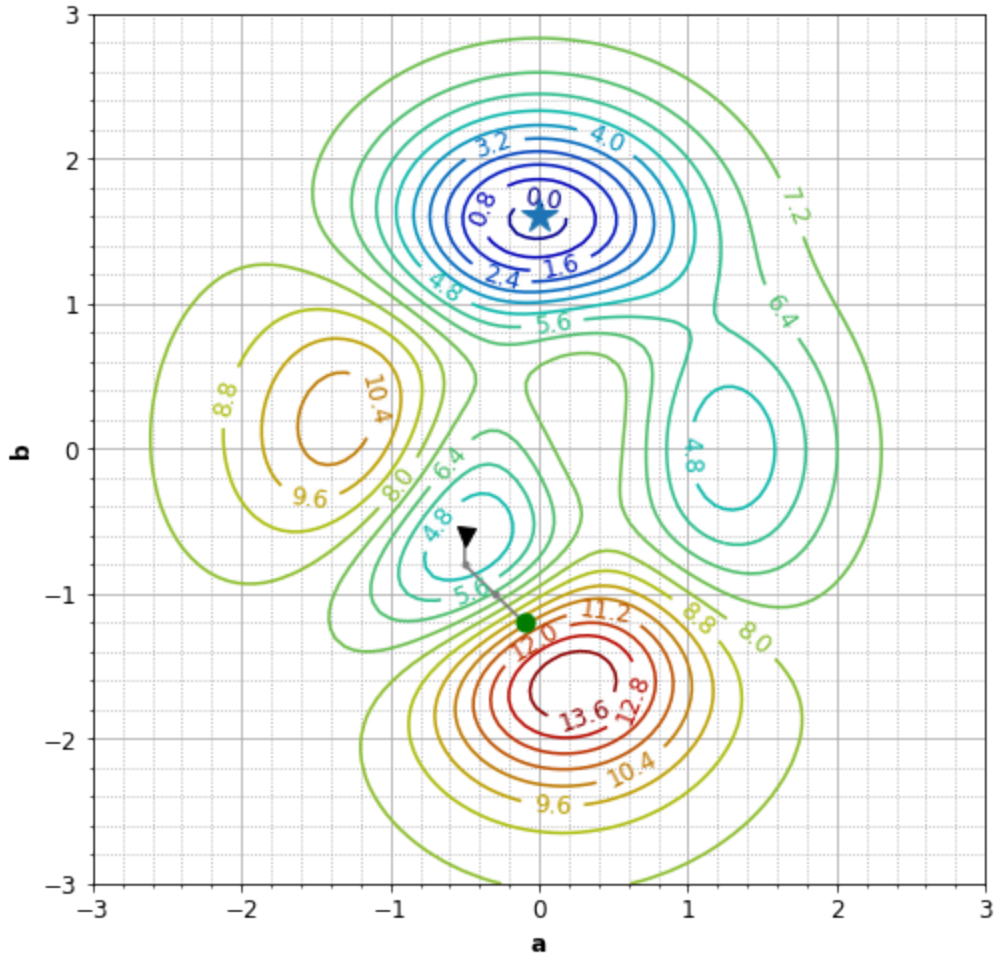

```
Final location (a,b) = (0, 1.5)
Final value of Error(a,b) = -0.197
```



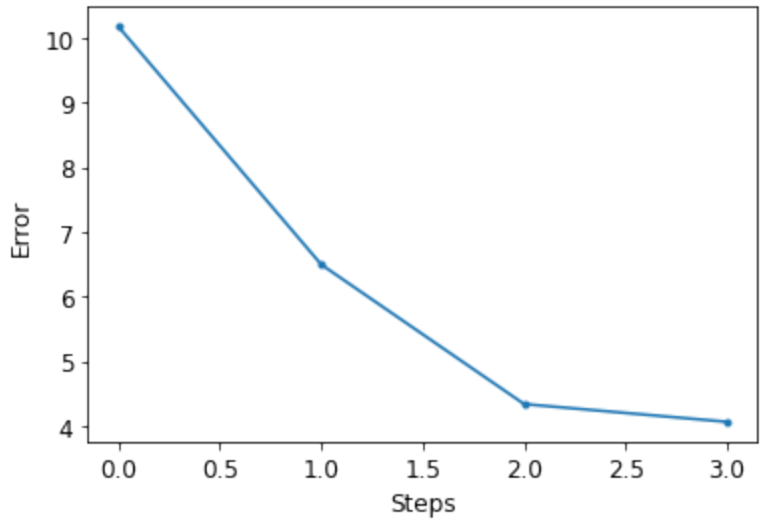


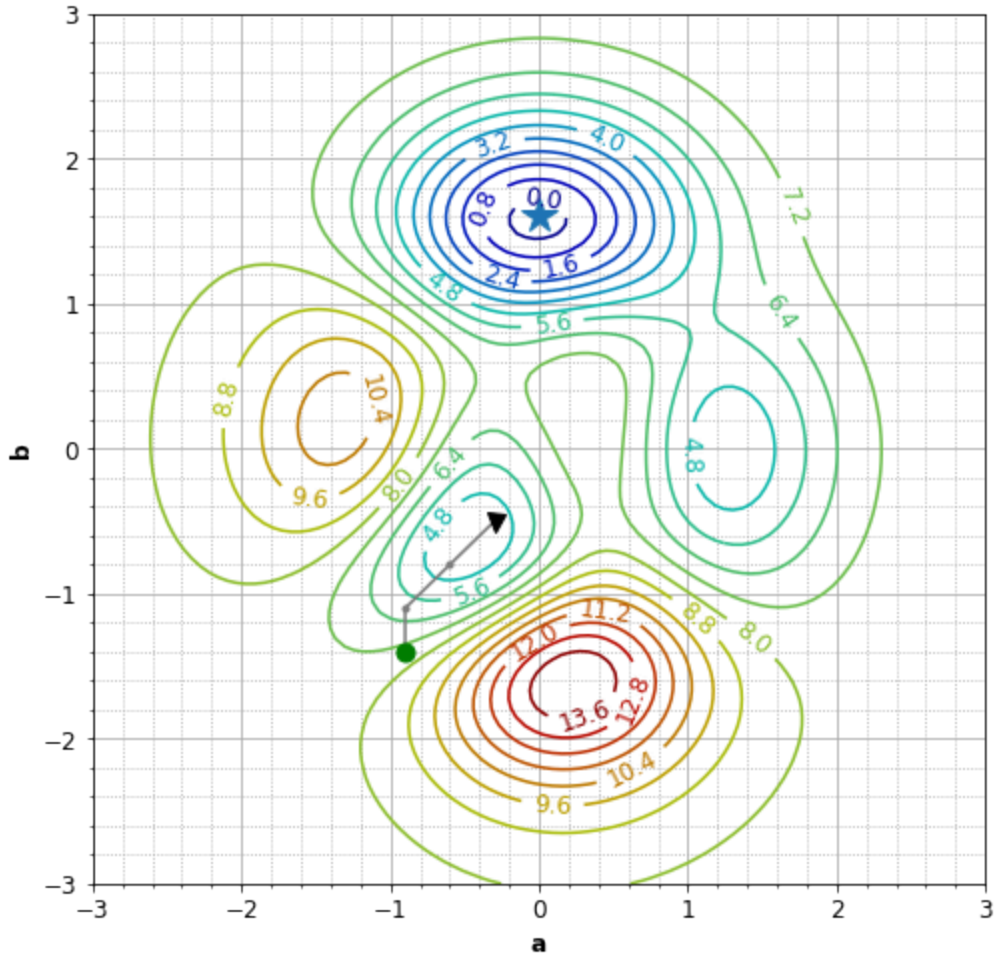

```
Final location (a,b) = (-0.4, -0.6)
Final value of Error(a,b) = 4.06
```



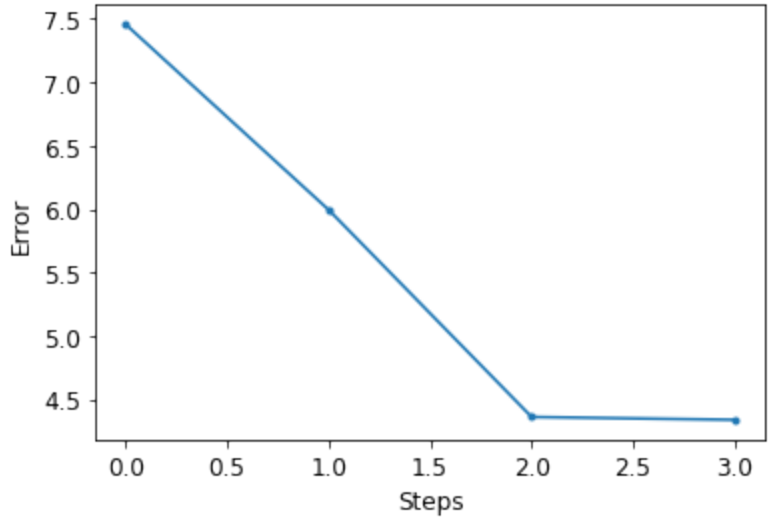


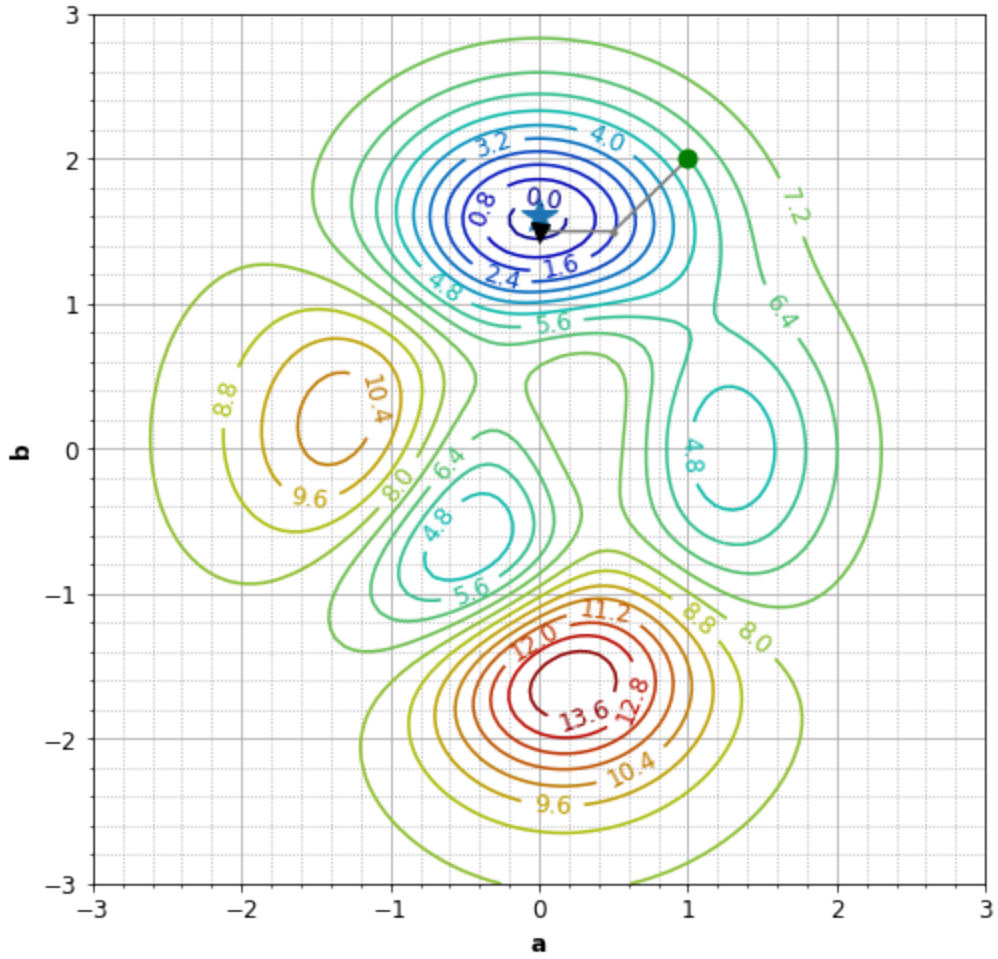

```
Final location (a,b) = (-0.5, -0.6)
Final value of Error(a,b) = 4.06
```



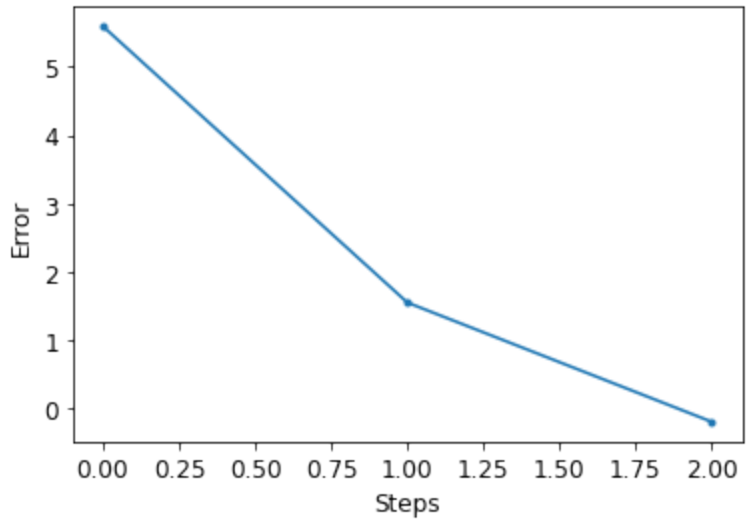


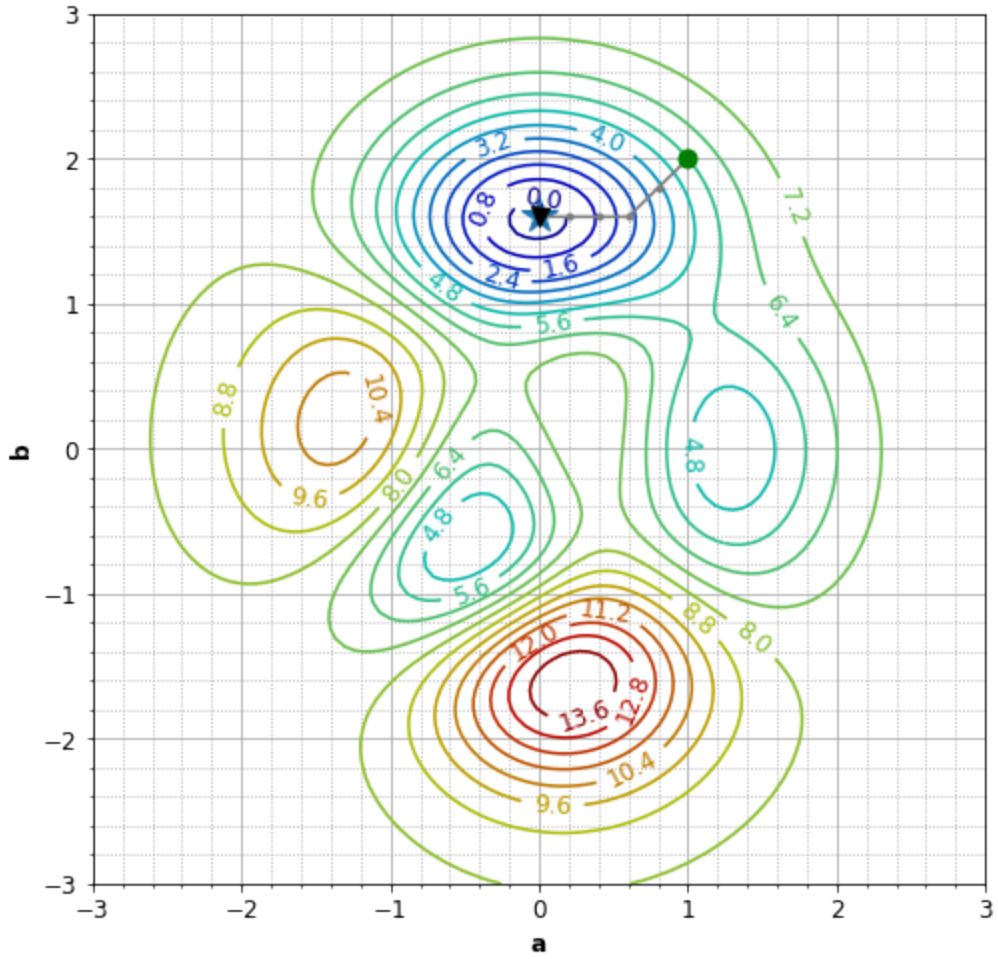

```
Final location (a,b) = (-0.3, -0.5)
Final value of Error(a,b) = 4.338
```



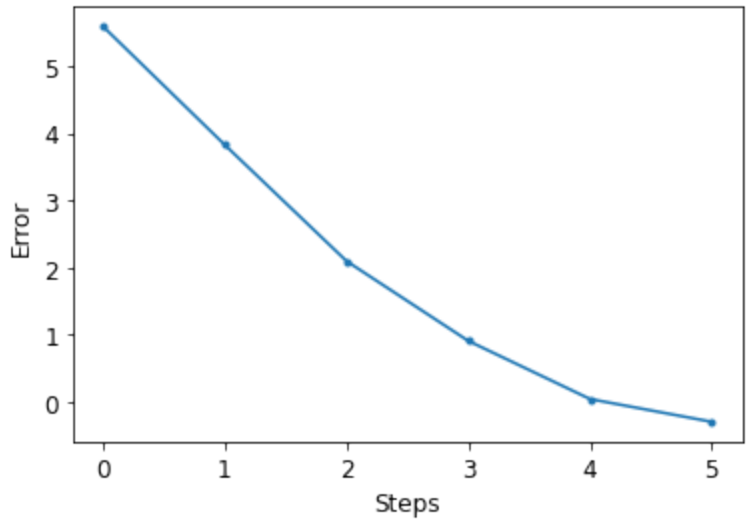



```
Final location (a,b) = (0.0, 1.5)
Final value of Error(a,b) = -0.197
```






```
Final location (a,b) = (0.0, 1.6)
Final value of Error(a,b) = -0.3
```



```
Final location (a,b) = (0.0, 1.6)
Final value of Error(a,b) = -0.3
```


Question 5 of 15

Optional challenge: Write your own algorithm in Python to solve the same optimization problem. It need not be a "hill climbing" algorithm at all [#algorithms]

- This problem will only be scored if completed with a sufficiently thorough explanation to describe how the code works, motivate why you are taking this approach, and justify that it is an improvement compared to the existing function.
- Comment on whether the original definition of the optimization problem from the first question needs modification based on your new approach
- As above, your explanation should comment on the algorithm's advantages, limitations, and ability to reach the global optimum.
- Also as above, be sure to test your algorithm and provide evidence that you tested it sufficiently.
- This will likely require going beyond class content, so you must cite all resources used outside of class materials following APA conventions and explain your work in sufficient detail so that another student could understand your approach.

Normal

One way of doing it is brute force.
First, it creates a grid of points within certain limits and with a specified resolution. At each point on the grid, it calculates the error using a specific function. This function tells us how good or bad a solution is. Lower error values indicate better solutions. It keeps track of the lowest error found as we traverse the grid. Once it searches the entire grid, it returns the coordinates of the point with the lowest error. This tells where the best solution is located.
My brute force algorithm creates a grid of points within given limits and resolution (I tried multiple resolutions, e.g. 10 led to an error of 0.64, while 20 gives an error of 0.24 which I found to be the lowest error when also taking into consideration the computational code). It checks the error at each point and keeps track of the lowest error found. In the end, it returns the coordinates of the point with the lowest error. I wanted simplicity and a guarantee of finding the global minimum, so this is what motivated me to choose a brute-force algorithm instead of any other type.
However, the algorithm becomes impractical for complicated problems with very large search spaces due to the increasing number of evaluations. Also, accuracy depends on the resolution and lower resolutions may miss narrow minima. I tested the algorithm with different resolutions and compared the solutions.

Another way to do it is by using a genetic algorithm (my algorithm is certainly not perfect and I do not have much coding experience so I tried my best to put my idea into practice based on what I found online).
I approached making a genetic algorithm by using the following steps:

1. Initialization: Initialize a population of solutions, each consisting of random values for variables 'a' and 'b'.
2. Evaluation: Calculate the fitness of each solution using the error function.

3. Selection: Select the best solutions as parents for the next generation based on their fitness.
4. Crossover: Combine parents to create offspring solutions.
5. Mutation: Introduce variability by randomly tweaking the offspring solutions.
6. Repetition: Repeat steps 2-5 for a certain number of generations.
7. Termination: Return the solution with the lowest fitness (error) as the best solution found.

This is how I approached each step in-depth:

1. Initialization:
    a. It starts with a population of potential solutions. Each solution is represented by a pair of values (a, b), where 'a' and 'b' are randomly chosen numbers within a certain range, in this case, between -3 and 3.
2. Evaluation (Fitness Calculation):
    a. Each solution's fitness is evaluated using a specific function called the error function. This function takes the values 'a' and 'b' of each solution and computes a numerical value that indicates how good or bad the solution is. Lower values represent better solutions.
3. Selection of Parents:
    a. Solutions with better fitness values have a higher chance of being selected as parents for the next generation. The selection process is biased towards solutions with lower error values, meaning solutions that are closer to the desired outcome.
4. Crossover:
    a. Selected parent solutions are combined to create offspring solutions. This is done by taking characteristics from each parent and combining them in some way. In this algorithm, the average of the 'a' and 'b' values of the parents is taken to produce a new solution.
5. Mutation:
    a. To introduce diversity into the population and prevent the algorithm from converging to local optima (suboptimal solutions), a random mutation is applied to some of the offspring solutions. This mutation randomly alters the 'a' and 'b' values of the offspring with a small probability.
6. Generation Update:
    a. The new offspring solutions, along with selected parent solutions, form the population for the next generation.
7. Repetition:
    a. Steps 2-6 are repeated for a certain number of generations or until a termination condition is met.
8. Result:
    a. After the number of generations, the best solution found (the one with the lowest error) is returned as the final output.

I chose this approach because genetic algorithms can effectively explore vast solution spaces, making them good for complex optimization problems that maybe brute force wouldn't be the best for. While not guaranteed to find the global optimum, this algorithm has a high probability of reaching good solutions, making them suitable for a wide range of optimization problems (in the case of the map given, I tried it 10 times, and each time it found a point very close to the global optimum). However, genetic algorithms may run slowly, especially for problems with rugged spaces. While brute force evaluates all possible solutions, genetic algorithms reduce computational complexity, which makes them more effective in this case.

I used this website to understand genetic algorithms better and try to do my own: Brownlee, J. (2021, March 2). *Simple Genetic Algorithm From Scratch in Python*. Machine Learning Mastery. https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/

Code Cell 6 of 8

In [8]:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d


# Error function as defined earlier
def error(a, b):
    func = 3*(1 - a)**2*np.exp(-a**2-(b+1)**2) - 10*(a/5-a**3-b**5)*np.exp(-a**2-b**2) - (1/3)*np.exp(-b**2-(a+1)**2)
    return -func + 7.8


# Brute force optimization function
def brute_force_optimize(f, a_limits=(-3, 3), b_limits=(-3, 3), resolution=20): #Defines a function brute_force_optimize that takes an objective function f, limits for the decision variables a and b, and a resolution determining how the search space is sampled.
    a_range = np.linspace(a_limits[0], a_limits[1], resolution) #Generates arrays a_range and b_range using np.linspace, which evenly spaces points between the given limits (a_limits and b_limits) according to the specified resolution.
    b_range = np.linspace(b_limits[0], b_limits[1], resolution)

    min_error = np.inf #Initializes min_error to infinity (np.inf) to ensure any first calculated error is smaller, and initializes min_a and min_b to None to store the coordinates of the minimum error found.
```

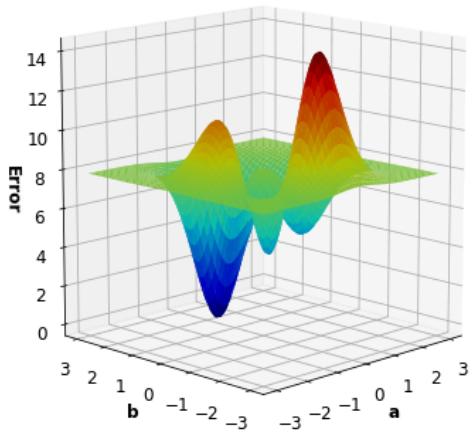
Python 3 (384MB RAM) | Edit Run All Cells Kernel Ready |

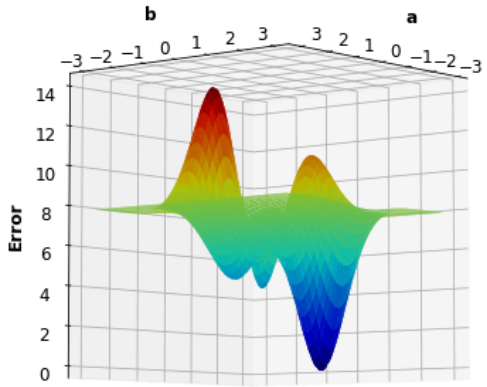
```
    min_a, min_b = None, None

    for a in a_range:
        for b in b_range:
            current_error = f(a, b) # loops iterate over every combination of a and b in the search space, calculating the error using the function f passed as an argument.
            if current_error < min_error:
                min_error = current_error
                min_a, min_b = a, b #If the current_error for a given combination of a and b is lower than the current min_error, update min_error and the corresponding coordinates min_a and min_b.


    return min_a, min_b, min_error #Returns the coordinates (min_a, min_b) of the minimum error found and the value of that min_error.


# Example
min_a, min_b, min_error = brute_force_optimize(error)  #Calls the brute_force_optimize function with the error function as the objective function to optimize. The function returns the coordinates of the minimum error and the error value, (note to self: change line 11 - done)

print("Minimum found at (a, b):", (min_a, min_b))
print("Minimum error:", min_error)

# Using the same plot functions to visualize as in the algorithm used above
plot2d(error, max=(min_a, min_b))
plot3d(error)

```

Run Code

Out [8]

```
Minimum found at (a, b): (-0.1578947368421053, 1.7368421052631575)
Minimum error: 0.24392782121182943
```





Code Cell 7 of 8

```
In [21]  1 import numpy as np
         2 import matplotlib.pyplot as plt
         3
```

Python 3 (384MB RAM) | Edit Run All Cells Kernel Ready | |

```
# Let's keep the error function unchanged since it's specific to the problem.
def error(a, b):
    return -(3*(1 - a)**2*np.exp(-a**2-(b+1)**2) - 10*(a/5-a**3-b**5)*np.exp(-a**2-b**2) - 1/3*np.exp(-b**2-(a+1)**2)) + 7.8


# Simplified functions for the genetic algorithm
def initialize_population_simple(size):
    """Create a starting population with random a and b values. Initializes a population of solutions where each solution is a pair (a, b), with a and b being random values between -3 and 3. The size of the population is determined by the size parameter.
"""
    population = []
    for _ in range(size):
        a = np.random.uniform(-3, 3)
        b = np.random.uniform(-3, 3)
        population.append((a, b))
    return population


def calculate_fitness_simple(population):
    """Calculate the fitness for each solution."""
    fitness = []
    for individual in population:
        fitness.append(error(individual[0], individual[1]))
    return fitness #Calculates the fitness of each individual in the population by applying the error function to their a and b values. Lower values indicate better fitness in the context of searching for a minimum.


def select_parents_simple(population, fitness, num_parents):
    """Select the best solutions to be parents. Calculates the fitness of each individual in the population by applying the error function to their a and b values. Lower values indicate better fitness in the context of searching for a minimum."""
    # Adjusted to prioritize lower fitness values
    sorted_population = sorted(zip(population, fitness), key=lambda x: x[1], reverse=False) #organizes pairs of individuals from the population with their corresponding fitness values. These pairs are arranged in order of their fitness values, from lowest to highest.
    selected_parents = [individual for individual, _ in sorted_population[:num_parents]] #ick out the best-performing individuals from the sorted list. These individuals are chosen to be the parents for the next generation of solutions. We only select a certain number of parents, determined by num_parents.
    return selected_parents


def crossover_simple(parents, num_offspring):
```

```
    """Combine parents to create offspring. Selects the best solutions from the population to be parents for the next generation. It sorts the population based on fitness (with better fitness having lower error values) and selects the top num_parents."""
    offspring = []
    for _ in range(num_offspring):
        # Use random.sample to select two unique parents for crossover
        parent1, parent2 = random.sample(parents, 2) # randomly selects two parents from a list of parents. random.sample(parents, 2) means it randomly picks two parents from the parents list. The selected parents are assigned to the variables parent1 and parent2.
        child = ((parent1[0] + parent2[0]) / 2, (parent1[1] + parent2[1]) / 2) #For example, the first element (a) of the child is the average of the first elements of parent1 and parent2, and the second element (b) of the child is the average of the second
        offspring.append(child) # adds the newly created child (with its a and b values) to the list of offspring. So, after this line, the list of offspring will have one more child in it.
    return offspring


def mutate_simple(offspring):
    """Randomly tweak the solutions. Randomly alters the a and b values of the offspring with a 10% chance to introduce variability and prevent the algorithm from getting stuck in local minima.
"""
    for i in range(len(offspring)): # sets up a loop that will iterate over each individual in the list of offspring. It's going to check each child one by one.
        if np.random.rand() < 0.1:  # randomly generates a number between 0 and 1. If this randomly generated number is less than 0.1, meaning there's a 10% chance (0.1 probability), the mutation process will occur. So, it's like flipping a coin where there's a 10% chance of it landing on heads.
            offspring[i] = (offspring[i][0] + np.random.uniform(-0.1, 0.1), offspring[i][1] + np.random.uniform(-0.1, 0.1)) # If the condition above is met (the random number is less than 0.1), then mutation happens. For each offspring, their a and b values are modified randomly. This line adds a small random value (uniformly sampled from -0.1 to 0.1) to both a and b values of the offspring.
    return offspring


import numpy as np
import matplotlib.pyplot as plt
import random


# Assume all other functions (initialize_population_simple, calculate_fitness_simple, select_parents_simple, crossover_simple, mutate_simple) are defined as before
```

```
58 def genetic_algorithm_simple(generations=10, population_size=10,
num_parents=5):
59     population = initialize_population_simple(population_size)
60     best_fitness_history = []  # To track the best fitness over generations
61
62     for _ in range(generations):
63         fitness = calculate_fitness_simple(population) #the fitness values for
each individual in the current population are calculated.
calculate_fitness_simple function is called with the current population as
input, and it returns a list of fitness values corresponding to each individual
in the population.
64         best_fitness_history.append(min(fitness))  # Assuming lower fitness is
better  ---  The minimum fitness value among all individuals in the current
population is added to a list called
65         parents = select_parents_simple(population, fitness, num_parents) # The
best individuals (parents) from the current population are selected based on
their fitness values. The select_parents_simple function chooses the top
individuals from the population to be parents for the next generation.
66         offspring = crossover_simple(parents, population_size - num_parents)  #
Using the selected parents, new offspring are generated through crossover. The
crossover_simple function combines pairs of parents to create new individuals
(offspring) for the next generation.
67         population = parents + mutate_simple(offspring)  #The new population
for the next generation is formed by combining the selected parents with the
mutated offspring. The mutate_simple function introduces random changes to some
of the offspring to add diversity to the population and prevent convergence to
local optima.
68
69     final_fitness = calculate_fitness_simple(population) #final_fitness list,
which corresponds to the index of the individual with the lowest fitness in the
population.
70     best_solution_index = final_fitness.index(min(final_fitness))
71     best_solution = population[best_solution_index]
72
73     # Plot the best fitness over generations
74     plt.figure()
75     plt.plot(best_fitness_history)
76     plt.xlabel('Generation')
77     plt.ylabel('Best Fitness')
78     plt.title('Best Fitness Over Generations')
79     plt.show()
80
81     return best_solution, final_fitness[best_solution_index]
82
83 # Run the genetic algorithm
84 best_solution, best_fitness = genetic_algorithm_simple(generations=17,
population_size=20, num_parents=10)  #  runs the genetic algorithm with
```

```
selected per generation). It returns the best solution found by the algorithm (best_solution) and its corresponding fitness value (best_fitness).
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)

def plot_error_landscape(): #create a vizualizaion of the landscape with values between -3 and 3
    a = np.linspace(-3, 3, 100)
    b = np.linspace(-3, 3, 100)
    A, B = np.meshgrid(a, b)
    Z = error(A, B)  # Assuming the error function is defined

    plt.figure(figsize=(10, 7))
    cp = plt.contourf(A, B, Z, levels=15, cmap='viridis')
    plt.colorbar(cp)
    plt.title('Error Function Landscape')
    plt.xlabel('a')
    plt.ylabel('b')

    # Plot the best solution on the landscape
    plt.scatter(best_solution[0], best_solution[1], color='red', label='Best Solution')
    plt.legend()
    plt.show()

# Call the function to plot the error landscape with the best solution
plot_error_landscape()
```

Run Code

Out [21]


```
Best Solution: (-0.005711523009802941, 1.5804561729014868)
Best Fitness: -0.30609258379428805
```


Code Cell 8 of 8

In [10]

```
1 # optional new optimization algorithm implementation and testing
```

Run Code

PART 2: Hill Climbing in San Francisco

Write an algorithm in natural language to find San Francisco's highest peak (or lowest valley - you choose). Start with a basic hill climber and add extensions as you wish. It might help to consider one block to be one step of the algorithm, meaning that each intersection is one (x,y) coordinate on the grid. Here's an example to get you started.

1. Pick a starting point:
    a. Ask six different people for their favorite place in the city and number them 1-6.
    b. Roll a 6-sided die to pick one of these as your starting location.
    c. Go to the closest intersection.
2. Decide where to go next: Examine all directions you could go next (forward, backward, left, or right), and determine if you can identify the direction with the steepest ascent.
    a. If there is one direction with an obviously steepest incline, turn to face this direction, and roll a 6-sided die.
        i. If the result is 1-4: Go straight up the identified steepest direction until you arrive at the next intersection. Repeat step 2.
        ii. If the result is 5-6: Roll the die again.
            1. If the result is 1-2: go right to the next intersection. Repeat step 2.
            2. If the result is 3-4: go left to the next intersection. Repeat step 2.
            3. If the result is 5-6: go backward to the next intersection. Repeat step 2.
    b. If two directions appear to be equally steep, flip a coin to decide which direction to face. Turn to face the selected direction, roll a 6-sided die, and follow the outcomes in step 2a.
    c. If there is no direction that ascends to a higher elevation, roll a 6-sided die.
        i. If the result is 1-5: Declare that you have found the highest peak and terminate the algorithm. You're done!
        ii. If the result is 6: Close your eyes and spin around for 30 seconds. Open your eyes and go in the direction that is most aligned with the direction you are currently facing until you reach the next intersection. Repeat step 2.

Guidelines for your algorithm:

1. *Collaboration*: As an option, you're welcome to work with a partner on writing the algorithm. If you do, please state this clearly here and in the acknowledgments section below. You're still

2. *Length*: Once you start writing your algorithm, you might feel compelled to add more detail, conditional clauses, and steps. While these can be legitimate improvements, we need to draw the line somewhere. Please try not to exceed 500 words total for your algorithm.
3. *Good algorithms*: Strive to adhere to the properties of good algorithms that we learned in class. Review the class readings, activities, and the #algorithms HC Handbook.
4. *Robustness*: The goal of this exercise is not necessarily to produce the best most robust algorithm. You're expected to think through possible edge cases, but it will be virtually impossible to account for all of them. It's important to demonstrate that you've put consideration into a few different edge cases, but if your algorithm ends up failing upon real-life implementation, it doesn't mean that the assignment is a failure! Above all, this is meant to be a learning experience. The assessment will reflect the depth of your analysis above the perfection of the algorithm.

Question 6 of 15

Write the algorithm here.

Normal

Set a maximum number of repetitions of this algorithm depending on your time constraints:

Step 1. Gather Local Recommendations

- Make a list of all San Francisco neighborhoods (e.g. The Presidio, SoMa, Mission District, Chinatown, Fisherman's Wharf, etc)
- Put the list through a random name selection program like Wheel of Names
- Randomly turn the online wheel and pick 1 neighborhood.
- Go to the neighborhood and ask one person on the street "Where is your favorite place in this area?"
- Proceed to the nearest intersection from this chosen location to begin exploration.

Step 2: Core Exploration Loop

1. Visual Assessment at Intersections:
    a. Stand at each intersection and look around in all four directions: north, east, south, and west.
2. Movement Decision:
    a. Roll a 6-sided die to decide your next move.
        i. If you roll a number from 1 to 5:
            1. Look for the direction that appears to have the steepest incline. Is one path noticeably steeper than the others?
                a. If only one direction seems steeper,

i. If you reach an intersection you've previously visited, take the next steepest incline that you haven't explored yet, instead of repeating any prior paths.
ii. If you've never visited that intersection, move in that direction to the next intersection.

b. If multiple directions seem equally steep
i. if there are people around you: ask someone nearby for advice. You could say, "Which way do you think is the highest hill?" or "Which path looks like it goes up the most?"
ii. If there are no people around you:
1. assign numbers from 1 to 4 to the four directions
2. use a random number generator to choose the number from 1 to 4
3. go in the chosen direction

c. If you can't find any direction with a higher incline
i. Look around and assess if you are indeed at the highest peak. If you visually see any place higher than when you currently are, locate it on the map and move in that direction. If you can't find any other higher peak:
1. Roll a dice
a. If you roll a 1, 2, 3, or 4: Assume you've reached a peak and move to Step. 3: Verification.
b. If you roll a 5, or 6:
i. assign numbers from 1 to 4 to the four directions
ii. use a random number generator to choose the number from 1 to 4
iii. go in the chosen direction

ii. If you roll a 6:
1. assign numbers from 1 to 4 to the four directions
2. use a random number generator to choose the number from 1 to 4
3. go in the chosen direction

b. Return to the beginning of the Exploration Loop if you have not moved to Step. 3: Verification.

1. Peak Verification: Once a peak is presumed found, record its location and height for later comparison. If the maximum number of repetitions initially set isn't met, return to Step 1.
2. Consult all recorded peaks and select the highest one as the exploration's culmination point (use the FreeMapsTool to find your current elevation)

Question 7 of 15

Define the optimization problem that your algorithm is trying to solve: What is the objective function? What are the decision variables? Are there any constraints? Clearly articulate each component so that it's clear how the objective value would be measured and how the decision variables would impact it. How is this version of the problem different than the one above? (<200 words) [#optimization].

Normal

The objective of the algorithm is to identify the highest peak in San Francisco based on the recommendations gathered from locals. The **objective function** is to maximize the elevation of the peak reached during exploration. The value would be measured using the FreeMapsTool to quantify the elevation of the presumed highest point found.

The **decision variables** include initial neighborhood choice, direction at each intersection, and input from locals. The algorithm randomly selects the starting point

These variables are under the user's control and directly affect the elevation outcome. They are categorical and discrete. They are independent in the sense that a choice of direction at one intersection does not limit future choices at other intersections.

Direct Constraints: Must stay within San Francisco neighborhoods, follow pedestrian routes, and adhere to dice roll results for movements.
Indirect Constraints: Depend on local knowledge, which is influenced by whom you ask, and the reliability of the FreeMapsTool for elevation verification.

My version starts with local recommendations to select a neighborhood and location starting point, which could lead to more diverse areas if the algorithm is done multiple times. In the movement decision loop, my algorithm avoids repeated paths by taking the next steepest incline that hasn't been explored, reducing redundancy. When faced with equally steep inclines, my algorithm suggests asking locals for advice, adding a qualitative aspect to the decision-making process. I also added a verification step using visual

assessment and a mapping tool, which provides more objective confirmation of the peak's elevation.

Question 8 of 15

Write 1-2 paragraphs to justify your algorithm. Why did you write it this way? What are its advantages? What are its drawbacks? See the guidelines above (<200 words) [#algorithms].

Normal

In my algorithm, I check the slope in all four directions at every intersection, giving a good understanding of the land. This helps make smarter decisions about which way to go, which can lead us to the highest peak more accurately. When I am unsure because multiple paths seem equally steep, I ask locals for advice if this is an option, or randomly choose a direction. The direction isn't chosen by closing my eyes and spinning around for a few seconds because this can be ineffective (the turning speed is likely to be almost the same every time I do it so it may not be truly random. The locals' knowledge of the area can guide us better than relying solely on random guesses or personal judgments. Since it is a real-life algorithm, I decided to take advantage of human intelligence so, by only allowing randomness on a roll of 6, the algorithm focuses more on exploring the terrain based on observed inclines (the chance of taking a random step is now 1/6 at every move and not 1/3). Allowing randomness on rolls of both 5 and 6 could encourage more exploration, potentially uncovering new areas, but it might also hinder efficient climbing. Random choices could divert from optimal paths, this is why I limited my use of randomness.

The algorithm, however, doesn't guarantee finding the global optimum. The algorithm might take a while to finish, especially if the neighborhood is large or if the exploration loop repeats several times. Depending on human judgment, like visually assessing slopes or asking for directions, introduces subjectivity that could affect the results. It's best suited for hilly neighborhoods and might not work as well in flat areas or places with limited pedestrian access.

Question 9 of 15

It's time to go test your algorithm in real life.

- Aim to follow it exactly as written, but you can patch up small bugs as you go if needed.
- Your algorithm will not be perfect, and that's ok! We expect edge cases and failure modes to

- Pay attention to your surroundings at all times. Do not do anything unsafe or illegal (e.g., jaywalking, trespassing).
- Do not spend the entire day on this activity. If your algorithm is still running after 4 hours, initiate manual termination, grab a snack, and just work on your notes and reflections.
- Document your experience with at least three photos, one of which must include yourself (a selfie).
- Include an image of a map of your journey. Clearly label the starting and ending points.
- Make notes as you go about the experience. Paste those notes here or take a photo/screenshot and upload them along with the photos below.

Normal

The first location I randomly selected was the Financial District so I went around on Montgomery St and asked a stranger about their favorite place in the district. They told me they liked the Palace Hotel so I started there. I went to the closest intersection and did my algorithm from there. The die rolled 6 two times during my road, so I took a random path twice. Once, on Sutter St., when I was unsure of the incline, I decided to ask a local and they told me to go on Mason St. I ended up on California St. and I think this is a local maximum in the neighborhood (even though it is in Nob Hill). I couldn't find any obvious incline so I rolled a digital die and got a 2, which means I recorded my elevation and wrote it on the map.

The second neighborhood I randomly selected was the Castro District and, when I got there, the person I asked told me their favorite place is the Castro Theater. I started from there, but this route was definitely more challenging. I followed the steepest hills which was straightforward until I ended up in the beginning of the Corona Heights hike. Since my algorithm was based on intersections, I couldn't fully follow it. I just followed the path, and every time there was an intersection I chose the highest peak. Some inclines didn't seem safe so I decided to not fully follow the algorithm in these cases. When I reached the top, I actually went in a circle because the steepest incline wasn't always evident or safe. In the end, I ended up very close to the Corona Heights viewpoint. I rolled a digital die and got 4, so I recorded my elevation.

Because of time constraints, I only did the algorithm twice (it already took me around four hours to perform these two and I did them on two different days. The second day's peak was higher (412 feet) so I concluded that it was my highest found hill.

Question 10 of 15

Upload three photos from the trip here, one of which must include yourself (a selfie). If you didn't

20240218_125905.jpg (7.38 MB)

20240218_130429.jpg (6.28 MB)

Screenshot_20240219_205851_Instagram.jpg (996 kB)

20240218_130819.jpg (8.31 MB)

20240218_130816.jpg (5.12 MB) ×

Drop or upload a file here

Question 11 of 15

Upload an image of a map (e.g., Google Maps) that clearly labels the starting and ending points and pathway taken.

86.7 m or 284.3 feet (3).pdf (609 kB) ×

*For remote students located outside San Francisco or students who are unable to complete the algorithm implementation for other reasons, consider the following options and discuss them with your professor at least one week prior to the assignment due date (Thurs Feb 15).

- If you're studying remotely, complete the assignment in your own location, making suitable adaptations depending on the terrain of your city or town.
- Test out the algorithm *virtually* in San Francisco using Google Maps Street View.

Question 12 of 15

Reflect on your experience (< 250 words) [#algorithms]:

- How did it go? Tell us about which parts of the algorithm worked well and which parts were problematic.
- Was it overall successful? Did you arrive at a local or global optimum?
- What key insights about #optimization and #algorithms did you glean from this activity?
- In hindsight, what changes would you make to your existing algorithm?

Normal

I didn't reach the highest point, Mount Davids at 928 feet, only making it to 412 feet. I believe that if we repeat the algorithm more times, the chances of getting to the global optimum get higher, but so does the time we spend on execution (if we end up going to every neighborhood in SF, then the algorithm would be very similar to a brute force, which is definitely not optimal in real life because of time resources). A big problem with the

algorithm is that it assumes intersections. In an urban US city like San Francisco, the algorithm can work will in finding at least a pretty high local maximum, however, in European cities or in regions in the US that are mountain-like/beach-like and don't have intersections, it would pose a lot of problems. The algorithm, as it is currently, is not generalizable. If I were to optimize it, I would define multiple step size types that are different from street intersections based on my climate. For example, I would choose three steps in one direction to be my step size or even one step, since it would be hard in a mountain region to just go in a single direction. I would also add more than the 4 directions the algorithm assesses (for example, add diagonals between the cardinal points as I did in Question 3).

Another thing I would change is taking advantage of even more of human intelligence. . If you've hit the peak but suspect there's a higher point nearby, you should be free to explore that direction. You'd still log your current elevation but also rely on human intuition. For example, I spotted Russian Hill from California St, knowing it's higher, but the algorithm didn't let me head there.

Optional safety concern problem: Another mainly problematic part was that asking strangers for recommendations put me at risk especially when I did this in Union Square since I looked like I didn't know where I was going. This algorithm is a little dangerous depending on the area it randomly generates and I think the safest option would be to follow it in a group rather than individually.

ACKNOWLEDGMENTS

We thank Irhum Shafkat (M24) for helping to write the hill climbing functions and visualizations in this workbook.

Question 13 of 15

Now it's your turn. Keeping the FA assignment tradition, let's end with acknowledgments. Give attribution to the people and/or external resources that you used to complete it. This doesn't have to be an exhaustive list, but is a chance for an honest reflection and recognition. Conversations with peers and visits to office hours can be mentioned here but do not require a formal APA citation. If you collaborated on writing the location-based natural language algorithm, describe what you learned from your peer and how you worked together.

As part of this reflection, please include your note on your use of AI Tools, as required for all MU

Your FA professors do not have specific suggestions regarding how to use AI tools for this assignment. Before considering whether or not to use AI for the assignment, reflect on the purpose of the assignment (learning how to use statistics tools to uncover trends in data) and use metacognition to ensure your process for working on the assignment aligns with this purpose (#selfawareness). To that end, we encourage you to rely first and foremost on the class resources that have been curated for this course. If you believe you have an idea for how to use AI in a way that enhances your learning and does not diminish the goals of this assignment, the reflection here must present a compelling argument for this. It's also imperative to fact-check any output from the AI along the way with external validation. **Here, fully document your methods as required by MU's AI policy, including the prompt(s), the output(s), how you modified the output(s), how you fact-checked or tested the output(s), and why it was beneficial for your learning**. We also suggest discussing this with your professor well in advance (5 days or more) of the due date.

Normal

AI Statement: I only used Grammarly for checking my spelling and grammar errors.

References:
Brownlee, J. (2021, March 2). *Simple Genetic Algorithm From Scratch in Python*. Machine Learning Mastery. https://machinelearningmastery.com/simple- genetic-algorithm-from-scratch-in-python/

PYTHON TIPS

The following tips and reminders may be useful:

- Talk through the code with your peers and Peer Tutors, TAs, and Professors in their office hours. This is perfectly acceptable as long as you are an active participant, and your written work is completely your own. Be mindful of the guidelines for appropriate collaboration on assignments (see Q6 and A6 in the Avoiding Plagiarism guide), such as always starting and finishing the assignment by yourself, avoiding looking directly at other student's work, and avoiding directly sharing/sending your work to others.
- Reminder: no matter what, your code needs comments. Read this resource about the importance of comments and this one for further guidance.
- If you used any coding resources that are external to class materials, please cite them following APA convention in your acknowledgments.

HC GUIDELINES

Targeted HCs: You will be graded on #optimziation and #algorithms as indicated above in square brackets, at the discretion of the professor, as well as #professionalism. "Footnotes" for these are *not required*. Strong applications on this assignment will result from following the assignment instructions, reviewing class readings and activities, and integrating feedback from previous HC assessments. The HC Handbook can provide further guidance, especially the "Applying the HC" section. The optional challenges will only be given extra HC scores for correct and well-explained implementations that demonstrate deep knowledge.

Other HCs: Please focus on the targeted HCs for this assignment - this is your best chance to practice them and receive feedback! However, if you believe that you have strong applications of other HCs**,** such as #designthinking or #breakitdown, you can *optionally tag up to one additional HC* with an explanation (<150 words). The explanation should showcase your understanding of how the HC is being applied in your assignment. This is meant to go beyond simply *identifying* that an HC applies to actually analyzing how the core essence of the HC is being used to elevate your work. You can think of this as beefy footnotes that are richer and more robust. Strive to make your explanation concrete by referencing exactly where the HC was applied in your assignment and supplying additional support and justification as needed. Avoid general descriptions of the HC and strive to be *specific* about how the HC was *applied*!
To maximize your HC explanations, you're encouraged to review the outcome index, your readings, class activities, feedback, handbooks, etc to ensure you are capturing the essential elements of the HC. In particular, The HC Handbook's "Applying the HC" section will help steer your applications in the right direction. For further advice and guidelines on how to integrate HCs in your work and write meaningful HC annotations, please watch these videos, "How to Apply HCs in Assignments" and "All About Footnotes," from the Academic Foundations series. Please also see this guide, "Addressing HCs in your work," noting the importance of the "*prospective approach*" detailed in section 2.1 and the importance of annotations as described in section 3. For more samples of effective annotations, refer to the footnotes that go along with each General Example in the HC Handbook.

Question 14 of 15

Optional: Provide a short <150 word explanation of an additional HC application within this assignment following the guidelines above.

Normal

Question 15 of 15

ANYTHING ELSE?

Optional: Use the box below to upload any supplemental files needed. For example, you might use this cell to upload supplementary code in an ipynb file that wouldn't fit in the code cells above.

Normal

YOU'RE DONE

Be sure to review all of your work and follow the formatting guidelines above before you submit.