STOR 235 — Lab 6

Instructions

There are two parts to this lab:

1. Calculus with Python;
2. Gradient descent.

Each part will be composed of a number of tasks for you to complete. The tasks are labeled in bold as **Problem 1**, **Problem 2**, and so on. Please make sure that you do not forget to complete any of the problems. Each problem will be worth 4 points.

There is a single code block in under this introductory material that imports various things you will use later. Please run it.

Important Information About Submitting Your Assignment

Please upload your assignment as a PDF file, and make sure to select to all relevant pages for each part. Make sure that your PDF files contains all of your work, and that nothing has been cut off at the end. **You will not receive credit for solutions that are not in the PDF.**

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
import sympy as sym
import scipy as sp
```

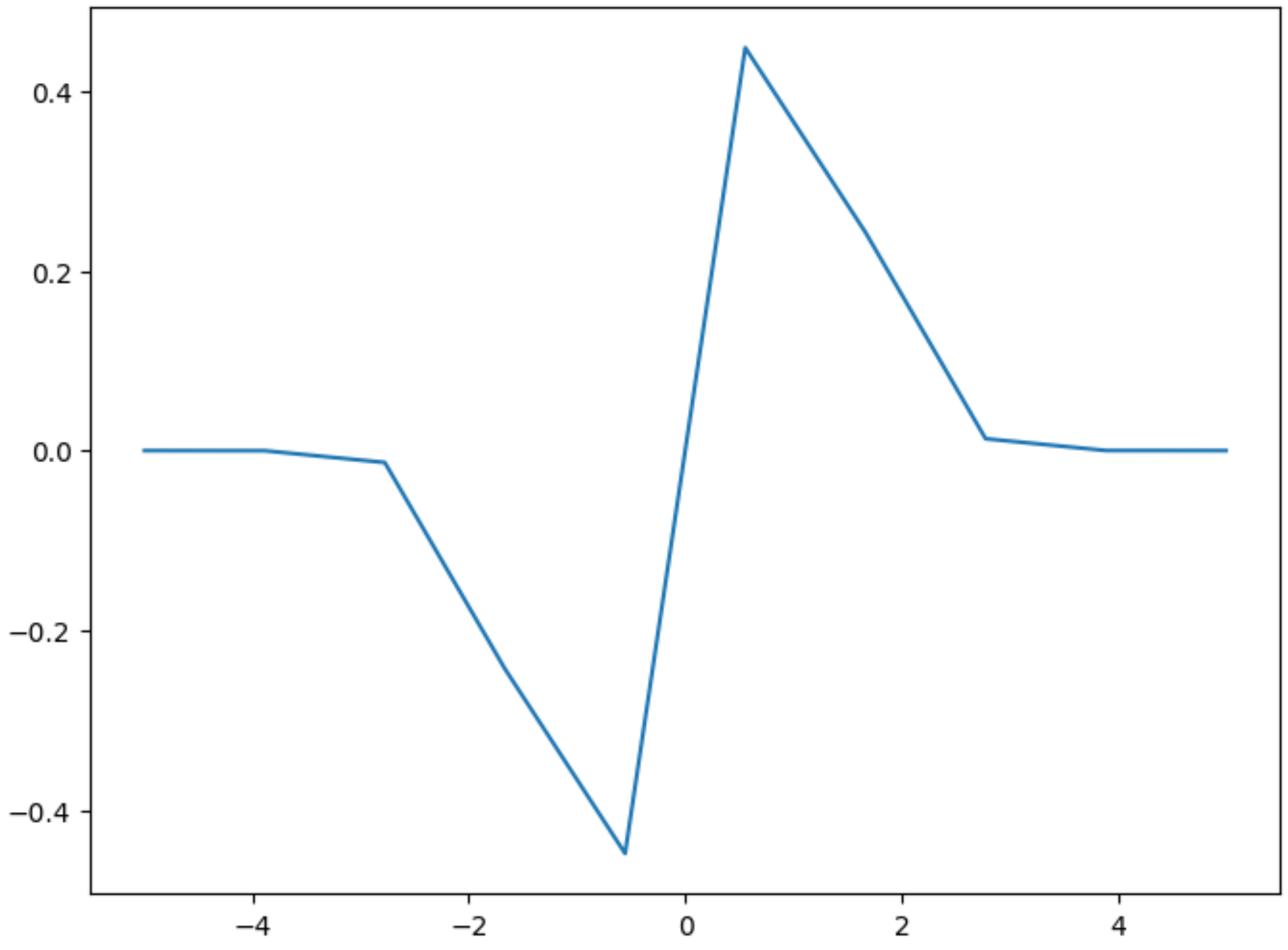
Part 1: Calculus

This part discuss how to perform some basic tasks, like plotting and computing derivatives, in Python. (I suggest using these commands to check your homework, if you are ever in doubt about something.)

The following code demonstates how to plot functions of a single variable. The `linspace` command creates a grid of values evenly spaced between the first and second arguments. The third argument gives the number of values in the grid. Note that the second graph is much clearer, since the more evenly spaced grid better captures the curvature of the function.

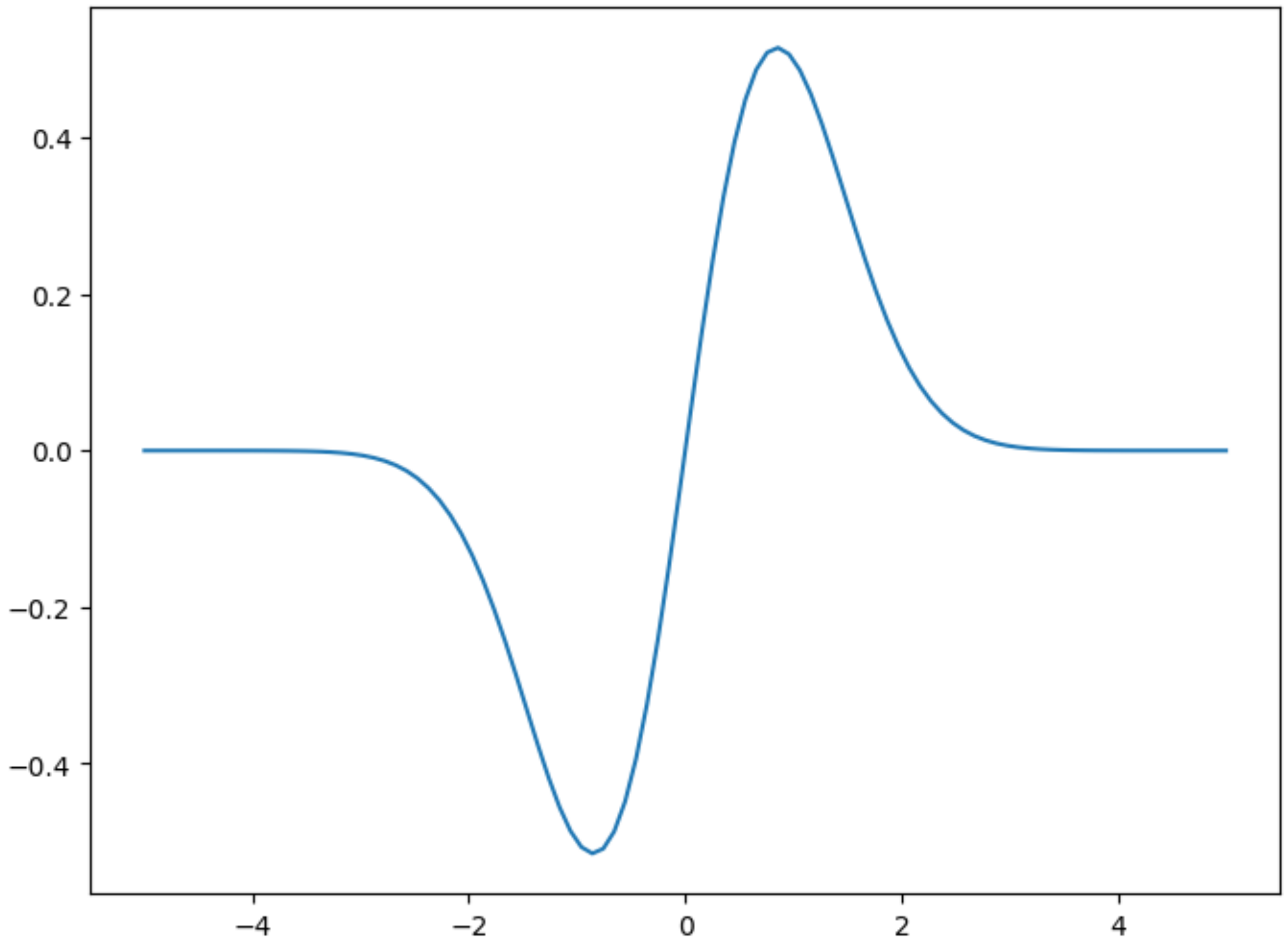
```
# Generate x values
x = np.linspace(-5, 5, 10)

# Calculate y values using the function f(x) = x * 2^(-x^2)
y = x*(2**(-x**2))

# Create the plot
plt.figure(figsize=(8, 6))
plt.plot(x, y)
```

```
[<matplotlib.lines.Line2D at 0x7af16f329310>]
```



```
# Generate x values
x1 = np.linspace(-5, 5, 100)

# Calculate y values
y1 = x1*(2**(-x1**2))

# Create the plot
plt.figure(figsize=(8, 6))
plt.plot(x1, y1)
```

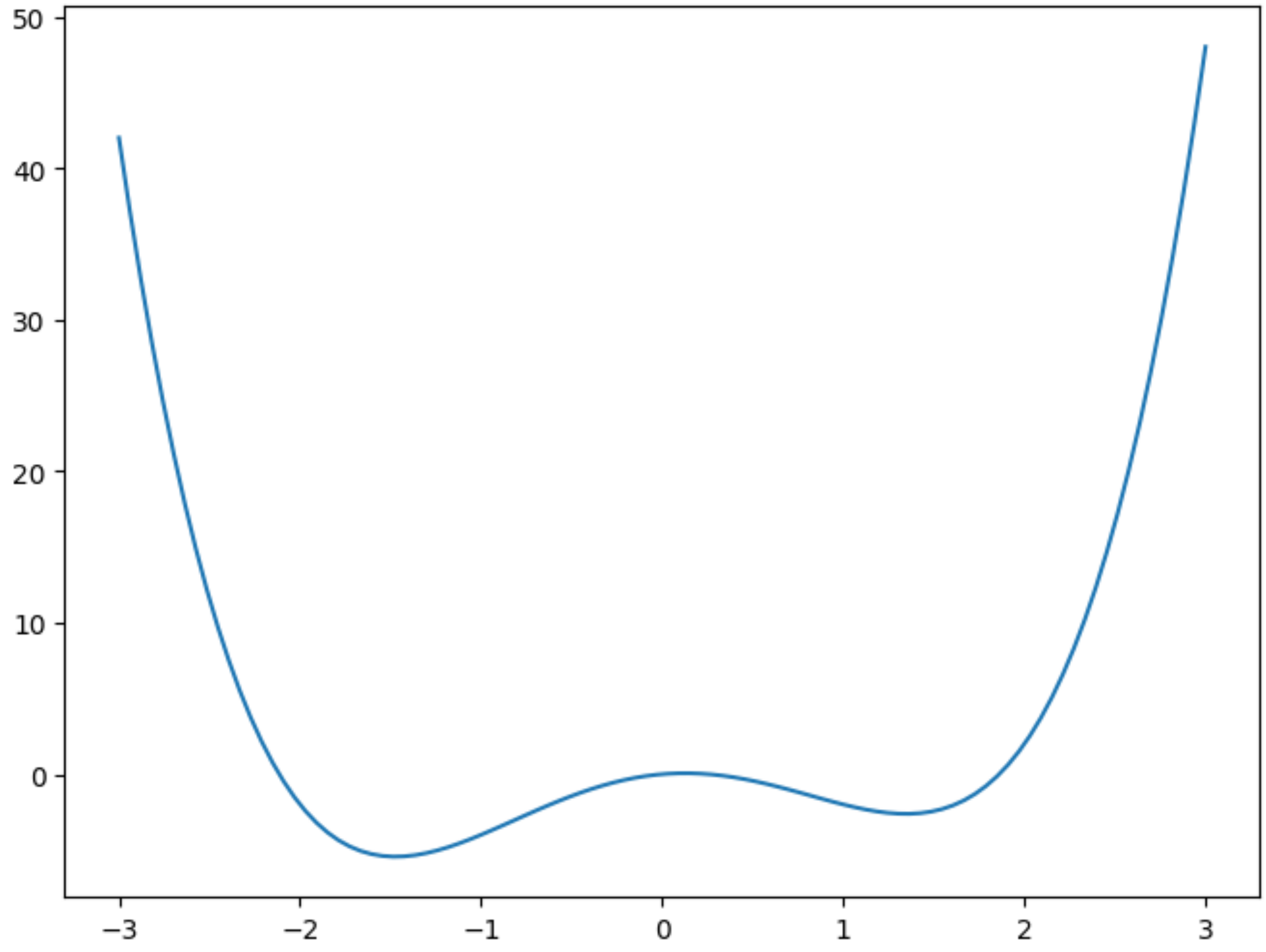
```
[<matplotlib.lines.Line2D at 0x7af16ee67090>]
```



Problem 1. Create a plot of the function `x^4 - 4x^2 + x` for x values in the interval `[-3,3]`. (Use enough grid points to make the plot look reasonably smooth.)

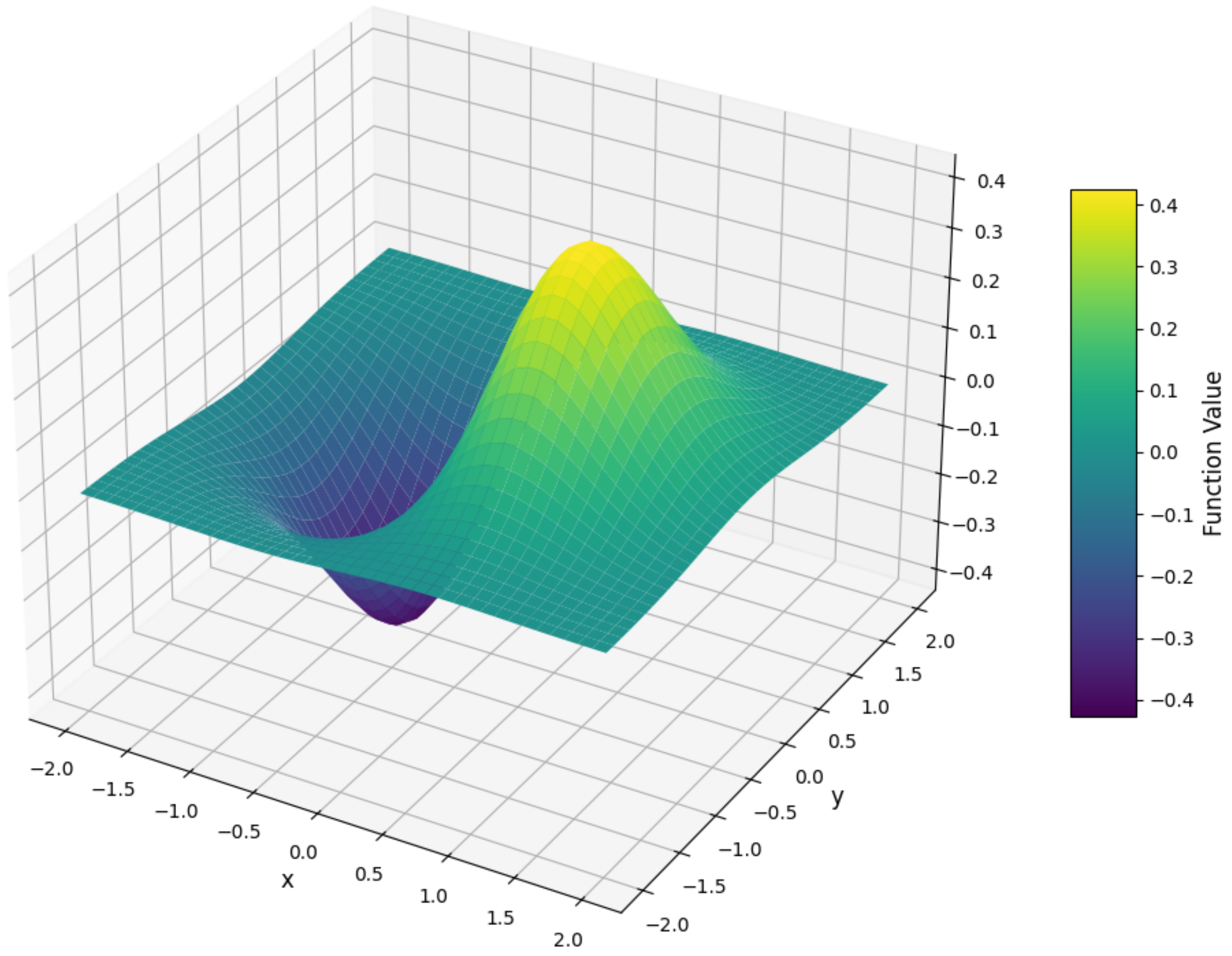
```
x1 = np.linspace(-3, 3, 100)


y1 = x1**4 - 4*x1**2 + x1

plt.figure(figsize=(8, 6))
plt.plot(x1, y1)
```

```
[<matplotlib.lines.Line2D at 0x7af16ee66e50>]
```



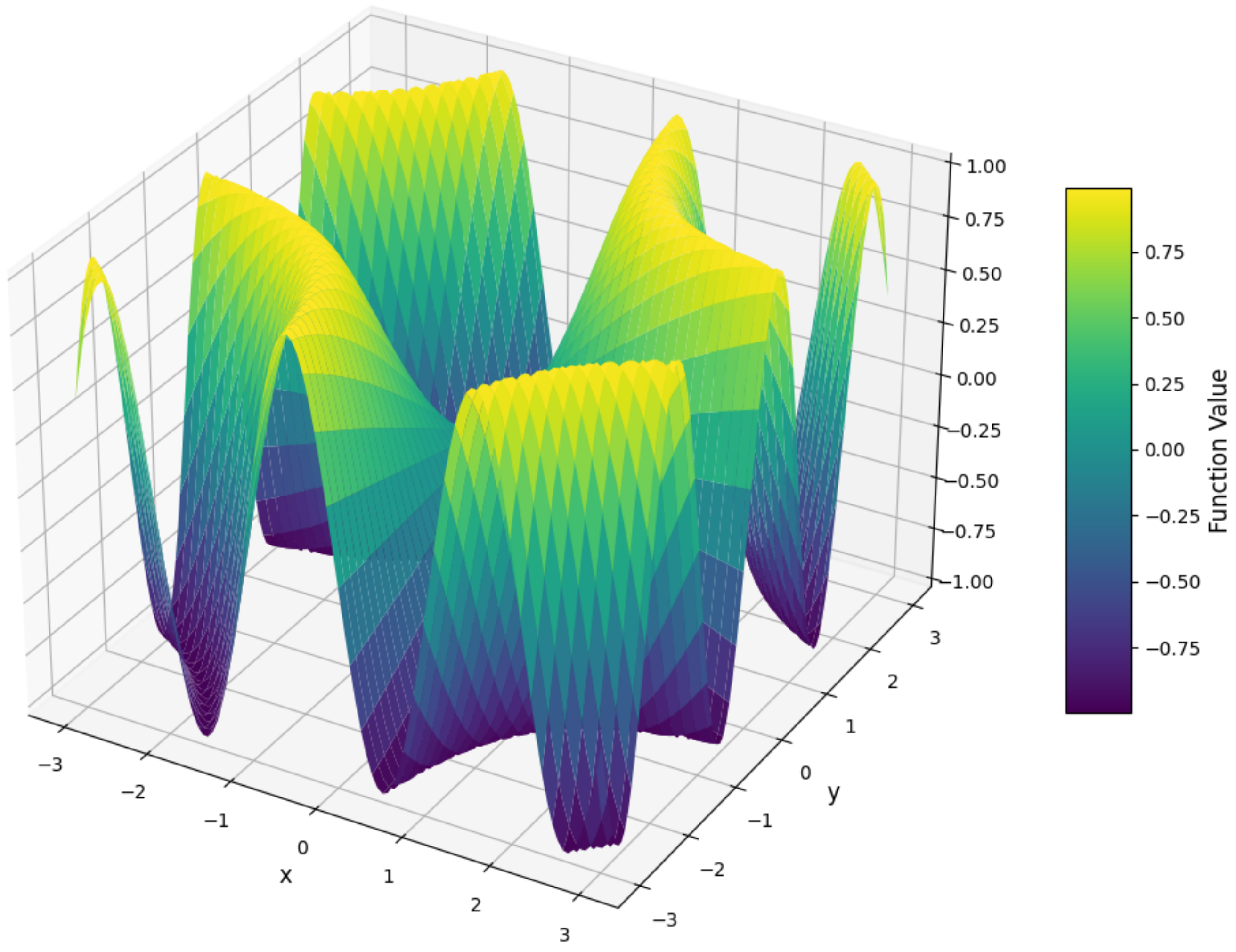
The following code demonstrates how to plot functions of two variables in three dimensions. The `meshgrid` command creates a two-dimensional grid. The function is then evluated at the grid points and plotted. I also included additional code for a colorbar that indicates the value of the function for each color in the plot.

```
def mv_function(x, y):
    return x*np.exp(-x**2 - y**2)


x = np.linspace(-2, 2, 40)
y = np.linspace(-2, 2, 40)

X, Y = np.meshgrid(x, y)
Z = my_function(X, Y)

fig = plt.figure(figsize=(10, 8))
ax = plt.axes(projection='3d')

surface = ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_xlabel('x', fontsize=12)
ax.set_ylabel('y', fontsize=12)



cbar = fig.colorbar(surface, ax=ax, shrink=0.5, aspect=8)
cbar.set_label('Function Value', fontsize=12)
plt.tight_layout()


plt.show()
```


x
y
Function Value
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4

Problem 2. Plot the function `sin(xy)` with `x` and `y` each in the interval `[-3,3]`. Try to get a reasonably smooth plot. (You can increase the resolution in the `linspace` command if the plot looks too blocky.)

```
def my_function(x, y):
    return np.sin(x*y)


x = np.linspace(-3, 3, 100)
y = np.linspace(-3, 3, 100)

X, Y = np.meshgrid(x, y)
Z = my_function(X, Y)

fig = plt.figure(figsize=(10, 8))
ax = plt.axes(projection='3d')

surface = ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_xlabel('x', fontsize=12)
ax.set_ylabel('y', fontsize=12)



cbar = fig.colorbar(surface, ax=ax, shrink=0.5, aspect=8)
cbar.set_label('Function Value', fontsize=12)
plt.tight_layout()


plt.show()
```



x
y
Function Value
−3
−2
−1
0
1
2
3
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

The following code demonstrates how to compute derivatives.

```
x, y, z = sym.symbols('x y z') #you only need to declare which symbols are variab
```

```
my_expression = x**3 * y + y**3 + z
```

```
derivative1_x = sym.diff(my_expression, x)
print(derivative1_x)
```

```
3*x**2*y
```

You can use dictionaries to evaluate the result at specific points.

```
point = {x: 1, y: 1, z: 1}
value = derivative1_x.subs(point)
print(value)
```

```
3
```

You can also take higher derivatives.

```
derivative2_x = sym.diff(my_expression, x, 2)
print(derivative2_x)
```

```
6*x*y
```

Problem 3. Write code that prints the answer to the first problem on homework 9 by having Python perform the required partial derivatives. (That is, use the `sym.diff` function to find the answer, don't just print the answer you found by hand!) Use `sym.exp` for the exponential function.

We can also do integrals with Python, both definite and indefinite.

```
x, y = sym.symbols('x y')


f = sym.exp(x**2 + y)

partial_fx = sym.diff(f, x)
print(partial_fx)


partial_fy = sym.diff(f, y)
print(partial_fy)
```

```
2*x*exp(x**2 + y)
exp(x**2 + y)
```

```
exp1 = sym.cos(x)
integral1 = sym.integrate(sym.cos(x), x)
print(integral1)
```

```
sin(x)
```

The following code computes the definite integral of cosine between 0 and 1.

```
integral2 = sym.integrate(sym.cos(x), (x, 0, 1))
print(integral2)
```

```
sin(1)
```

We can evaluate this answer using `numpy`.

```
print(np.sin(1))
```

```
0.8414709848078965
```

We can also directly do numerical integration using `SciPy`. This additionally provides an error estimate for the returned value.

```
def f(x):
    return np.cos(x)

result_quad, error_quad = sp.integrate.quad(f, 0, 1)
print(f"SciPy quad result: {result_quad:.10f} (error: {error_quad:.10e})")
```

```
SciPy quad result: 0.8414709848 (error: 9.3422046189e-15)
```

Problem 4. Use SciPy to find a numerical value for the integral of `exp(-x^2)` from `x=0` to `x=4`.

```
def f(x):
    return np.exp(-x**2)

result_quad, error_quad = sp.integrate.quad(f, 0, 4)
print(f"SciPy quad result: {result_quad:.10f} (error: {error_quad:.10e})")
```

```
SciPy quad result: 0.8862269118 (error: 1.3180149476e-08)
```

Finally, we can use Python to compute Taylor series. The last two arguments of the following command give the point to expand around, and how many terms of the series to compute. (More specifically, if you put in the value `n` for the numer of terms, it will give the terms in the series up to and incuding the value for the coefficient of `x^(n-1)`.)

```
series = sym.series(sym.exp(x), x, 0, 4)
print(series)
```

```
1 + x + x**2/2 + x**3/6 + O(x**4)
```

```
series2 = sym.series(sym.exp(x), x, 2, 5)
print(series2)
```

```
exp(2) + (x - 2)*exp(2) + (x - 2)**2*exp(2)/2 + (x - 2)**3*exp(2)/6 + (x - 2)
```

Problem 5. Use Python to compute the first 6 terms of the Taylor series for `(1+x^2)exp(-x^2)` at `x=0`.

```
series3 = sym.series((1+x**2)*sym.exp(-x**2), x, 0, 6)
print(series3)
```

```
1 - x**4/2 + O(x**6)
```

Part 2: Gradient Descent

The following problems ask to you implement the gradient descent algorithm in one dimension and explore some of its properties. See the notes for details of this algorithm.

The first problem requires you to provide a non-code answer, in addition to writing code. This should be written in after the bold **Answer** prompt.

Problem 1. Write a function with two arguments, a learning rate parameter and an initial guess. The function should perform 10 iterations of gradient descent with `f(x) = x^2/2` and the given initial guess, then return the resulting value. Print the result of the function applied with initial value 10 and the following learning rates: .25, .5, .75, 1, 1.25, 1.5, 1.75, 2, 3. The gradient descent calculations/iterations should be written out explicitly; don't just call a library function.

Which learning rate gives the most accurate value for the minimum? Which gives the last accurate value? (You may wish to look at the discussion of the rate of convergence for gradient descent for this function in the notes. The answers can also be deduced from the analysis there.)

Answer:

The learning rate that gives the most accurate value for the minimum is 0.25 because after the given steps, it is the closest to the true minimum. On the other hand, the least accurate result comes from the largest learning rate tested which was 3.0 as this is where the value diverged furthest from the minimum.

This behavior is similar to what we saw in the notes as the gradient descent with small learning rates converges slowly but conciself and I guess effectively. As the learning rate increases, the steps become too large and it shows that it fails too converge for large learning rates.

```
def gd(learning_rate, initial_guess):
    x = initial_guess
    derivative_x = x
    for i in range(10):
        x = x - learning_rate * derivative_x
    return x

initial_guess = 10
learning_rates = [0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75, 2, 3]

for lr in learning_rates:
    result = gd(lr, initial_guess)
    print(result)
```

```
-15.0
-40.0
-65.0
-90
-115.0
-140.0
-165.0
-190
-290
```

Problem 2. Consider the function `f(x)= x^4 - 4x^2 + x`. Use gradient descent to find the value of `x` that achieves the global minimum, and the value of `f(x)` at that point. Print both values. To get full points, you must get the first three digits of each number right. (For example, if the true minimizer is 3.14.5..., then your answer must begin 3.14... to get full credit.)

The gradient descent calculations/iterations should be written out explicitly; don't just call a library function.

Hint 1. This function is not convex, so it is possible for gradient descent to converge to the wrong local minimum. Use the graph of this function that you produced above to choose a good initial value to prevent this. You can also use this graph to estimate the location and value of the global minimum to make sure your answer makes sense.

Hint 2. Choosing a good learning rate is crucial here. If it is too large, it won't converge (as we saw in the notes). You will need values a bit smaller than the previous example. Try a range of values. If you try many small learning rate parameters and they all give the same result, this should increase your confidence in your answer. Also, try increasing the number of iterations.

```
def gd2(learning_rate, initial_guess, iterations):
    x = initial_guess
    for i in range(iterations):
        derivative_x = 4*x**3 - 8*x + 1
        x = x - learning_rate * derivative_x
        return x, x**4 - 4*x**2 + x

initial_guess = -1.5
learning_rates = [0.01, 0.02, 0.04, 0.05, 0.1, 0.2, 0.25, 0.5]

for lr in learning_rates:
    result = gd2(lr, initial_guess, 100)
    print(result)
```

```
(-1.495, -5.439763249375)
(-1.49, -5.44155599)
(-1.48, -5.44374784)
(-1.475, -5.444155859375)
(-1.45, -5.4394937500000005)
(-1.4, -5.3984)
(-1.375, -5.363037109375)
(-1.25, -5.05859375)
```

```
gd2(0.05, -1.5, 100)
```

```
(-1.475, -5.444155859375)
```