Implementing the Gradient Descent Algorithm from First Principles

Charlie Bailey (peba2926)

"What I cannot create, I do not understand.."

—**Richard Feynman**

"The back-propagation algorithm is very simple. To compute the gradient of some scalar z with respect to one of its ancestors x in the graph, we begin by observing that the gradient with respect to z is given by $\frac{dz}{dz} = 1$. We can then compute the gradient with respect to each parent of z in the graph by multiplying the current gradient by the Jacobian of the operation that produced z."

—**Deep Learning** by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

Introduction

While the paragraph above may seem "very simple" to the leading minds in field of Artificial Intelligence, for those new to the field, even the fundamental concept of a gradient can be confusing. What exactly is a `gradient`? How do we actually go about finding one? And once we have this gradient, how is it actually used?

Answering these questions will take us to the heart of modern Artificial Intelligence, where methods like `gradient descent` power the training of everything from simple regression models to advanced neural networks. Many people—myself included—are introduced to gradient descent through code libraries and black-box optimization routines. Without a deeper understanding though, these training methods can seem like black magic rather than a systematic mathematical procedure.

The goal of this project is to demystify gradient descent by building it from the ground up—without relying on any pre-built functions or libraries. Guided by Callahmn and Hoffman's computation-based approach to understanding calculus we will start with the foundational calculus concepts underlying gradients and then, building from first principles, we will create a custom implementation of the gradient descent algorithm.

In [1]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from IPython.display import Image
```

Chapter Nine: Functions of Several Variables

Section 9.1: Graphs and Level Sets

Before we can understand what exactly a gradient is, we first need to understand the geometry of functions with two or more variables. In reality, AI models may have thousands, millions or even billions of variables (also known as parameters), so it is impossible to truly visualize what the gradient would "look like" in these high-dimensional spaces. Luckily for us though, the concept of a gradient in two dimensions extends well enough to these high-dimensional spaces to allow us to get a foot hold.

Exercise 1:

Graph the function $z = sin(x)sin(y)$ on the domain $0 \leq x \leq 2\pi, 0 \leq y \leq 2\pi$.

How many maximum points do you see?

How many minimum points?

How many saddles?

In [2]:
```
# generate points for plotting
two_pi = 2*np.pi
x = np.linspace(0, two_pi, 1000)
y = np.linspace(0, two_pi, 1000)

# create matrices for plotting
X, Y = np.meshgrid(x, y)

# calculate z-points
Z = np.sin(X) * np.sin(Y)
```

In [3]:
```
# generate plot
fig1 = go.Figure(data=[go.Surface(x=X, y=Y, z=Z,)])
fig1.update_layout(
    title='Graph of z = sin(x)sin(y)',
    scene=dict(
        xaxis_title='X axis',
        yaxis_title='Y axis',
        zaxis_title='Z axis',
        camera=dict(
            up=dict(x=0, y=0, z=1),
```

```
            center=dict(x=0, y=0, z=0.25),
            eye=dict(x=2, y=2, z=1.5)
        )
    ),
    width=800,
    height=800
)
fig1.show()
```

Note: if you are accessing this project in PDF format, this is a 3D interactive graph that unfortunately won't render to PDF. This project is best viewed in it's Jupyter Notebook format to get the full benefit of interacting with the graph.

We can "walk around" this graph by clicking and dragging to form different perspectives. We can also hover our cursor over various points in the graph to see the three coordinates at any location.

By moving our cursor to inspect the various `critical points`, we can answer the questions above:

Maximums: 2

	Point 1	Point 2
x	1.56	4.72
y	1.56	4.72
z	1	1

Minimums: 2

	Point 1	Point 2
x	4.72	1.56
y	1.56	4.72
z	-1	-1

Saddle Points: 1

	Point 1
x	3.14
y	3.14
z	9.89

It's worth pausing for a moment to realize that what we just did—specifically using hovering our cursor to find the lowest z values—is **exactly** how the gradient descent algorithm works!

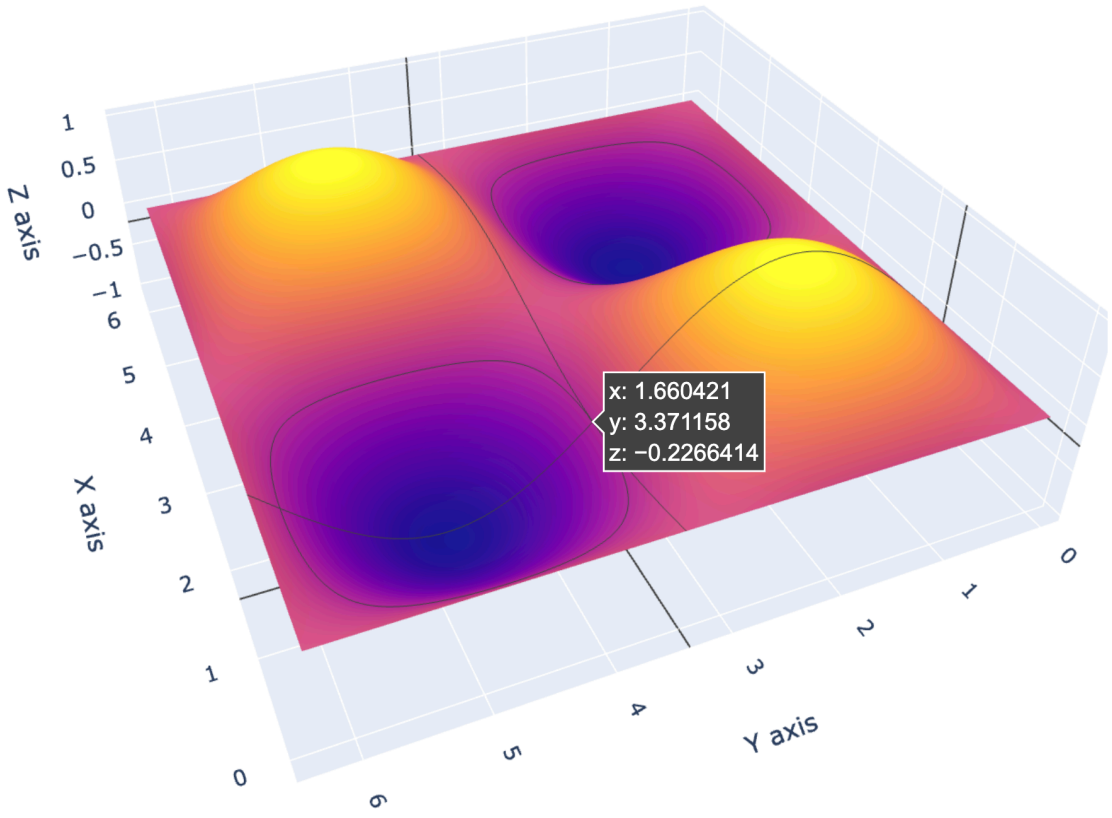
What we just did visually with a cursor is what the gradient descent algorithm needs to do to minimize an `objective function`.

Alright, so conceptually we know where we need to go. Bring the cursor to the lowest point on the graph. Simple enough. But how did we know exactly what to do? Even more so, how would we tell a computer to do what we just did? How do we explain to a computer the concept of "move to the lowest point" on a graph?

If you scroll over the graph again you will notice that there are several gray lines that appear as you move your cursor around. The horizontal lines are called the `contour` lines and the vertical lines are the `grid` lines.

In [4]:
```
Image('assets/contour_grid_lines.png', width=600)
```

Out[4]:



More formally, the contour of $z = f(x, y)$ is the set of points in the x, y plane where f has some fixed value:

$$f(x, y) = c$$

This fixed value c is called the `level`.

With these concepts in mind, we can use the metaphor of pressing a piece of paper "down" through the graph to better understand the contour lines. Imagine this piece of paper being connected to your cursor as you drag it down the graph toward the minimum. The paper stays flat and perpendicular to the z-axis as you move the cursor. At each step of the way, all of the points that the paper is currently touching have the same c value. And at each step of the way, this c value is smaller than the value at the step above. Eventually, we reach the bottom of the bowl and the paper is just barely touching the bottom of the graph at a single point.

More formally, we would say that this piece of paper—or *the plane at contour level* c—is `tangent` to the function $f(x, y)$ at points (a, b). This point $(a, b, f(a, b))$ is one of the `minimums` of the graph. Here we can recognize that a horizontal tangent to a function indicates that the instantaneous rate of change at that point is zero. Said another way, the `derivative` of our function at the lowest possible contour level still touching the graph is zero.

Now this is sounding a bit more like something we could program into a computer. If we were able to somehow follow these contour lines down—we would be able to navigate to the minimum.

Let's examine in a bit more detail what exactly is going on with these changing contour lines to figure out how we might tell the computer to "bring the cursor to the lowest point on the graph."

Section 9.2: Local Linearity

For this next part, we will be looking at the graph of $f(x, y) = x^3 - 4x - y^2$.

Let's take a look at what life is like down on the surface of our function. Using the "microscope" metaphor, we will zoom in on the point $(x, y) = (1.8, -1)$ to examine the `local linearity` of this function—which is the idea that the graph approaches a flat plane when viewed under intense magnification.

In [5]:
```
# generate points for a new graph
x_1 = np.linspace(-1.5, 4.5, 1000)
y_1 = np.linspace(-4, 2, 1000)

# create matrices for plotting
X_1, Y_1 = np.meshgrid(x_1, y_1)

# calculate z-points
Z_1 = X_1**3 - 4*X_1 - Y_1**2
```

In [6]:
```
# update the plot to show contour lines and new function
# generate plot
fig2 = go.Figure(data=[go.Surface(x=X_1, y=Y_1, z=Z_1)])
fig2.update_traces(
```

```
    contours=dict(
        x=dict(show=True, start=X_1.min(), end=X_1.max(), size=0.3, color='b
        y=dict(show=True, start=Y_1.min(), end=Y_1.max(), size=0.3, color='b
        z=dict(show=True, start=Z_1.min(), end=Z_1.max(), size=0.3, color='b
    )
)
fig2.update_layout(
    title='Graph of z = x^3 - 4x -y^2',
    scene=dict(
        xaxis_title='X axis',
        yaxis_title='Y axis',
        zaxis_title='Z axis',
        camera=dict(
            up=dict(x=0, y=0, z=1),
            center=dict(x=0, y=0, z=0.25),
            eye=dict(x=0, y=-2, z=1.75)
        )
    ),
    width=800,
    height=800
)
fig2.show()
```

Exercise 2:

Graph the function $z = x^3 - 4x - y^2$ on a domain centered at th point $(x, y) = (1.8, -1)$ and magnify until it looks like a plane.

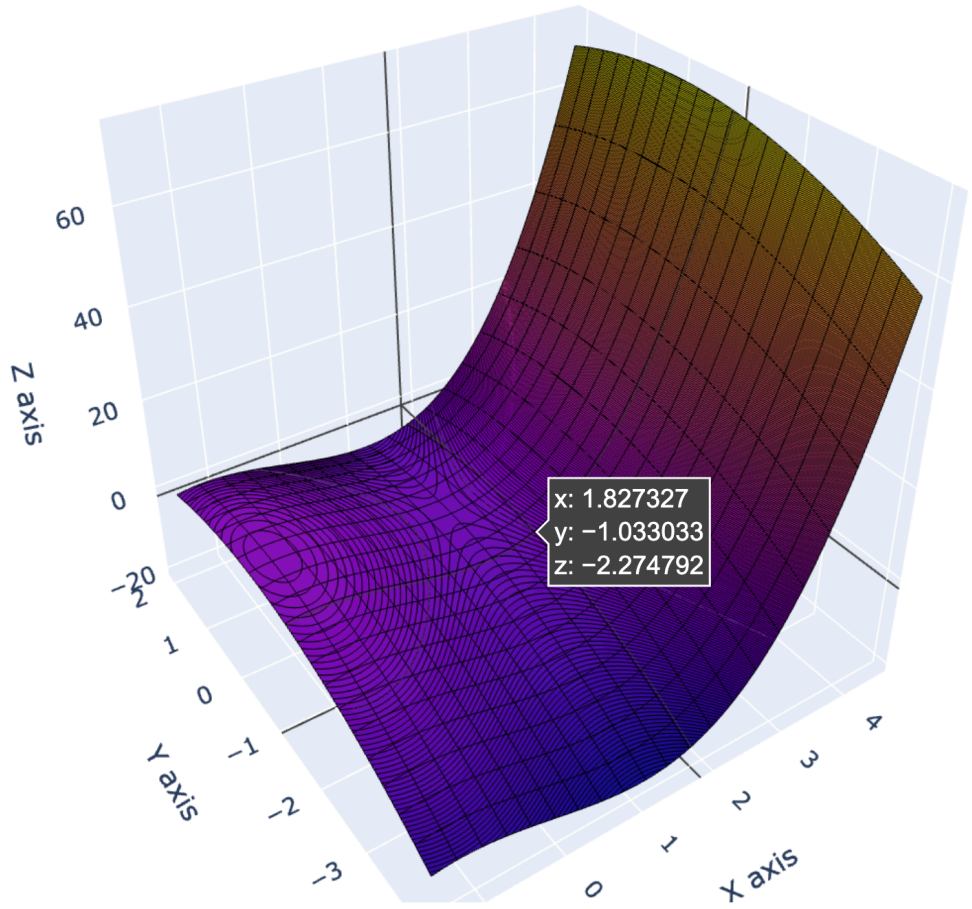
Use the following steps to estimate the rate of change $\Delta z/\Delta x$ at $(x, y) = (1.8, -1)$:

1. What is the horizontal spacing between the two contours closest to the point $(1.8, -1)$?
2. Find the z-levels z_1 and z_2 of those contours and then compute $\Delta z = z_2 - z_1$.
3. Finally, calculate the value $\Delta z/\Delta x$.

Finally, repeat these steps to find to estimate the rate of change $\Delta z/\Delta y$ at $(x, y) = (1.8, -1)$.

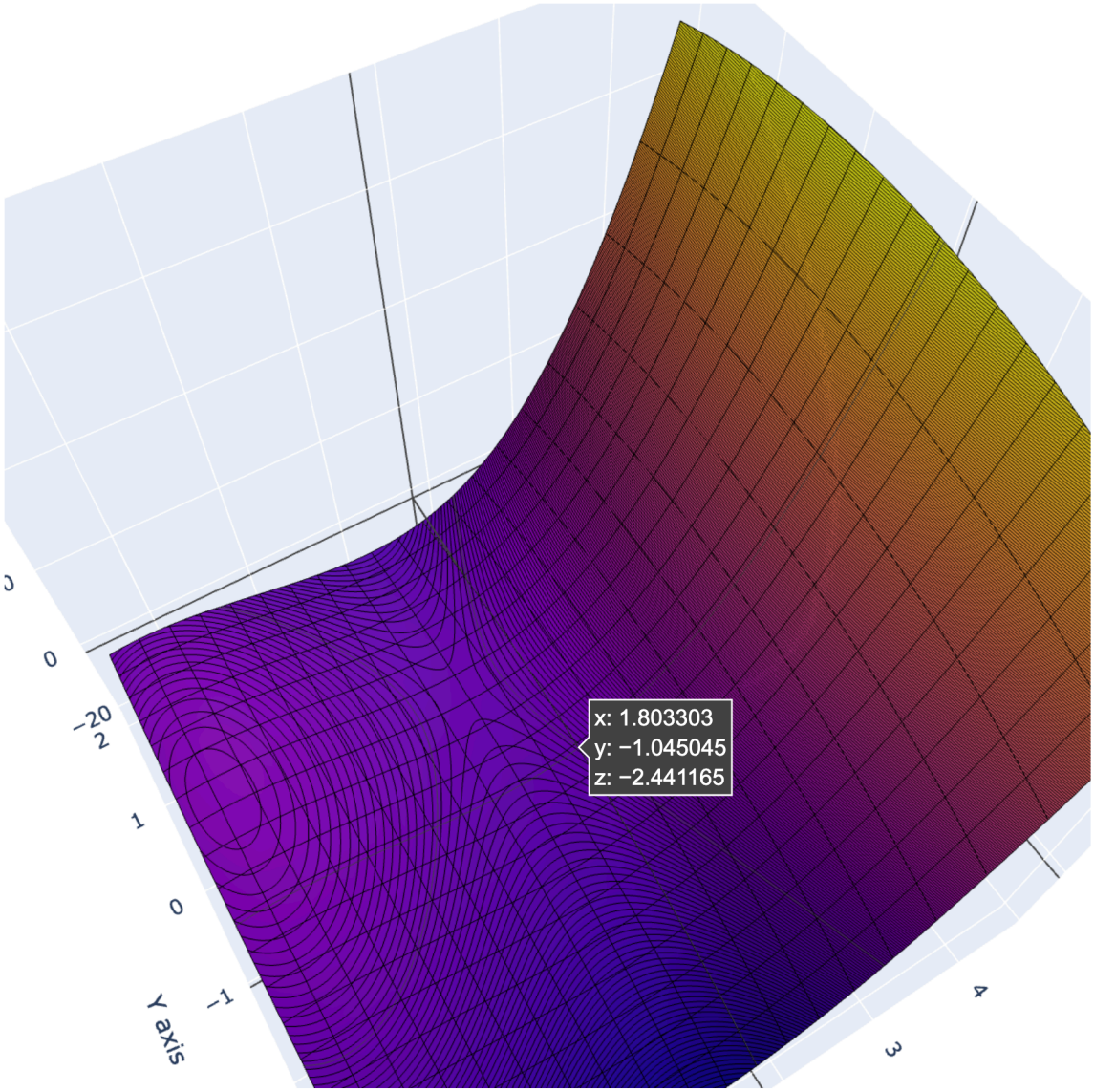
In [7]:
```
Image('assets/zoom_1.png', width=600)
```

Out[7]:



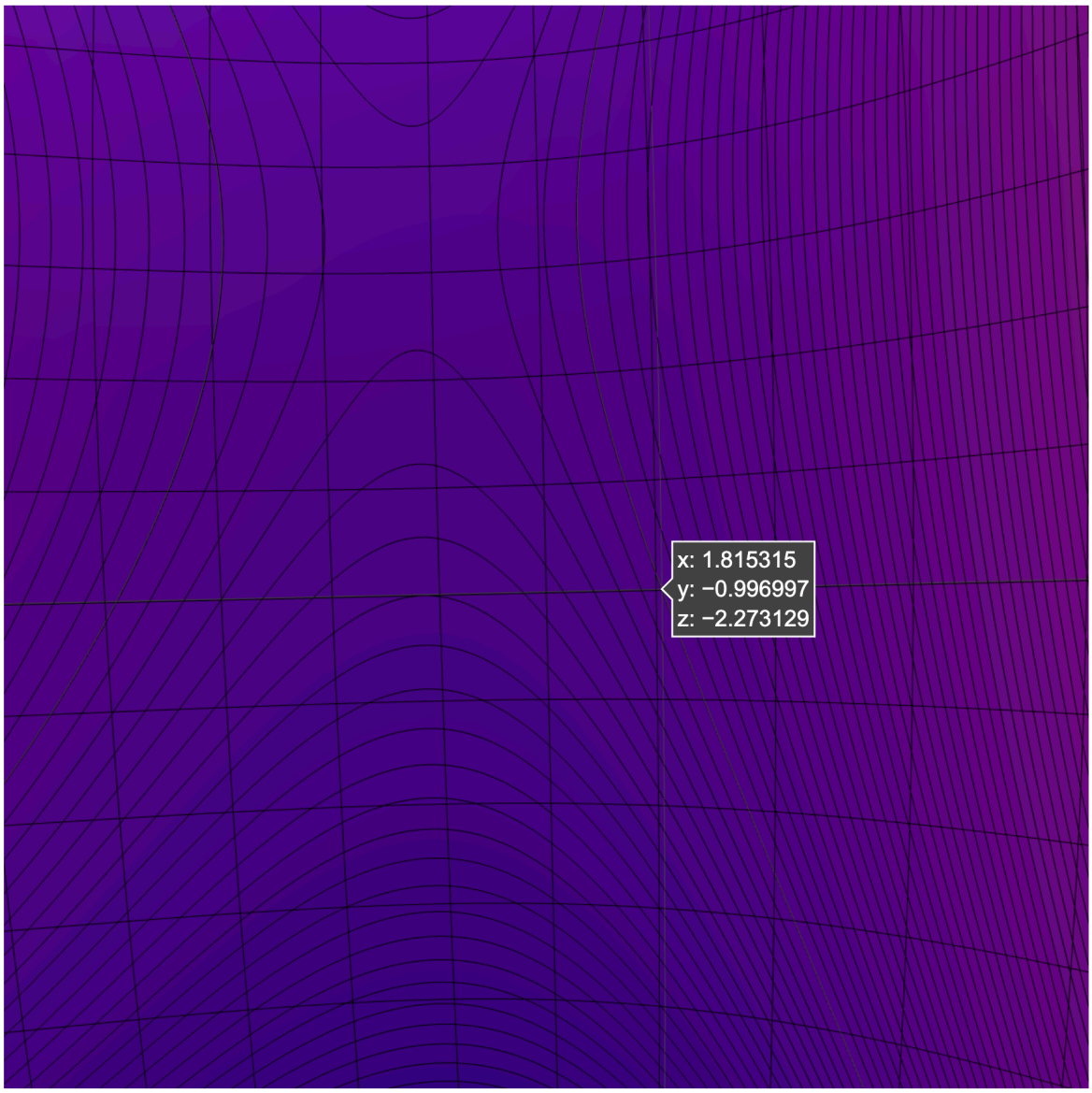
In [8]:
```
Image('assets/zoom_2.png', width=600)
```

Out[8]:



In [9]:
```
Image('assets/zoom_3.png', width=600)
```

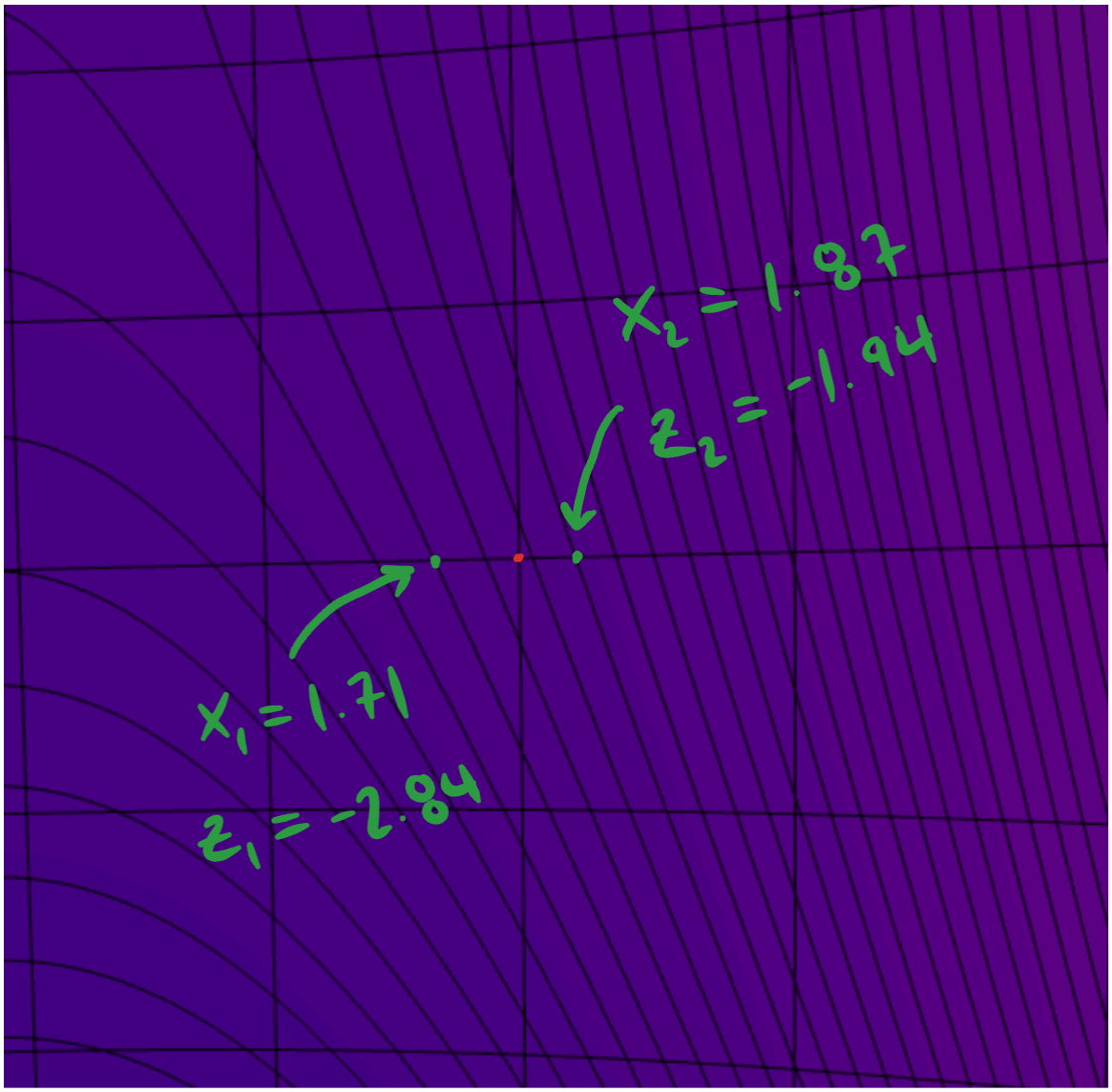
Out[9]:



Unfortunately, the Plotly zoom capabilities are not robust enough to get a perfect locally linear plane, but we can get pretty close around our desired (x, y) coordinate using some manual zooming on the image above.

In [10]:
```
Image('assets/zoom_x_work3.png', width=600)
```

Out[10]:

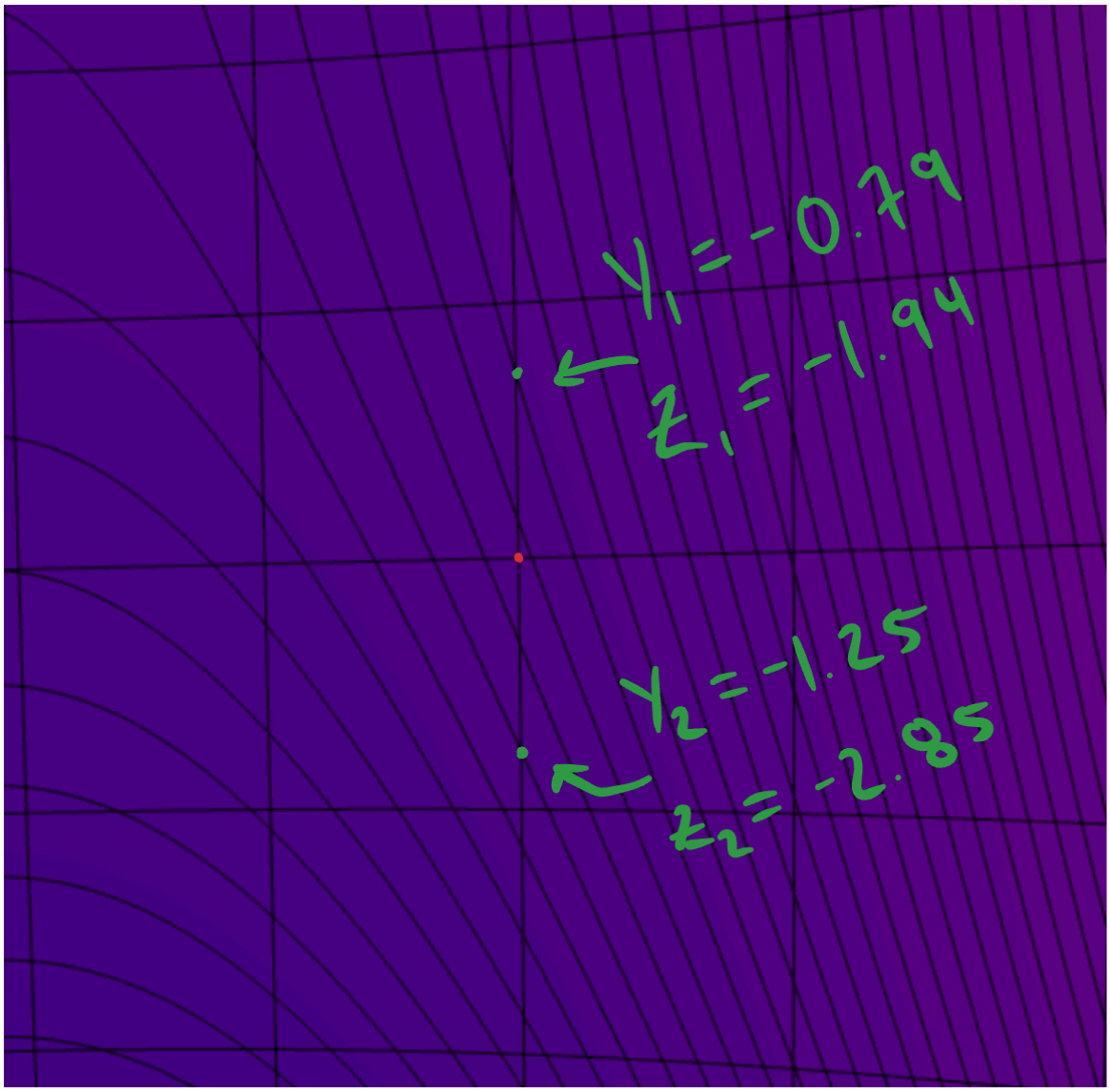


Alright, we are in flatland! The little red dot here is our cursor. Let's use the steps above to calculate $\Delta z / \Delta x$.

1. Calculate $x_2 - x_1$ which is $1.87 - 1.71 = 0.16$
2. Calculate $z_2 - z_1$ which is $-1.94 - (-2.84) = 0.9$
3. Therefore, $\Delta z / \Delta x$ is $0.9 / 0.16 = 5.625$

In [11]:
```
Image('assets/zoom_y_work4.png', width=600)
```

Out[11]:



We repeat the same steps above to calculate $\Delta z / \Delta y$

1. Calculate $y_2 - y_1$ which is $-1.25 - (-0.79) = -0.46$
2. Calculate $z_2 - z_1$ which is $-2.85 - (-1.94) = -0.91$
3. Therefore, $\Delta z / \Delta y$ is $-0.91 / (-0.46) \approx 1.98$

And with these two calculations, we can now introduce the concept of the `gradient`.

The gradient of a function $f = f(x, y)$ is the vector whose components are its partial rates of change:

$$grad f = \nabla f = (f_x, f_y) = (\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y})$$

This means that, based on our numerical approximation using the contours of the locally linear plane, the gradient of the function $z = f(x, y) = x^3 - 4x - y^2$ at the point $(x, y) = (1.8, -1)$ is the vector:

$$\nabla z = (5.625, 1.98)$$

Exercise 3: Coding the Gradient Descent Algorithm

We now have nearly all the pieces we need to be able to tell a computer how to find the minimum of a function. Let's pull in some of the numerical point derivative work we did earlier in the course to see if we can automate the gradient finding process we just went through by hand.

In [12]:
```
# just for reference: numerical point derivative of a single variable functi
def numerical_point_derivative(f, a, h=0.0001):
    return (f(a + h) - f(a - h)) / (2*h)
```

In [13]:
```
# extrapolation to a partial derivatives
def calc_partial_deriv_x(f, x, y, h=0.05):
    x_1 = x+h
    x_2 = x-h
    z_1 = f(x_1, y)
    z_2 = f(x_2, y)
    return ((z_2 - z_1) / (x_2 - x_1))

def calc_partial_deriv_y(f, x, y, h=0.05):
    y_1 = y+h
    y_2 = y-h
    z_1 = f(x, y_1)
    z_2 = f(x, y_2)
    return ((z_2 - z_1) / (y_2 - y_1))
```

In [14]:
```
# test on points used in hand-worked example
def func(x, y):
    return x**3 - 4*x - y**2

dz_x = calc_partial_deriv_x(func, x=1.8, y=-1)
dz_y = calc_partial_deriv_y(func, x=1.8, y=-1)
print(dz_x)
print(dz_y)
```

```
5.722499999999998
2.0
```

Awesome! Using similar h values to the ones provided by the contours in the locally linear plane, we are now able to programmatically get the partial derivatives of both x and y. As noted above, these are the components of our gradient vector ∇z. This is our compass telling us the direction of `steepest ascent` —that if we went ∇z_x units in the x-direction and then ∇z_y units in the y-direction, this would put us at an (x, y) point with the highest value of z.

So, we can logically deduce that if this gradient vector is pointing in the direction of steepest ascent, all we need to do to find our minimum is head in the opposite direction.

There's just one last piece of information we need to be able to implement our gradient descent algorithm. This is the idea of a `learning rate`. Since our gradient can be very steep in certain places, this means that taking a "full unit step" in terms of following the vector displacement may cause us to overshoot our minimum by taking too large of a step. Therefore, we update our location by a small proportion of the full displacement calculated by our gradient vector. For this implementation, we'll set this learning rate to be 0.1.

In [15]:
```
def calc_grad(f, x, y):
    # better approximation with smaller h value
    dz_x = calc_partial_deriv_x(f, x, y, h=0.0001)
    dz_y = calc_partial_deriv_y(f, x, y, h=0.0001)
    return (dz_x, dz_y)
```

In [16]:
```
def gradient_descent(f, x_init, y_init, alpha=0.1):
    # store path for plotting
    x_points = [x_init]
    y_points = [y_init]
    z_points = [f(x_init, y_init)]

    # initialize grad and displacements
    init_grad_z = calc_grad(f, x_init, y_init)
    step_x = x_init - (alpha*init_grad_z[0])
    step_y = y_init - (alpha*init_grad_z[1])
    delta_z = f(step_x, step_y)


    # iterate until we can't descend any further
    while (delta_z < z_points[-1]):
        # store previous step vals
        x_points.append(step_x)
        y_points.append(step_y)
        z_points.append(delta_z)

        # calculate new grad and next steps
        new_grad_z = calc_grad(f, step_x, step_y)
        next_step_x = step_x - (alpha*new_grad_z[0])
        next_step_y = step_y - (alpha*new_grad_z[1])

        # update steps
        delta_z = f(next_step_x, next_step_y)
        step_x = next_step_x
        step_y = next_step_y

    # store final step vals
    x_points.append(step_x)
    y_points.append(step_y)
    z_points.append(delta_z)

    return (x_points, y_points, z_points)
```

In [17]:
```
# test on original function at x = 1.66, y = 3.37
x_path, y_path, z_path = gradient_descent(lambda x, y: np.sin(x) * np.sin(y)
```

```
print('x::: ', x_path[-1])
print('y::: ', y_path[-1])
print('z::: ', z_path[-1])
print('steps:::', len(z_path))
```

```
x:::  1.5707963291341602
y:::  4.712388955843883
z:::  -0.9999999999999997
steps::: 172
```

HUZZAH!!!

We can compare these values against the values we found using our manual inspection above and see that we have indeed found one of the minimum points of the function!

Minimums:

	Point 1	Point 2	GD
x	4.72	1.56	**1.57**
y	1.56	4.72	**4.71**
z	-1	-1	**-0.99999**

Let's plot the path our computed cursor took.

In [18]:
```
# generate points for plotting
two_pi = 2*np.pi
x = np.linspace(0, two_pi, 1000)
y = np.linspace(0, two_pi, 1000)

# create matrices for plotting
X, Y = np.meshgrid(x, y)

# calculate z-points
Z = np.sin(X) * np.sin(Y)
```

In [19]:
```
# generate plot
fig3 = go.Figure(data=[go.Surface(x=X, y=Y, z=Z,)])
fig3.update_layout(
    title='Graph of z = sin(x)sin(y)',
    scene=dict(
        xaxis_title='X axis',
        yaxis_title='Y axis',
        zaxis_title='Z axis',
        camera=dict(
            up=dict(x=0, y=0, z=1),
            center=dict(x=0, y=0, z=0.25),
            eye=dict(x=2, y=2, z=1.5)
        )
    ),
    width=800,
    height=800
)
```

```
# create gradient descent path using stored points
path_trace = go.Scatter3d(
    x=x_path,
    y=y_path,
    z=z_path,
    mode='lines',
    line=dict(
        color='red',
        width=5
    ),
    name='Steepest Descent Path'
)

# add path
fig3.add_trace(path_trace)
fig3.show()
```

Conclusion

So there we have it—gradient descent from the ground up! By working from the first principles, we have (hopefully) gained a deeper intuition for how this algorithm works and the calculus principles that power it's operation.

Thank you for joining me on this journey!