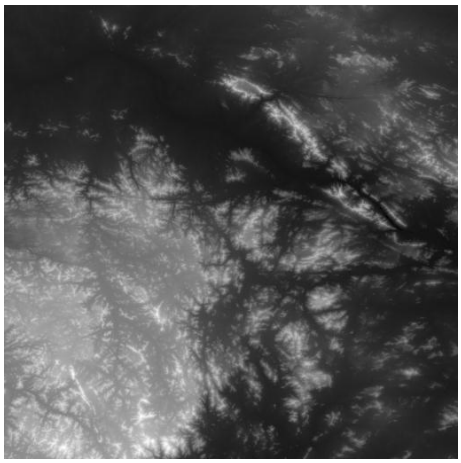
ASSIGNMENT 03

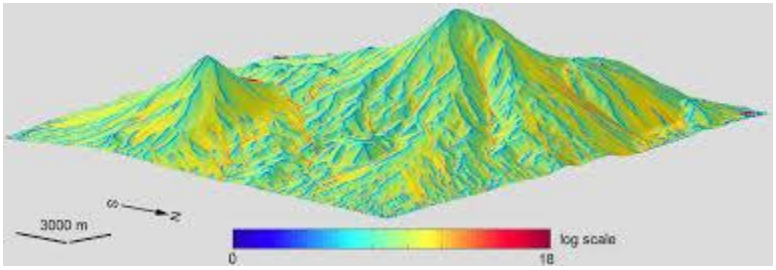
1. Digital Elevation Model and Catchment Prediction from Satellite Data

Use Digital Elevation Model (DEM) data to identify catchment areas by simulating water flow using gradient descent. The goal is to understand how water would naturally flow across a landscape and accumulate in catchment basins.



(This .jpg file has been given separately.)

A representative image in 3d is shown (you are not expected to plot the 3d plot; 2d contour plot etc. will suffice).



You may use the jpg image provided (Cartosat data for latitude 20.31 and longitude 84.09) or choose a DEM data from Bhoonidhi website (or any other satellite data such as Asterr etc.)

Task [1]: **The image which is grayscale, shows the peaks as bright and valleys as dark. You may load the entire jpg into a numpy array. Once done, show the terrain using a contour plot. You can use a normalized map, meaning that the extent of the domain can be simply chosen as the pixel height and width of the image and the peak can be chosen as 1 while the valley can be chosen as 0. All intermediate values in the image are therefore going to intermediate altitudes. In reality, one can simply multiply this map of altitude between (0, 1) to the appropriate scaling function, e.g. for Mt. Everest, we can multiply by 8828 m and get the actual height instead of 1.**

Hints:

1. Import libraries:

- NumPy for array operations

- Matplotlib for plotting

- PIL for image processing

Syntax :

```
import numpy as np              # Import NumPy for numerical operations on arrays
import matplotlib.pyplot as plt # Import Matplotlib for creating plots
from PIL import Image           # Import PIL (Pillow) for image processing
```

2. Load the grayscale image:

- Use PIL to open the image file

- Convert the image to grayscale mode:

Syntax: image = Image.open('terrain.jpg').convert('L')

3. Convert the image to a NumPy array:

- Convert the grayscale image into a NumPy array

Syntax : dem_data = np.array(image)

4. Normalize the pixel values:

- Divide the pixel values by 255 to scale them to the range [0, 1]

Syntax: dem_data = dem_data / 255.0

5. Plot the terrain:

- Create a new figure for the plot

- Generate a filled contour plot of the normalized data

- Add a colorbar with a label indicating normalized elevation

- Set the title and axis labels for the plot

- Display the plot

Syntax:

```
plt.figure(figsize=(10, 8))                    # Create a new figure for the plot
plt.contourf(dem_data, cmap=plt.cm.terrain)    # Create a filled contour plot with the terrain colormap
plt.colorbar(label='Normalized Elevation')     # Add a colorbar to the plot with a label
plt.title('Contour Plot of Terrain')           # Set the title of the plot
plt.xlabel('X-axis')                           # Label the x-axis
plt.ylabel('Y-axis')                           # Label the y-axis
plt.show()                                     # Display the plot
```

Task [2]: Your task is to run the gradient descent for a certain number of iterations. Create a function which will run the gradient descent.

Task[3]: The output of the function must be just the coordinates of the minima, i.e. the final (x,y) coordinate. Plot this overlaid on the contour plot as a blue dot.

The inputs to the function, amongst other things, would be the initial location of the particle, the number of iterations, and any other inputs which you deem necessary.

a. **Initial Point:** This initial point would be passed by the main function to the function. The scope of the initial point, will therefore, remain as local to the function

b. **Gradient Calculation:** At each step, calculate the gradient of the elevation, which indicates the steepest descent direction.
c. **Update Rule:** Move to the neighboring grid point that has the lowest elevation (in the direction of the gradient).
d. **Termination:** Stop when you reach a point where all surrounding points are higher, indicating a local minimum or catchment area. This would happen at either the end of max iterations, or when a certain threshold is reached.
e. **The function should resemble something like:**
   def find_catchment(dem_data, x_start, y_start, tol=1e-3, max_iter=1000):

Hints:

Function find_catchment(dem_data, x_start, y_start, tol=1e-3, max_iter=1000):

1. Initialize:

- Set current position (x, y) to (x_start, y_start)

- Define the size of the DEM data: num_rows, num_cols

2. For iteration from 1 to max_iter:

a. Compute the Gradient:

- Use `np.gradient` to estimate the gradient at the current point (x, y).

- `np.gradient` automatically computes the gradient of the `dem_data` array.

- Example Syntax: `grad_y, grad_x = np.gradient(dem_data)`

- The gradient components at the current point (x, y) can be accessed as:

- `grad_x_value , grad_y_value = np.gradient[dem_data, x, y]`

c. Find Neighboring Points and Move:

- Calculate new potential positions: (x_new, y_new)

- Move in the direction of the steepest descent:

- x_new = x - learning_rate * grad_x_value

- y_new = y - learning_rate * grad_y_value

d. Boundary Check:

- Ensure (x_new, y_new) is within the bounds of the DEM data

- If out of bounds, clip to the nearest valid position

e. Check Local Minimum:

- Evaluate elevation at (x_new, y_new)

- Compare to elevations of surrounding points:

- If all neighboring points have higher elevations, stop

f. Check Convergence:

- Calculate the change in position: delta_x = |x_new - x|, delta_y = |y_new - y|

- If delta_x < tol and delta_y < tol, convergence achieved

- Update (x, y) to (x_new, y_new)

3. Return the final position (x, y) as the coordinates of the catchment area

Tak[4]: Run the program for atleast 10000 randomly generated initial points. (You may generate the random points using the random number generator in numpy). For this, you have to call the function which you have just created in a loop which will create a new set of random initial points for the total number of points that you want to run the code.

Hints:

1. Set the number of random points (e.g., 10,000).

2. Initialize a list to store the coordinates of the catchment areas.

Syntax: n_points = 10000

np.random.seed(42) # For reproducibility

points = []

3. For each point:

- Generate a random x-coordinate within the width of the DEM.

- Generate a random y-coordinate within the height of the DEM.

- Run the gradient descent function to find the catchment area from the generated point.

- Store the coordinates of the catchment area.

Syntax:

```
for _ in range(n_points):
    x_start = np.random.randint(0, dem_data.shape[1])
    y_start = np.random.randint(0, dem_data.shape[0])
    x_min, y_min = find_catchment(dem_data, x_start, y_start)
    points.append((x_min, y_min))
```

Task[5]: The final plot should reveal the catchment area, i.e. where rainfall will run-off.

Task[6]: Bonus: You can plot the DEM data as a surface plot as well using the 3d plotting function available in matplotlib:

Function Call	Syntax	Meaning
Image.open()	Image.open('filename')	Opens an image file.
np.array()	np.array(image)	Converts an image to a NumPy array
np.clip()	np.clip(array,min_value,max_value)	Clips values in an array to a specified range.
plt.figure()	plt.figure(figsize=(width, height))	Creates a new figure with specified size.
plt.contourf()	plt.contourf(X, Y, Z, cmap=colormap)	Creates a filled contour plot.
np.random.seed()	np.random.seed(seed_value)	Sets the seed for the random number generator.
np.random.randint()	np.random.randint(low, high)	Generates random integers within a range.

2. Exploring isotherms and heat fluxes

```
import numpy as np

# Parameters

nx, ny = 50, 50  # Grid size

T_bottom = 100.0  # Fixed temperature on the bottom wall

T_left = 50.0    # Fixed temperature on the left wall

heat_source_value = 500.0  # Intensity of the heat source

# Initialize the temperature grid

temperature = np.zeros((ny, nx))

# Apply fixed boundary conditions

temperature[-1, :] = T_bottom   # Bottom wall (last row)

temperature[:, 0] = T_left      # Left wall (first column)

# Simulate the effect of the heat source (placed in the middle of the domain)

heat_source_position = (ny//2, nx//2)

temperature[heat_source_position] = heat_source_value

# Simple diffusion-like spread from the heat source (for demonstration)

For i in range(1, ny-1):

    for j in range(1, nx-1):

        if (i , j) != heat_source_position:

temperature[i, j] = 0.25 * (temperature[i+1, j] + temperature[i-1, j] +

temperature[i, j+1] + temperature[i, j-1])
```

Save the temperature data to a file

np.savetxt('temperature_data_with_heat_source.txt', temperature, fmt='%.2f')

This will create a file with the filename given in the last line of the listing. The specifics of the finite difference code is that it is the solution for the temperature field in a plate of size 1x1 with a grid of 50x50 where the bottom wall and left wall are kept at 100 and 50 degree celcius respectively. The other two walls are adiabatic. There is also a heat source in the middle of the domain with a fixed intensity 100 units.

TASK [1]: Modify the above code so that apart from the temperature, the x and y coordinates are also printed.

Hint:

- **Coordinate Creation**:
  - **Initial Code**: No coordinates are included.
  - **Modified Code**: Generates x and y coordinate grids using `np.meshgrid`.
- **Flattening Arrays**:
  - **Initial Code**: Only temperature data is handled.
  - **Modified Code**: Flattens the 2D coordinate and temperature arrays into 1D arrays using `.flatten()`.
- **Combining Data**:
  - **Initial Code**: Directly saves the temperature data.
  - **Modified Code**: Stacks x, y, and temperature data into a single 2D array using `np.vstack()`.
- **Saving Data**:
  - **Initial Code**: Saves only the temperature data.
  - **Modified Code**: Saves x coordinates, y coordinates, and temperature data in one file with a header. You can print multiple items from a single np.savetxt command just like you would in print command

- **File Content**:
    - **Initial Code**: The file contains only temperature values.
    - **Modified Code**: The file includes columns for x coordinates, y coordinates, and temperature values.

Task [2]: Load the data in a separate cell/file and visualize the isotherms of temperature after generating a finer meshgrid and interpolating the data over the meshgrid.

Hint:

1. **Data Loading**:
    - Use np.loadtxt to load the saved file containing x, y, and temperature values.
    - Skip the header row if one exists.
2. **Finer Meshgrid**:
    - Create a meshgrid with a higher resolution than the original grid using np.linspace.
    - This will allow you to perform interpolation on a finer grid.
3. **Interpolation**:
    - Use scipy.interpolate.griddata to interpolate the temperature values onto the finer meshgrid.
    - Choose an appropriate interpolation method like 'cubic' for smooth results.
4. **Isotherms Visualization**:
    - Use matplotlib.pyplot.contourf to plot the interpolated temperature data as isotherms.
    - Customize the number of contour levels and the color map to enhance the visualization.

Pseudo Code

1. **Load Data**:
    - Read the data file containing x, y coordinates, and temperature values.
2. **Extract and Prepare Data**:
    - Separate the x, y coordinates and temperature values from the loaded data.
3. **Generate Finer Meshgrid**:
    - Create a finer x, y grid using np.linspace to define a higher resolution.
4. **Interpolate Temperature Data**:
    - Use griddata to interpolate the temperature onto the finer meshgrid.
5. **Plot Isotherms**:
    - Plot the interpolated temperature data using contour to display the isotherms.
    - Add labels, a title, and a color bar for clarity.

Task [3]: Take a numerical gradient of the data and visualize the heat flux vectors using the quiver function of matplotlib. The gradient calculation has to be done numerically

over the meshgrid you have defined earlier. Hint: you can use delta_x and delta_y for calculation of the gradient. Hint 2: Non-Uniform Grids: If the grid is non-uniform, pass the actual grid arrays (e.g., x, y) to np.gradient instead of constant delta_x and delta_y.

Hint:

1. **Numerical Gradient:**
   - **Use np.gradient to compute the numerical gradient of the temperature data.**
   - **Pass the meshgrid arrays (x_fine, y_fine) and the temperature data to np.gradient to obtain the partial derivatives ($\partial T/\partial x$ and $\partial T/\partial y$), which represent the heat flux components in the x and y directions.**
2. **Heat Flux Calculation:**
   - **The heat flux vector components can be visualized using the gradients: $qx=-k\partial T/\partial x$ and $qy=-k\partial T/\partial y$, where k is the thermal conductivity (you can assume k=1).**
3. **Visualization with quiver:**
   - **Use matplotlib.pyplot.quiver to plot the heat flux vectors.**
   - **Pass the meshgrid (x_fine_grid, y_fine_grid) and the gradient components to quiver to create a vector field plot.**
4. **Non-Uniform Grids:**
   - **If using a non-uniform grid, pass the actual grid arrays (x_fine, y_fine) to np.gradient instead of constant delta_x and delta_y.**

Pseudo Code

1. **Calculate Gradient:**
   - **Compute the gradient of the interpolated temperature data with respect to x and y using np.gradient.**
2. **Compute Heat Flux Components:**
   - **Multiply the negative of the gradient components by a constant (e.g., k = 1) to get the heat flux vectors.**
3. **Visualize with quiver:**
   - **Use quiver to plot the heat flux vectors on the finer meshgrid.**

Task [4]: Analyze how the temperature varies along the x-direction for different fixed y-values (cut-lines).

a. **Choose Cut-Lines: Select different values of y (e.g., y = 0.25, y = 0.5, and y = 0.75). You can simply interpolate using appropriate values of (x,y) over the cut lines.**

b. **Extract Temperature Profiles: For each cut-line, extract the temperature along the x-direction.**
c. **Plot the Profiles: Plot the temperature profiles for each cut-line on the same graph for comparison.**

Hint:

1. **Choose Cut-Lines**:
   - Select different y-values (e.g., y=0.25y = 0.25y=0.25, y=0.5y = 0.5y=0.5, and y=0.75y = 0.75y=0.75) where you want to analyze the temperature variation.
   - Interpolate the temperature data along these specific y-values.
2. **Extract Temperature Profiles**:
   - For each selected y-value, extract the corresponding temperature values along the x-direction.
   - Use the griddata function or another interpolation method to get temperature data at the specified cut-lines.
3. **Plot the Profiles**:
   - Plot the temperature profiles for each selected y-value on the same graph.
   - Label each profile with its corresponding y-value for clear comparison.

Pseudo Code

1. **Choose Cut-Lines**:
   - Define y-values for the cut-lines, e.g., y_cuts = [0.25, 0.5, 0.75] * y_fine.max().
2. **Extract Temperature Profiles**:
   - Loop through the y_cuts values.
   - For each y_cut, interpolate the temperature data along the x-direction at that specific y-value.
3. **Plot the Profiles**:
   - For each y_cut, plot the temperature vs. x on the same graph.
   - Add legends, labels, and a title for clarity

FUNCTION	PURPOSE
np.loadtxt()	Load data from a text file
griddata()	Interpolate data onto a new grid
np.gradient	Compute the gradient of an array

3. Steepest descent for double-well potential

Consider the function: $f(x,y) = (x^2-1)^2 + y^2$

This function has two minimas, one at x = 1 and the other at x = -1

TASK [1]: **A function which would implement steepest descent algorithm for mimima finding.**

TASK [2]: The output of the function must be final iterated point.

Hints: def steepest_descent(x0, y0, alpha=0.1, tol=1e-6, max_iter=1000):

1. Define the Function and Gradient

Function: - Define f(x, y) = (x^2 - 1)^2 + y^2

Gradient: - Compute the gradient components:

- ∂f/∂x = 4 * x * (x^2 - 1)

- ∂f/∂y = 2 * y

- The gradient vector is [∂f/∂x, ∂f/∂y]

Pseudocode:

```
Function f(x, y):
    return (x^2 - 1)^2 + y^2
Function grad_f(x, y):
    df_dx = 4 * x * (x^2 - 1)
    df_dy = 2 * y
    return [df_dx, df_dy]
```

2. Initialize the Algorithm

Starting Point: - Set the initial point using x0 and y0.

Path Tracking:

- Create a list to keep track of points visited during the iterations.

Pseudocode:

Initialize point = [x0, y0]

Initialize path = [point] // Store the initial point

3. Iteration Process

Calculate Gradient:

- Compute the gradient at the current point.

Update Rule:

- Update the current point using:

new_point = current_point - alpha * gradient

Stopping Criteria:

- Check if the magnitude of the gradient is less than tolerance tol.

- If yes, stop the iterations.

- If no, continue until reaching max_iter.

Pseudocode:

For iteration from 1 to max_iter:

gradient = grad_f(point[0], point[1])

norm_grad = magnitude of gradient

If norm_grad < tol:

Break the loop // Converged

point = point - alpha * gradient // Update point

Append point to path // Track the path

4. Return Values

Final Output:

- Return the final point and the path taken during the iterations.

Pseudocode:

Return point

TASK [3]: In the double well potential, the steepest descent method's final solution depends heavily on the initial point (x0,y0). If the initial point is closer to the left well, the method will

likely converge to (−1,0). If closer to the right well, it will converge to (1,0). This behavior illustrates the importance of choosing appropriate initial conditions in optimization problems. Establish this by using a bunch of randomly generated pairs of points over the domain x in [-2, 2] and y in [-1, 1] using numpy.rand.

Hints:

Generate Random Initial Points

Function generate_random_points(num_points):

x_values = Random values in range [-2, 2] // Generate random x-coordinates
y_values = Random values in range [-1, 1] // Generate random y-coordinates
Return x_values, y_values // Return the lists of random x and y values

3. Run Steepest Descent for Each Initial Point

Define num_points = 20 // Number of random initial points to test
Define alpha = 0.1 // Learning rate
Define tol = 1e-6 // Convergence tolerance
Define max_iter = 1000 // Maximum number of iterations

x_values, y_values = generate_random_points(num_points) // Generate initial points
Initialize final_points list // List to store the final points after convergence

For each (x0, y0) in zip(x_values, y_values):
final_point, _ = steepest_descent(x0, y0, alpha, tol, max_iter) // Run the algorithm
Append final_point to final_points list // Store the result

4. Plot the Final Points
Function plot_final_points(final_points):
Create scatter plot with final_points // Plot the final points
Add labels for x and y axes

Add a title to the plot

Display the plot

Call plot_final_points(final_points) // Visualize the results

TASK [4]: Variation of convergence for different relaxation factor alpha. How does the choice of alpha affect the convergence speed and the optimization path? Can you modify the function to also show the path taken by the intial point towards convergence?

Hints:

Modify the Steepest Descent Function

Initialize a list to store the path taken at each iteration. Append the current point to this list after updating it in each iteration.

Pseudocode:

Function steepest_descent(x0, y0, alpha, tol=1e-6, max_iter=1000): Initialize point = [x0, y0]

Define f(x, y) as (x^2 - 1)^2 + y^2

Define grad_f(x, y) as the gradient of f: df_dx = 4 * x * (x^2 - 1)

df_dy = 2 * y

Initialize point = [x0, y0]

Initialize path = [point] // List to track the path taken

For iteration from 1 to max_iter:

Compute gradient at current point using grad_f

Compute norm of gradient

If norm of gradient < tol:

Break the loop // Convergence criterion met

Update point using: point = point - alpha * gradient

Append updated point to path // Track the path

Return final point, path // Return both the final point and the path

Define alpha_values as [0.01, 0.1, 0.5, 1.0] // Example values for alpha to test

Define num_points as 5 // Number of initial points to test Generate random initial points (x_values, y_values) within the range [-2, 2] for x and [-1, 1] for y

Initialize paths_by_alpha as an empty dictionary to store paths for each alpha

For each alpha in alpha_values: Initialize an empty list for storing paths for current alpha For each (x0, y0) in zip(x_values, y_values): Call steepest_descent(x0, y0, alpha) // Run the algorithm Append path to the list for current alpha

Task [5]: **Modify the steepest_descent function to return the number of iterations required to converge for different initial conditions. Compare these results.**

Hints: Function steepest_descent(x0, y0, alpha, tol=1e-6, max_iter=1000):

Define f(x, y) as (x^2 - 1)^2 + y^2

Define grad_f(x, y) as the gradient of f:

df_dx = 4 * x * (x^2 - 1)

df_dy = 2 * y

Initialize point = [x0, y0]

Initialize path = [point] // Track the path taken

Initialize iteration_count = 0 // Counter for iterations

For iteration from 1 to max_iter:

Increment iteration_count

Compute gradient at current point using grad_f

Compute norm of gradient

If norm of gradient < tol:

Break the loop // Convergence criterion met

Update point using: point = point - alpha * gradient

Append updated point to path // Track the path

Return final point, path, iteration_count // Return final point, path, and number of iterations