\*\*aksam en son bunu özetle\*\*

Your code already implements a function with two parameters (x, y) for optimization, and the gradient function returns two values: one for the partial derivative with respect to \(x\) (denoted as \(d/dx\)) and one for the partial derivative with respect to \(y\) (denoted as \(d/dy\)).

Here's a breakdown of how your code achieves this:

1. \*\*Function `f(point)`\*\*: This function defines the function \( f(x, y) = x^2 + y^2 \) that you want to optimize (minimize in this case). It takes a point as an argument, where the point is a tuple or list containing two elements, \( x \) and \( y \).

2. \*\*Function `g(point)`\*\*: This is your gradient function. It computes the gradient of the function \( f(x, y) \) at a given point `(x, y)`. The gradient of a function is a vector that points in the direction of the greatest rate of increase of the function. In the case of your function \( f(x, y) = x^2 + y^2 \), the gradient is \( \nabla f(x, y) = [2x, 2y] \). This means:

- The partial derivative with respect to \(x\) (\(d/dx\)) is \(2x\).

- The partial derivative with respect to \(y\) (\(d/dy\)) is \(2y\).

3. \*\*How `g(point)` Works\*\*: When you call `g(point)` with a point like `[x, y]`, it returns a NumPy array `[2\*x, 2\*y]`. This array contains two values: the first one is the partial derivative of \(f\) with respect to \(x\) at the point `(x, y)`, and the second one is the partial derivative of \(f\) with respect to \(y\) at the same point.

In summary, your code already does what you're asking for: it uses a function with two parameters (`x` and `y`) for optimization, and the gradient function (`g(point)`) returns the two values representing the partial derivatives \(d/dx\) and \(d/dy\). This is exactly the setup needed for gradient descent in a two-dimensional space.

**Bak   
  
Creating the 2D grids using `np.meshgrid` and then flattening them with `np.ravel()` might seem roundabout at first, but this approach is actually very useful for evaluating functions over a two-dimensional domain, especially when preparing data for surface plots. Here's why:**

1. \*\*2D Grid Creation with `np.meshgrid`\*\*:

- The purpose of creating 2D grids with `np.meshgrid` is to generate a grid of points covering the area of interest in the 2D space.

- `np.meshgrid` takes one-dimensional arrays of x and y coordinates and produces two 2D arrays: one for the x-coordinates (`X`) and one for the y-coordinates (`Y`) of each point in the grid. This is essential for surface plots because each point in the grid corresponds to a point on the surface.

2. \*\*Flattening for Function Evaluation\*\*:

- To evaluate the function `f` at each point in this grid, it's convenient to iterate over the grid points. However, `f` is designed to take a single point (an array or list of two numbers) as its input.

- By flattening the 2D arrays `X` and `Y` using `np.ravel()`, you convert the grid into a format that makes it easy to iterate over each point. The `zip` function then pairs each x-coordinate with its corresponding y-coordinate, effectively recreating the individual points of the grid, but in a format that's easy to loop over.

3. \*\*Why Not Just Use 1D Arrays Directly?\*\*:

- You might wonder why not just use the original 1D arrays of x and y coordinates directly. The reason is that `np.meshgrid` produces a comprehensive 2D grid, which covers the combination of every x-coordinate with every y-coordinate.

- If you only had the 1D arrays, you'd have to manually pair each x with each y, which is exactly what `np.meshgrid` does efficiently.

4. \*\*Surface Plot Requirements\*\*:

- For surface plotting, you need the original 2D arrays `X`, `Y`, and the corresponding function values at each grid point. While the function values are calculated using the flattened arrays (for ease of iteration), the surface plot itself uses the 2D structure of `X`, `Y`, and the reshaped function values array to properly render the surface.

In summary, the 2D grids are created to map out the entire area over which we want to evaluate the function and then plot it. The flattening step is a necessary intermediate step for efficient function evaluation at each point in this grid. The combination of these steps allows for both an efficient calculation of the function values across the grid and the correct representation of these values in a 3D plot.