AMATH301_Homework4_writeup

February 3, 2023

1 Homework 4 writeup solutions

1.1 Name: Aqua Karaman

1.2 Problem 1

1.2.1 Part a

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
import scipy.optimize
from mpl_toolkits.mplot3d import Axes3D

############## Problem 1 ################
## Part a
# Define x(t) below
x = lambda t: 11/6*(np.exp(-t/12) - np.exp(-t))

# Define x'(t) below. I did this one for you.
dx = lambda t: 11*(-1/12*np.exp(-t/12) + np.exp(-t))/6

# You need to do something to find the *maximum* using the *minimization*
# algorithms. How can you define a new anonymous function to make this work?
# Hint: you can define one anonymous function from another!

A1 = dx(1.5)

# Example: We use scipy.optimize.fsolve to find zeros of a function. For␣
↪instance, if
example = lambda t: t*(t-1)
# We know that there are two zeros: one at t=0 and one at t=1. We can find the
# zero at t=1 by choosing a guess close to 1.
# The syntax is scipy.optimize.fsolve(anonymous_function, guess)
dx_roots = scipy.optimize.fsolve(dx, 1.5)
A2 = dx_roots[0]
A3 = x(dx_roots[0])
print('The root near t = 1.5 of our example is =', dx_roots[0]) # Note that this
                             # is an array. To get the answer I have to index.
```

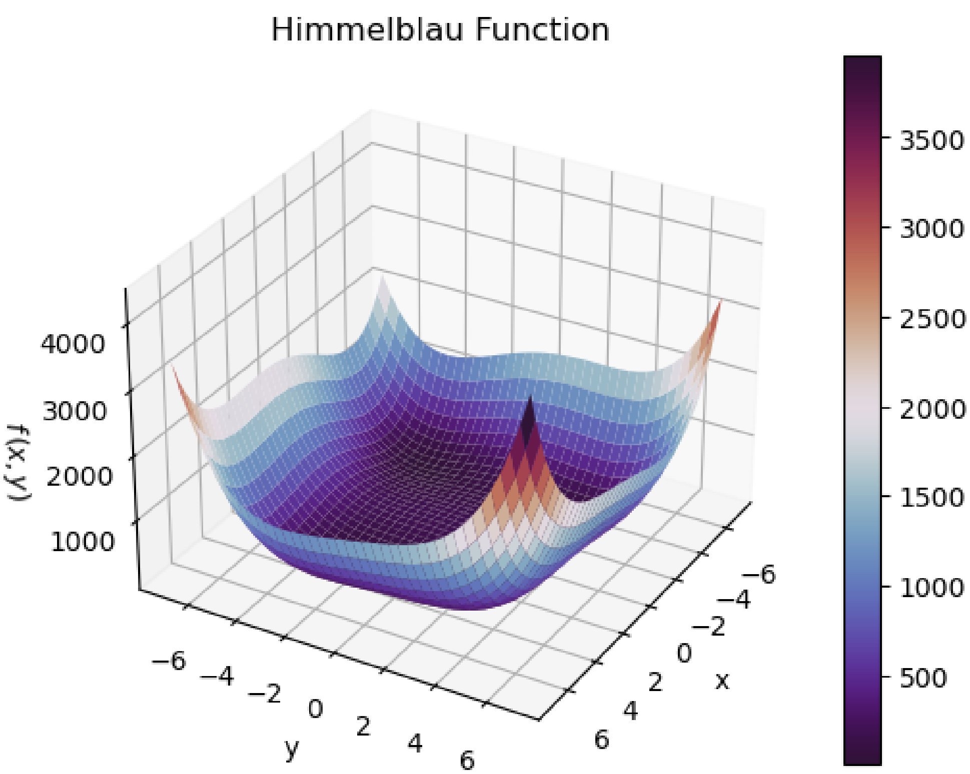
```python
## Part b
# Look at some examples of how we have used fminbound in class, for example on
# January 23, 24, or 27.
neg_x = lambda t: -11/6*(np.exp(-t/12) - np.exp(-t))
x_roots_data = scipy.optimize.fminbound(neg_x, 0, 10)
x_max_data = np.array([x_roots_data, x(x_roots_data)])
print('x func max data:', x_max_data)
A4 = x_max_data

############## Problem 2 ###############
## Part a
# Define Himmelblau's function using lambda x, y: ... first, and then use an
# adapter function! The adapter function is below, you need to define fxy and
# then you can uncomment the following line.
fxy = lambda x, y: (x**2+y-11)**2+(x+y**2-7)**2

f = lambda p: fxy(p[0], p[1]) # Assuming that fxy is defined in terms of x
                                # and y. Once you have that defined, uncomment
                                # this line.
A5 = f([3, 4])
print('test value for f function:', A5)

## Part b
# Recall that the syntax for scipy.optimize.fmin
# is scipy.optimize.fmin(anonymous_function, guess), where anonymous_function
# has to be a function of one variable.
argmin_f = scipy.optimize.fmin(f, [-3, -2])
A6 = argmin_f
print('argmin is', A6)

## Part c
# I'll start out this one by typing out the gradient, to limit the number of
# typos.
gradf_xy = lambda x,y: np.array([4*x**3 - 42*x + 4*x*y + 2*y**2 - 14,
                                 4*y**3 - 26*y + 4*x*y + 2*x**2 - 22])
# Now you need to turn it into a function of one variable using an adapter.
gradf = lambda p: gradf_xy(p[0], p[1])
A7 = gradf(argmin_f)
A8 = np.linalg.norm(A7)
print('2-norm is ', A8)

## Part d
# I'll start you off with a skeleton. You need to fill in parts. This is
# commented to start so that the code runs, you will need to uncomment it to
# use it.
```

```python
p = [-3, -2] # Initial guess defined in part (e)
tol = 1e-7 # you need to define tol!
phi = lambda t: p - t*grad
f_of_phi = lambda t: f(phi(t))
for k in range(2000): # perform 2000 iterations
    # Check if the gradient is small
    grad = gradf(p)
    tmin = scipy.optimize.fminbound(f_of_phi, 0, 1)

    if np.linalg.norm(grad)<tol:
        iter_num = k
        print('parameters are', p,'.')
        print('in coordinate form (x, y, z), the parameters are (', p[0], ', ',␣
 ↪p[1], ', ', f(p),').')
        print('loop has performed', iter_num, 'iterations.')
        break

    p = phi(tmin)

    # Do the steps here to redefine p.

## Part e
# Done above!
A9 = p
A10 = iter_num

# First create x
x = np.linspace(-7, 7, 40)
# Now you create y
y = np.linspace(-7, 7, 40)

# Once you have created them, you can uncomment and run the
# following line of code.
X, Y = np.meshgrid(x, y)

# Setup the figure
fig = plt.figure() # Create a figure
ax = plt.axes(projection='3d')  # Make it a "3D" figure
ax.view_init(30,30)
# ax.set_zlim([-1,3000])
plt.xlabel('x')
plt.ylabel('y')
ax.set_zlabel('$f \, (x,y)$')
plt.title('Himmelblau Function')
# Then you do the rest.
himmelblau = ax.plot_surface(X, Y, f([X, Y]), cmap='twilight_shifted',␣
 ↪label='f(x, y)')
```

```
fig.colorbar(himmelblau)
```

```
The root near t = 1.5 of our example is = 2.710807254314182
x func max data: [2.71080562 1.34074284]
test value for f function: 148
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 36
         Function evaluations: 69
argmin is [-3.77933378 -3.28318789]
2-norm is  0.002727728345314524
parameters are [-3.77931025 -3.28318599] .
in coordinate form (x, y, z), the parameters are ( -3.77931025348254 ,
-3.2831859911601855 ,  1.711579641918181e-18 ).
loop has performed 11 iterations.
```

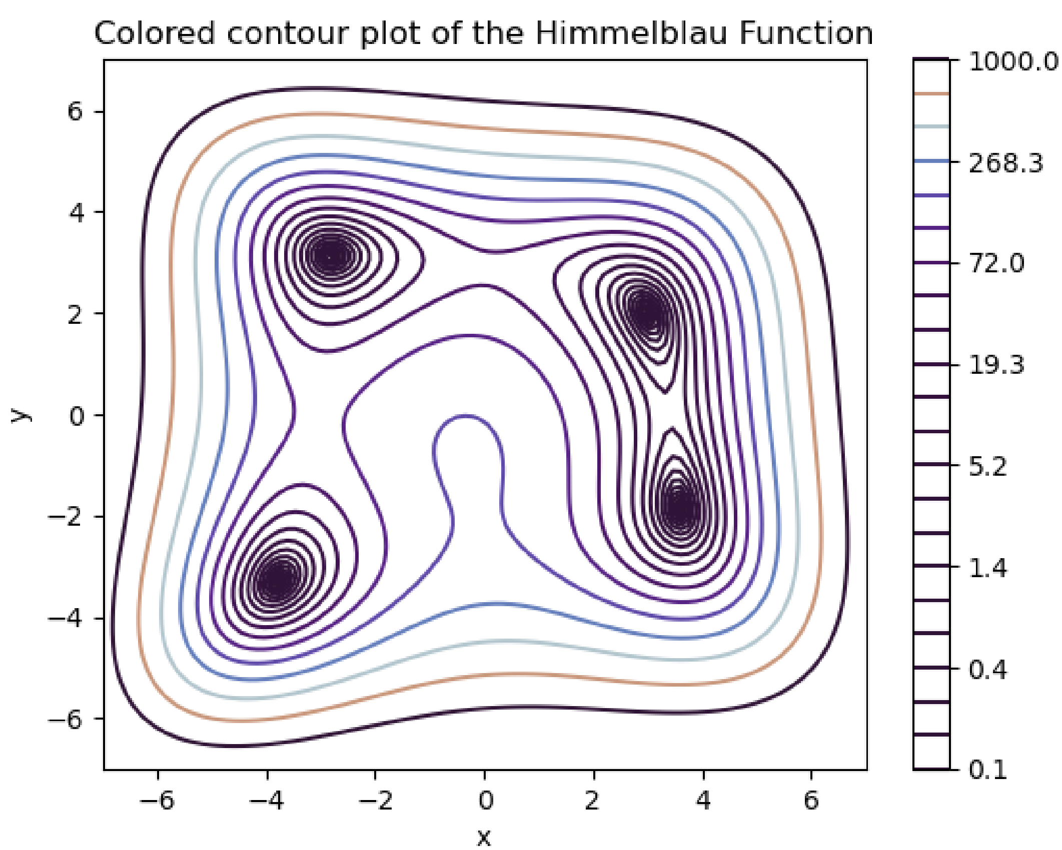
[ ]: <matplotlib.colorbar.Colorbar at 0x263adc729a0>



1.2.2 Part b

[ ]:
```python
# Setup a new figure
fig2, ax2 = plt.subplots() # Create a new figure and axes

# Define the new x, y, and X, and Y from the meshgrid.
x2 = np.linspace(-7, 7, 100)
y2 = np.linspace(-7, 7, 100)
X2, Y2 = np.meshgrid(x2, y2)

# Once you have defined those then you can create the contour plot with...
# ax2.contour(...) # Fill that in and remove the comment.
himmel_contour = ax2.contour(X2, Y2, f([X2,Y2]), levels=np.logspace(-1, 3, 22),
    cmap='twilight_shifted')

plt.xlabel('x')
plt.ylabel('y')
plt.title('Colored contour plot of the Himmelblau Function')

fig2.colorbar(himmel_contour)
```

[ ]:
```
<matplotlib.colorbar.Colorbar at 0x263adf4beb0>
```



1.2.3 Part c

Based on the plot again, we can see 4 approximate locations of minima.

```
[ ]: # Define the 4 initial guesses. Uncomment and add to the code here.
min_1 = [3, 2]
min_2 = [-2.8, 3.2]
min_3 = [-4, -3]
min_4 = [3.8, -2]

coords1 = scipy.optimize.fmin(f, min_1)
coords2 = scipy.optimize.fmin(f, min_2)
coords3 = scipy.optimize.fmin(f, min_3)
coords4 = scipy.optimize.fmin(f, min_4)

print(coords1,coords2,coords3,coords4)
```

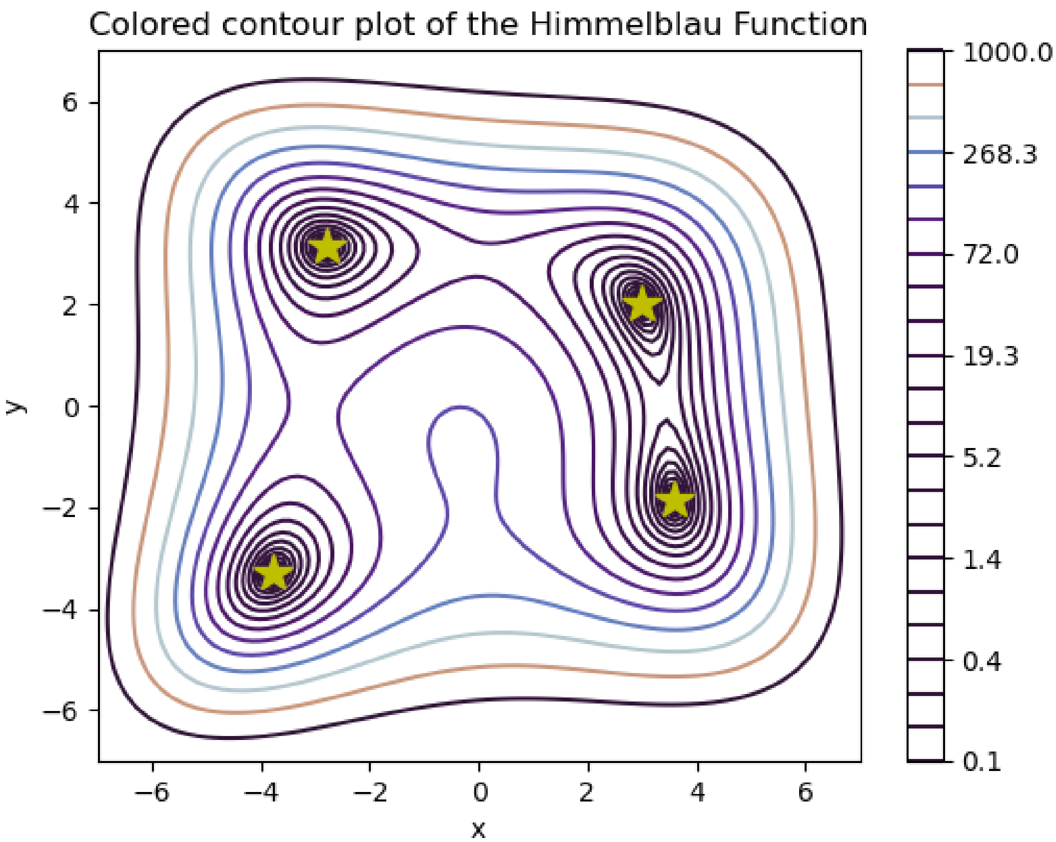
```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 23
         Function evaluations: 47
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 26
         Function evaluations: 51
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 28
         Function evaluations: 56
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 33
         Function evaluations: 64
[3. 2.] [-2.80511139  3.13133352] [-3.77932175 -3.28322441] [ 3.58444931
-1.84809728]
```

Once we have found the minima, we can plot them.

```
[ ]: # ax2.plot(...)
x_coords = [coords1[0],coords2[0],coords3[0],coords4[0]]
y_coords = [coords1[1],coords2[1],coords3[1],coords4[1]]
print(x_coords,y_coords)
ax2.plot(x_coords, y_coords, 'y*', markersize=15)
# Then we need to type "fig2" for it to show up again
fig2
```

[3.0, -2.805111393678241, -3.7793217548461713, 3.5844493143018497] [2.0, 3.1313335182260746, -3.2832244101277404, -1.848097282289666]

[ ]:

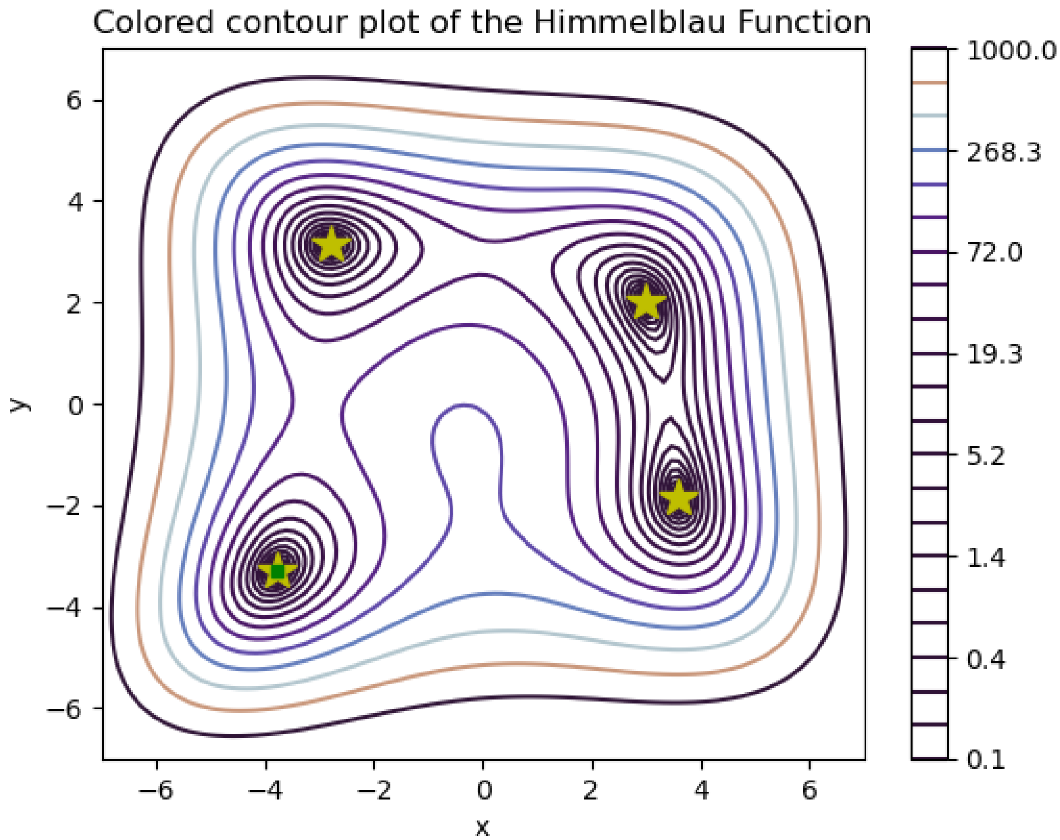


1.2.4 Part d

[ ]:

```
ax2.plot(p[0], p[1], 'gs', markersize=5)


fig2
```

[ ]:



1.3 Problem 2

1.3.1 Part a

[ ]:
```
import time # Import the timing algorithm,
            # we'll see how this works in Week 5.
start0 = time.time()
tol = 1e-9 # Set the tolerance - note that it changes
# Now run gradient descent!

p0 = [2, 3]
phi = lambda t: p0 - t*grad
f_of_phi = lambda t: f(phi(t))
for k in range(2000):
    grad = gradf(p0)
    tmin = scipy.optimize.fminbound(f_of_phi, 0, 1)

    if np.linalg.norm(grad)<tol:
        iter_num_fminbound = k
        print('parameters are', p0,'.')
```

```
        print('in coordinate form (x, y, z), the parameters are (', p0[0], ',␣
↪', p0[1], ', ', f(p0),').')
        print('loop has performed', iter_num_fminbound, 'iterations.')
        break

    p0 = phi(tmin)

stop0 = time.time()
print(stop0-start0, 's')
```

```
parameters are [3. 2.] .
in coordinate form (x, y, z), the parameters are ( 3.00000000000219 ,
1.999999999995336 ,  3.429519142906794e-22 ).
loop has performed 16 iterations.
0.006499528884887695 s
```

1.3.2 Part b-d

[ ]:

```
start001 = time.time()
tstep = 0.01
p1 = [2, 3]
for k in range(8000):
    grad = gradf(p1)
    p1 = p1 - tstep * grad

    if np.linalg.norm(grad)<tol:
        iter_num_001 = k
        print('parameters are', p1,'.')
        print('in coordinate form (x, y, z), the parameters are (', p1[0], ',␣
↪', p1[1], ', ', f(p1),').')
        print('loop has performed', iter_num_001, 'iterations.')
        break

    p001 = p1
print(p001)
stop001 = time.time()
print(stop001-start001,'s')


start001 = time.time()
tstep = 0.02
p1 = [2, 3]
for k in range(8000):
    grad = gradf(p1)
    p1 = p1 - tstep * grad

    if np.linalg.norm(grad)<tol:
```

```
        iter_num_001 = k
        print('parameters are', p1,'.')
        print('in coordinate form (x, y, z), the parameters are (', p1[0], ',␣
 ↪', p1[1], ', ', f(p1),').')
        print('loop has performed', iter_num_001, 'iterations.')
        break

    p001 = p1
print(p001)
stop001 = time.time()
print(stop001-start001,'s')


start001 = time.time()
tstep = 0.025
p1 = [2, 3]
for k in range(8000):
    grad = gradf(p1)
    p1 = p1 - tstep * grad

    if np.linalg.norm(grad)<tol:
        iter_num_001 = k
        print('parameters are', p1,'.')
        print('in coordinate form (x, y, z), the parameters are (', p1[0], ',␣
 ↪', p1[1], ', ', f(p1),').')
        print('loop has performed', iter_num_001, 'iterations.')
        break

    p001 = p1
print(p001)
stop001 = time.time()
print(stop001-start001,'s')
```

```
parameters are [3. 2.] .
in coordinate form (x, y, z), the parameters are ( 2.9999999999917444 ,
2.000000000019931 ,  5.984162345392522e-21 ).
loop has performed 81 iterations.
[3. 2.]
0.0025076866149902344 s
parameters are [3. 2.] .
in coordinate form (x, y, z), the parameters are ( 3.0000000000048472 ,
2.0000000000020077 ,  1.1324335078936936e-21 ).
loop has performed 55 iterations.
[3. 2.]
0.0019991397857666016 s
[3.23482283 2.12502178]
0.14100909233093262 s
```

1.3.3 Part e - the results

	Number Iterations	Time	Converged (Yes/No)
tstep = 0.01	81	0.002508	Yes
tstep = 0.02	55	0.001999	Yes
tstep = 0.025	8000	0.141009	No
`fminbound`	16	0.006499	Yes

1.3.4 Part f - discussion

- The Gradient Descent method did not always converge. From the example problems, it only converged for the fminbound method and with t-step sizes of 0.01 and 0.02. It did not converge with a step size of 0.025. I assume the answer did not converge for tstep = 0.025 because of the step size being too large for the tolerance input. In other words, I imagine the step size was too large and the algorithm would jump near the minimum multiple times, just outside of the tolerance range.
- From my work, it seems the Gradient Descent method with a fixed step size of 0.02 was fastest (at less than 2 ms).
- The fminbound Gradient Descent method converged the fastest by far with only 16 iterations.
- My answers to the above two question parts are not the same algorithm method. I assume the reason behind is is that, although the fminbound method is far more accurate, the process behind the scipy function we use takes longer to compute than simply multiplying a number by 0.02; in brief, quality over quantity $\implies$ more time and accuracy over less time.