ChatGPT

```
import numpy as np
import pandas as pd
from scipy.interpolate import Rbf
import matplotlib.pyplot as plt
from noise import pnoise2
from scipy.fft import fft2, ifft2, fftshift
from scipy.stats import norm
import gstools as gs  # Add gstools import

# ------------------------------------------------------------------------
# STEP 1: Read in your real dataset
# ------------------------------------------------------------------------
# Assume your data is stored in 'real_data.csv' with columns: X, Y, Z
df_real = pd.read_excel('/Users/akshat/Documents/WORK/Github/Runopt/InputFile_NEW.xlsx')

# Extract NumPy arrays
x_real = df_real['X'].values
y_real = df_real['Y'].values
z_real = df_real['Z (Existing)'].values

# ------------------------------------------------------------------------
# STEP 1.1: Sample the real dataset to reduce memory usage
# ------------------------------------------------------------------------
sample_size = 10000  # Adjust this number based on your memory constraints
if len(x_real) > sample_size:
    indices = np.random.choice(len(x_real), sample_size, replace=False)
    x_real = x_real[indices]
    y_real = y_real[indices]
    z_real = z_real[indices]

# ------------------------------------------------------------------------
# STEP 2: Fit an RBF interpolator
# ------------------------------------------------------------------------
```

```
# - function='thin_plate' or 'multiquadric' are common choices
# - Adjust 'epsilon' or 'smooth' to tweak how tightly/loosely it fits.
rbf_func = Rbf(x_real, y_real, z_real, function='multiquadric', smooth=1)

# ------------------------------------------------------------------------
# STEP 3: Define a new grid where you'll generate synthetic data
# ------------------------------------------------------------------------
# First, calculate the min and max values from real data
x_min, x_max = np.min(x_real), np.max(x_real)
y_min, y_max = np.min(y_real), np.max(y_real)

# Then calculate the expanded boundaries
margin_factor = 0.1  # 10% extension on each side
x_range = x_max - x_min
y_range = y_max - y_min
x_min_ext = x_min - margin_factor * x_range
x_max_ext = x_max + margin_factor * x_range
y_min_ext = y_min - margin_factor * y_range
y_max_ext = y_max + margin_factor * y_range

num_points = 100  # Increased number of points per dimension for higher resolution
x_synth = np.linspace(x_min_ext, x_max_ext, num_points)
y_synth = np.linspace(y_min_ext, y_max_ext, num_points)

# Create a mesh for evaluation
X_synth, Y_synth = np.meshgrid(x_synth, y_synth)
X_flat = X_synth.flatten()
Y_flat = Y_synth.flatten()

# ------------------------------------------------------------------------
# STEP 4: Evaluate the RBF to create synthetic Z
# ------------------------------------------------------------------------
z_synth = rbf_func(X_flat, Y_flat)

# ------------------------------------------------------------------------
# STEP 5: Synthetic Data Generation Methods
# ------------------------------------------------------------------------
```

```
def generate_perlin_terrain(num_points, x_range=(0, 10),
y_range=(0, 10)):
    """Generate terrain using Perlin noise"""
    x = np.linspace(x_range[0], x_range[1], num_points)
    y = np.linspace(y_range[0], y_range[1], num_points)
    X, Y = np.meshgrid(x, y)

    # Parameters for Perlin Noise
    scale = 100.0
    octaves = 6
    persistence = 0.5
    lacunarity = 2.0

    Z = np.zeros((num_points, num_points))
    for i in range(num_points):
        for j in range(num_points):
            Z[i][j] = pnoise2(X[i][j] / scale,
                              Y[i][j] / scale,
                              octaves=octaves,
                              persistence=persistence,
                              lacunarity=lacunarity,
                              repeatx=1024,
                              repeaty=1024,
                              base=42)

    # Normalize Z
    Z_min, Z_max = np.min(Z), np.max(Z)
    Z_normalized = 1000 * (Z - Z_min) / (Z_max - Z_min)

    return X, Y, Z_normalized

def generate_grf_terrain(num_points, x_range=(0, 10),
y_range=(0, 10)):
    """Generate terrain using Gaussian Random Fields"""
    x = np.linspace(x_range[0], x_range[1], num_points)
    y = np.linspace(y_range[0], y_range[1], num_points)
    X, Y = np.meshgrid(x, y)

    # GRF parameters
    model = gs.Spherical(dim=2, var=1.0, len_scale=1.0)
    srf = gs.SRF(model, seed=42)
```

```
    # Generate GRF values
    Z = srf.structured([x, y])

    # Normalize Z
    Z_min, Z_max = np.min(Z), np.max(Z)
    Z_normalized = 1000 * (Z - Z_min) / (Z_max - Z_min)

    return X, Y, Z_normalized

# Modify STEP 5 to use both methods
# Generate base terrain using RBF
z_synth_rbf = rbf_func(X_flat, Y_flat).reshape((num_points,
num_points))

# Generate Perlin noise terrain
X_perlin, Y_perlin, z_perlin = generate_perlin_terrain(
    num_points,
    x_range=(x_min_ext, x_max_ext),
    y_range=(y_min_ext, y_max_ext)
)

# Generate GRF terrain
X_grf, Y_grf, z_grf = generate_grf_terrain(
    num_points,
    x_range=(x_min_ext, x_max_ext),
    y_range=(y_min_ext, y_max_ext)
)

# Combine all terrain types with weights
terrain_weights = {
    'rbf': 0.5,
    'perlin': 0.3,
    'grf': 0.2
}

z_synth_combined = (
    terrain_weights['rbf'] * z_synth_rbf +
    terrain_weights['perlin'] * z_perlin +
    terrain_weights['grf'] * z_grf
)

# Add some random noise for additional variation
```

```
noise_strength = 0.05
z_synth_final = z_synth_combined + noise_strength *
np.random.randn(*z_synth_combined.shape)

# Update the synthetic dataset creation
df_synth = pd.DataFrame({
    'X': X_flat,
    'Y': Y_flat,
    'Z (Synthetic)': z_synth_final.flatten()
})

# Save to CSV if desired
df_synth.to_csv('/Users/akshat/Documents/WORK/Github/R
unopt/synthetic_data.csv', index=False)

# ------------------------------------------------------------------------
# STEP 7: (Optional) Quick Visualization
# ------------------------------------------------------------------------
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X_flat, Y_flat, z_synth_final, c=z_synth_final,
cmap='terrain', marker='o', s=10)
ax.set_title('Synthetic Topography')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z (Synthetic)')
plt.show()
```

Below is a step-by-step walkthrough of what's happening in your code, along with some tips and potential enhancements:

Overall Flow

1. **Load Real Data**: Load a real-world terrain dataset from Excel.
2. **Optional Sampling**: Randomly sample the data to reduce the dataset size.

3. **RBF Interpolation**: Fit an RBF (Radial Basis Function) interpolator to the sampled real data.
4. **Set Up Grid**: Define a new (potentially expanded) grid over which synthetic data will be generated.
5. **Generate Synthetic Terrains**:
    - **Base Terrain via RBF**: Evaluate the RBF over the new grid.
    - **Perlin Noise**: Generate a Perlin-based terrain.
    - **Gaussian Random Fields (GRF)**: Generate a terrain using Gaussian Random Fields.
    - **Combine Terrains**: Weight and sum all terrains together, then add some random noise.
6. **Save & Visualize**: Store the synthetic dataset and create a 3D scatter plot to visualize it.

Step-by-Step Explanation

1. Loading Real Data

```python
df_real =
pd.read_excel('/Users/akshat/Documents/WORK/Github/Runopt/InputFile_NEW.xlsx')

x_real = df_real['X'].values
y_real = df_real['Y'].values
z_real = df_real['Z (Existing)'].values
```

- You're loading an Excel file using `pandas.read_excel` and extracting columns X, Y, Z as NumPy arrays.
- This provides the "ground truth" data points for your terrain.

---

2. Optional Data Sampling

python

```python
sample_size = 10000
if len(x_real) > sample_size:
    indices = np.random.choice(len(x_real), sample_size, replace=False)
    x_real = x_real[indices]
    y_real = y_real[indices]
    z_real = z_real[indices]
```

- If the real dataset is very large, you sample it down to `sample_size` points to reduce memory usage or computational overhead.
- `np.random.choice` ensures a random subset of points is taken.

Tip:

- If you want to preserve some spatial structure, consider a spatially stratified sampling method (e.g., using `scipy.spatial` techniques) instead of purely random sampling.

---

3. RBF Interpolation

python

```python
rbf_func = Rbf(x_real, y_real, z_real, function='multiquadric', smooth=1)
```

- You create an RBF function based on the sampled data.
- `function='multiquadric'` is a good default for terrain-like surfaces; you can experiment with others (`thin_plate`, `gaussian`, etc.).
- `smooth=1` controls how "smooth" or "tight" the fit is to your data.

Tip:

- If you see overfitting or underfitting, adjust `smooth` or `epsilon`.
- For large data, consider alternative interpolation techniques (e.g., `scipy.interpolate.griddata`, Kriging from `gstools`, or other geostatistical packages).

---

4. Creating the Synthetic Grid

python

```python
x_min, x_max = np.min(x_real), np.max(x_real)
y_min, y_max = np.min(y_real), np.max(y_real)

margin_factor = 0.1
x_range = x_max - x_min
y_range = y_max - y_min
x_min_ext = x_min - margin_factor * x_range
x_max_ext = x_max + margin_factor * x_range
y_min_ext = y_min - margin_factor * y_range
y_max_ext = y_max + margin_factor * y_range

num_points = 100
x_synth = np.linspace(x_min_ext, x_max_ext, num_points)
y_synth = np.linspace(y_min_ext, y_max_ext, num_points)

X_synth, Y_synth = np.meshgrid(x_synth, y_synth)
X_flat = X_synth.flatten()
Y_flat = Y_synth.flatten()
```

- This block defines the grid boundaries slightly larger than the real dataset bounds (by 10%).
- `np.meshgrid` creates a 2D grid of x and y coordinates.
- Flattening them with `.flatten()` makes it easier to pass into the RBF function and to construct a single DataFrame later.

Tip:

- The choice of `num_points=100` results in 100×100 = 10,000 grid points. Increase if you want a finer resolution (though the memory usage will also increase).

---

5. Generating Synthetic Terrains

5.1 RBF Terrain

python

```
z_synth_rbf = rbf_func(X_flat, Y_flat).reshape((num_points, num_points))
```

- Evaluates the RBF on the entire 2D grid to produce a “base” terrain that’s structurally similar to your real dataset.

5.2 Perlin Noise

python

```
def generate_perlin_terrain(num_points, x_range=(0, 10), y_range=(0, 10)):
    ...
    Z_normalized = 1000 * (Z - Z_min) / (Z_max - Z_min)
    return X, Y, Z_normalized
```

- Creates a Perlin Noise “terrain” of size $\text{num_points} \times \text{num_points}$.
- **Key parameters**:
  - `scale`: Controls the “zoom” level of Perlin noise.
  - `octaves`, `persistence`, `lacunarity`: Control the fractal character of the noise.
- The result is then normalized to a 0–1000 range.

Tip:

- If you want smaller or larger elevation ranges, change the `1000` scale factor or tweak the normalization.
- For more fractal complexity, experiment with octaves/persistence/lacunarity.

5.3 Gaussian Random Fields (GRF) using `gstools`

python

```
def generate_grf_terrain(num_points, x_range=(0, 10), y_range=(0, 10)):
    ...
    model = gs.Spherical(dim=2, var=1.0, len_scale=1.0)
    srf = gs.SRF(model, seed=42)
    ...
    Z_normalized = 1000 * (Z - Z_min) / (Z_max - Z_min)
    return X, Y, Z_normalized
```

- Uses `gstools` to create a 2D Gaussian Random Field.
- You're defining a **Spherical** variogram model with variance = 1.0 and `len_scale=1.0`.
- Similar to Perlin, it's normalized to 0–1000.

Tip:

- Adjust `model = gs.Spherical(...)` to try different variogram types or different parameters (e.g., `Exponential`, `Gaussian`, different `len_scale`).
- Kriging or sequential Gaussian simulation can also be used if you want to condition on known measurements.

5.4 Combining Terrains

```python
terrain_weights = {
    'rbf': 0.5,
    'perlin': 0.3,
    'grf': 0.2
}

z_synth_combined = (
    terrain_weights['rbf'] * z_synth_rbf +
    terrain_weights['perlin'] * z_perlin +
    terrain_weights['grf'] * z_grf
)
```

- Weighted linear combination of the three surfaces.
- You can tweak these weights based on how much "realism" vs. "randomness" you want.

5.5 Adding Random Noise

```python
noise_strength = 0.05
z_synth_final = z_synth_combined + noise_strength *
np.random.randn(*z_synth_combined.shape)
```

- Adds small-scale variations across the entire surface.
- `noise_strength` can be adjusted to control how noisy your final surface is.

6. Saving the Synthetic Data

```python
df_synth = pd.DataFrame({
    'X': X_flat,
    'Y': Y_flat,
    'Z (Synthetic)': z_synth_final.flatten()
})

df_synth.to_csv('/Users/akshat/Documents/WORK/Github/Runopt/synthetic_data.csv', index=False)
```

- Converts the final 2D array into a single column of coordinates and saves it.
- You now have a CSV with columns $X, Y, Z(\text{Synthetic})$.

Tip:

- If you want a grid-based data format, consider storing it in a NetCDF or GeoTIFF for easier integration with GIS tools.
- For large grids, CSV might become unwieldy.

---

7. Quick Visualization

```python
fig = plt.figure(figsize=(10,6))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X_flat, Y_flat, z_synth_final, c=z_synth_final, cmap='terrain', marker='o', s=10)
...
plt.show()
```

- Simple 3D scatter plot of your final synthetic terrain.

- For smoother visualization, you can use a surface plot:

  ```python
  ax.plot_surface(X_synth, Y_synth, z_synth_final, cmap='terrain')
  ```

  Just ensure `z_synth_final` is in a 2D form matching `X_synth` and `Y_synth`.

Potential Enhancements

1. **Spatial Sampling**: Instead of random sampling, use spatial stratification or uniform grid sampling to ensure coverage of the domain.
2. **Kriging-Based Interpolation**: If you're already using `gstools`, you could leverage its Kriging or IDW functionalities to interpolate from your real dataset in a geostatistically rigorous way.
3. **Variable Weights**: Dynamically adjust the weights of the terrains based on location. For instance, you could blend more of the RBF in areas close to your real data boundaries and rely more on Perlin/GRF in extrapolated regions.
4. **Variogram Fitting**: If you have the real data, you can fit a variogram model using `gstools` or other geostatistical libraries, then use those parameters (variance, range, sill, nugget) to generate a more realistic GRF.
5. **Resolution**: If 100 × 100 is not high-resolution enough, increase `num_points`. Just keep an eye on performance and memory usage.
6. **File Formats**: For large grids, consider using binary or GIS-oriented file formats (e.g., NetCDF, GeoTIFF, HDF5) instead of CSV.
7. **GPU Acceleration**: For very large datasets, libraries like `cupy` or GPU-accelerated frameworks might help speed up the computations.

Conclusion

Your script already follows a solid workflow for combining real data with synthetic noise-based and geostatistical methods to generate a plausible "synthetic" terrain. By adjusting

parameters and potentially using more advanced sampling and geostatistical methods, you can refine the realism and control the spatial characteristics of the output.

Can you remove the use of existing data, and ask user which topograph they want based on concept of TOPOGRAPHIC SURVEYS in Civil Engineering, the code should produce a real world data, also integrate Resolution

Below is an **example Python script** that generates entirely synthetic (yet somewhat "real-world-like") terrain data **without using any existing dataset**. It asks the user to select the terrain type and desired resolution—both of which influence the final output. The concept is loosely inspired by **topographic surveys** in civil engineering, where surveyors collect (X, Y, Z) points across a region.

In this script:

1. **No external data** is used.
2. **User input** determines:
    - Terrain type: e.g., *"Flat"*, *"Rolling Hills"*, or *"Mountainous"*.
    - Resolution (number of points in each dimension).
3. The code then generates a **synthetic topographic surface** using different mathematical models:
    - A baseline random shape.
    - Optional Perlin noise for more "natural" variations.
    - Optional Gaussian random fields (commented out, can be included if desired).
4. Finally, the code **plots** the synthetic terrain and **saves** the data as (X, Y, Z) into a CSV file.

Full Code

python

```python
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
# from noise import pnoise2  # Uncomment if you want Perlin noise
# import gstools as gs      # Uncomment if you want to use Gaussian Random Fields

def generate_flat_terrain(nx, ny, extent=1000):
    """
    Generate a flat terrain (Z = constant).
    """
    x = np.linspace(0, extent, nx)
    y = np.linspace(0, extent, ny)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros_like(X)  # flat terrain at Z=0
    return X, Y, Z

def generate_rolling_hills(nx, ny, extent=1000, hill_height=100):
    """
    Generate rolling hills using a combination of sinusoidal waves.
    """
    x = np.linspace(0, extent, nx)
    y = np.linspace(0, extent, ny)
    X, Y = np.meshgrid(x, y)

    # Combine multiple sin/cos waves
    Z = (
        hill_height * np.sin(2 * np.pi * X / (extent/2)) * np.cos(2 * np.pi * Y /
(extent/3))
        + hill_height * 0.2 * np.cos(2 * np.pi * Y / (extent/4))
        + hill_height * 0.2 * np.sin(2 * np.pi * X / (extent/5))
    )

    # Optional: Add some random small-scale noise
    noise_strength = 0.05 * hill_height
    Z += noise_strength * np.random.randn(*Z.shape)

    return X, Y, Z

def generate_mountainous_terrain(nx, ny, extent=1000, max_height=500):
    """
    Generate mountainous terrain using random radial peaks + optional noise.
    """
```

```
    x = np.linspace(0, extent, nx)
    y = np.linspace(0, extent, ny)
    X, Y = np.meshgrid(x, y)

    # Create random "peaks" by summing multiple 2D Gaussians
    num_peaks = 5  # number of random peaks
    Z = np.zeros_like(X, dtype=float)

    np.random.seed(42)  # for reproducibility

    for _ in range(num_peaks):
        peak_x = np.random.uniform(0, extent)
        peak_y = np.random.uniform(0, extent)
        peak_height = np.random.uniform(0.5 * max_height, max_height)
        spread_x = np.random.uniform(50, 200)  # horizontal spread
        spread_y = np.random.uniform(50, 200)  # vertical spread

        # 2D Gaussian for each peak
        Z_peak = peak_height * np.exp(
            -(((X - peak_x) ** 2) / (2 * spread_x**2) + ((Y - peak_y) ** 2) / (2 *
spread_y**2))
        )
        Z += Z_peak

    # Optional: add random "roughness" to mimic smaller features
    roughness_strength = 0.05 * max_height
    Z += roughness_strength * np.random.randn(*Z.shape)

    return X, Y, Z

# ----------------------------------------------------------------------
# (Optional) If you want to integrate Perlin Noise, uncomment & install 'noise'
library
# ----------------------------------------------------------------------
# def generate_perlin_noise_terrain(nx, ny, extent=1000, scale=50.0):
#     """
#     Generate terrain using Perlin noise.
#     """
#     x = np.linspace(0, extent, nx)
#     y = np.linspace(0, extent, ny)
#     X, Y = np.meshgrid(x, y)
#
```

```
#     Z = np.zeros((ny, nx))
#     octaves = 4
#     persistence = 0.5
#     lacunarity = 2.0
#
#     for i in range(ny):
#         for j in range(nx):
#             Z[i][j] = pnoise2(
#                 X[i][j] / scale,
#                 Y[i][j] / scale,
#                 octaves=octaves,
#                 persistence=persistence,
#                 lacunarity=lacunarity,
#                 repeatx=999999,  # big repeats if you like
#                 repeaty=999999,
#                 base=42
#             )
#
#     # Scale Perlin noise to 0..max_height range
#     Z_min, Z_max = Z.min(), Z.max()
#     Z_normalized = (Z - Z_min) / (Z_max - Z_min)
#     max_height = 300
#     Z_final = Z_normalized * max_height
#
#     return X, Y, Z_final

# ---------------------------------------------------------------------
# (Optional) If you want to use Gaussian Random Fields via gstools
# ---------------------------------------------------------------------
# def generate_grf_terrain(nx, ny, extent=1000):
#     x = np.linspace(0, extent, nx)
#     y = np.linspace(0, extent, ny)
#     X, Y = np.meshgrid(x, y)
#
#     model = gs.Exponential(dim=2, var=30000, len_scale=100)
#     srf = gs.SRF(model, seed=2024)
#
#     # structured() requires [x, y] arrays in ascending order
#     Z = srf.structured([x, y])
#
#     # Optionally scale to match a realistic height range
#     Z_min, Z_max = Z.min(), Z.max()
```

```
#     Z_scaled = (Z - Z_min) / (Z_max - Z_min) * 500  # e.g. up to 500 meters
#     return X, Y, Z_scaled

def main():
    # ------------------------------------------------------------
    # 1. Ask user for terrain type
    # ------------------------------------------------------------
    print("Select the type of terrain you want to generate:")
    print("1. Flat")
    print("2. Rolling Hills")
    print("3. Mountainous")
    # Uncomment or add more options if using Perlin or GRF:
    # print("4. Perlin Noise")
    # print("5. Gaussian Random Field")

    choice = input("Enter a number (1/2/3): ")

    # ------------------------------------------------------------
    # 2. Ask user for resolution
    # ------------------------------------------------------------
    resolution_str = input("Enter resolution (e.g., 100 for a 100x100 grid): ")
    try:
        resolution = int(resolution_str)
        if resolution < 10:
            resolution = 10  # enforce a minimum
    except ValueError:
        resolution = 100  # default

    # ------------------------------------------------------------
    # 3. Generate the chosen terrain
    # ------------------------------------------------------------
    if choice == "1":
        X, Y, Z = generate_flat_terrain(resolution, resolution)
        terrain_name = "Flat"
    elif choice == "2":
        X, Y, Z = generate_rolling_hills(resolution, resolution)
        terrain_name = "RollingHills"
    elif choice == "3":
        X, Y, Z = generate_mountainous_terrain(resolution, resolution)
        terrain_name = "Mountainous"
    else:
        print("Invalid choice. Defaulting to 'Flat' terrain.")
```

```
        X, Y, Z = generate_flat_terrain(resolution, resolution)
        terrain_name = "Flat"

    # ------------------------------------------------------------
    # 4. Visualize the synthetic terrain
    # ------------------------------------------------------------
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111, projection='3d')
    ax.plot_surface(X, Y, Z, cmap='terrain', linewidth=0, antialiased=False)

    ax.set_title(f"Synthetic {terrain_name} Terrain ({resolution}x{resolution})")
    ax.set_xlabel('X')
    ax.set_ylabel('Y')
    ax.set_zlabel('Z')
    plt.tight_layout()
    plt.show()

    # ------------------------------------------------------------
    # 5. Save the data to a CSV file
    # ------------------------------------------------------------
    # Flatten arrays for CSV
    df = pd.DataFrame({
        'X': X.flatten(),
        'Y': Y.flatten(),
        'Z': Z.flatten()
    })
    filename = f"synthetic_{terrain_name}_{resolution}x{resolution}.csv"
    df.to_csv(filename, index=False)
    print(f"Terrain data saved to '{filename}'")

if __name__ == "__main__":
    main()
```

How the Code Works

1. **Terrain Generation Functions**
   - `generate_flat_terrain()`: Produces a perfectly flat surface ($Z = 0$).

- `generate_rolling_hills()` : Uses sinusoidal functions to imitate gently rolling hills.
- `generate_mountainous_terrain()` : Randomly places a few "peaks" using 2D Gaussian shapes, which yields a rugged, mountainous look.
- (Commented-out sections show how to integrate **Perlin Noise** or **Gaussian Random Fields** if you install additional libraries and uncomment the relevant code.)

2. **User Interaction**
   - The script prompts you to **select a terrain type** (1 = Flat, 2 = Rolling Hills, 3 = Mountainous).
   - It also prompts for a **resolution** (e.g., 100 produces a 100×100 grid).
   - The code ensures at least 10×10 to avoid trivial or error-prone grids.
3. **Plotting**
   - Matplotlib's `plot_surface` is used for a quick 3D surface plot. You can interactively rotate and zoom the figure.
4. **Saving Data**
   - Each point (X, Y, Z) is flattened into a pandas DataFrame and written to a CSV file: \text{Filename} = \text{"synthetic_<TerrainName>_<Resolution>x<Resolution>.csv"}.
   - For example, if you choose *Mountainous* with resolution 150, you might get `synthetic_Mountainous_150x150.csv`.
5. **Real-World-Like Data**
   - Although this data is not from an actual topographic survey, the shapes can mimic real-world conditions.
   - Adjust parameters (like the number of peaks, amplitude of hills, random noise strength, etc.) to create more or less extreme landscapes.

Tips & Modifications

1. **Scaling & Extent**
   - The `extent` parameter in each function is set to `1000` by default. You can think of it in meters or any consistent unit. Increase/decrease to simulate larger/smaller regions.

2. **Additional Terrain Types**
    - Feel free to add more complex or specialized terrain generation functions (e.g., plateaus, valleys, ridges, etc.).
3. **Resolution vs. File Size**
    - Higher resolutions (e.g., 1000 × 1000) will result in 1,000,000 grid points. Consider memory and performance implications.
4. **File Formats**
    - CSV is convenient for quick data inspection. For more advanced use, consider geospatial formats like GeoTIFF or NetCDF, especially if integrating with GIS software.
5. **Geostatistical Approaches**
    - If you want more physically or geologically accurate surfaces, you can leverage geostatistical libraries (e.g., `gstools`) to define variogram models and generate correlated random fields.
6. **Noise Libraries**
    - `pip install noise` if you want to use Perlin noise and uncomment the related code.

With this setup, you have a **self-contained** script that generates purely **synthetic topographic data** based on user choices, complete with adjustable resolution and multiple terrain styles—without needing any real-world dataset.

```
import numpy as np
import pandas as pd
from scipy.interpolate import Rbf
import matplotlib.pyplot as plt
from noise import pnoise2
from scipy.fft import fft2, ifft2, fftshift
from scipy.stats import norm
import gstools as gs  # Add gstools import
import plotly.graph_objects as go

# ------------------------------------------------------------------------
```

```
# STEP 1: Choose Topography Type and Grid Size
# ------------------------------------------------------------------------
print("\nDefine Grid Dimensions:")
grid_size = float(input("Enter the grid size in meters (e.g.,
1000 for 1km x 1km): "))
resolution = int(input("Enter the grid resolution (number of
points per side, e.g., 100): "))

topography_options = {
    '1': 'Hill',
    '2': 'Valley',
    '3': 'Plateau',
    '4': 'Mountain Range',
    '5': 'Depression'
}

print("\nSelect the type of topography to generate:")
for key, value in topography_options.items():
    print(f"{key}. {value}")

choice = input("Enter the number corresponding to your
choice: ")
selected_topography = topography_options.get(choice,
'Hill')  # Default to Hill if invalid input

# ------------------------------------------------------------------------
# STEP 2: Define Terrain Generation Parameters Based on
Selection
# ------------------------------------------------------------------------
def set_parameters(topography_type):
    if topography_type == 'Hill':
        return {
            'scale': 50.0,
            'octaves': 4,
            'persistence': 0.6,
            'lacunarity': 2.5,
            'grf_len_scale': 2.0
        }
    elif topography_type == 'Valley':
        return {
            'scale': 80.0,
            'octaves': 5,
```

```
            'persistence': 0.5,
            'lacunarity': 2.0,
            'grf_len_scale': 3.0
        }
    elif topography_type == 'Plateau':
        return {
            'scale': 100.0,
            'octaves': 3,
            'persistence': 0.4,
            'lacunarity': 2.2,
            'grf_len_scale': 1.5
        }
    elif topography_type == 'Mountain Range':
        return {
            'scale': 40.0,
            'octaves': 7,
            'persistence': 0.7,
            'lacunarity': 2.8,
            'grf_len_scale': 1.0
        }
    elif topography_type == 'Depression':
        return {
            'scale': 90.0,
            'octaves': 6,
            'persistence': 0.5,
            'lacunarity': 2.3,
            'grf_len_scale': 2.5
        }

params = set_parameters(selected_topography)

# ------------------------------------------------------------------------
# STEP 3: Define the grid
# ------------------------------------------------------------------------
num_points = resolution
x = np.linspace(0, grid_size, num_points)
y = np.linspace(0, grid_size, num_points)

# Create a mesh for evaluation
X_synth, Y_synth = np.meshgrid(x, y)
X_flat = X_synth.flatten()
Y_flat = Y_synth.flatten()
```

```
# ------------------------------------------------------------------------
# STEP 4: Generate base terrain
# ------------------------------------------------------------------------
# Generate initial terrain using combined Perlin noise and
GRF
X_perlin, Y_perlin, z_perlin = generate_perlin_terrain(
    num_points,
    params,
    x_range=(0, grid_size),
    y_range=(0, grid_size)
)

z_grf = generate_grf_terrain(
    num_points,
    x_range=(0, grid_size),
    y_range=(0, grid_size),
    len_scale=params['grf_len_scale']
)[2]  # Only take the Z values

# Combine Perlin and GRF with weights based on
topography type
if selected_topography in ['Mountain Range', 'Hill']:
    perlin_weight = 0.7
    grf_weight = 0.3
elif selected_topography in ['Valley', 'Depression']:
    perlin_weight = 0.4
    grf_weight = 0.6
else:  # Plateau
    perlin_weight = 0.5
    grf_weight = 0.5

z_base = (perlin_weight * z_perlin + grf_weight * z_grf)

# Add some random noise for additional variation
noise_strength = 0.05
z_synth_final = z_base + noise_strength *
np.random.randn(*z_base.shape)

# Scale the heights based on topography type
height_scales = {
    'Hill': 100,  # 100m high hills
```

```
    'Valley': -150,  # 150m deep valleys
    'Plateau': 500,  # 500m high plateau
    'Mountain Range': 2000,  # 2000m high mountains
    'Depression': -300  # 300m deep depression
}

z_synth_final = z_synth_final *
(height_scales[selected_topography] /
np.max(np.abs(z_synth_final)))

# ------------------------------------------------------------------------
# STEP 5: Create and save the synthetic dataset
# ------------------------------------------------------------------------
df_synth = pd.DataFrame({
    'X': X_flat,
    'Y': Y_flat,
    'Z (Synthetic)': z_synth_final.flatten()
})

# Save to CSV
df_synth.to_csv('synthetic_topography.csv', index=False)

# ------------------------------------------------------------------------
# STEP 6: Visualization
# ------------------------------------------------------------------------
fig = go.Figure(data=[
    go.Surface(
        x=X_synth,
        y=Y_synth,
        z=z_synth_final.reshape(X_synth.shape),
        colorscale='terrain'
    )
])

fig.update_layout(
    title=f'Synthetic {selected_topography} Topography',
    scene=dict(
        xaxis_title='X (meters)',
        yaxis_title='Y (meters)',
        zaxis_title='Elevation (meters)',
        camera=dict(
            eye=dict(x=1.5, y=1.5, z=1.2)
```

```
        )
    ),
    width=1000,
    height=800
)

fig.show()
```

Below is a **self-contained** Python script that does **not** rely on any external (existing) data. It generates a **synthetic topography** (Hill, Valley, Plateau, Mountain Range, or Depression) purely from mathematical/noise-based methods, combining Perlin noise and Gaussian Random Fields (GRF) for realistic variation. The script:

1. **Prompts the user** for:
    - **Grid size** in meters (e.g., 1000 for 1 km × 1 km).
    - **Resolution** (number of points in each dimension).
    - **Topography type** (Hill, Valley, Plateau, Mountain Range, Depression).
2. Generates **synthetic Z values** using:
    - **Perlin noise** for fractal-like surface details.
    - **Gaussian Random Fields** for broader spatial correlation.
3. **Combines** Perlin and GRF with type-dependent weights.
4. **Scales** the height range according to the chosen topography.
5. **Visualizes** the result in an interactive 3D plot (using Plotly).
6. **Saves** the resulting points (X, Y, Z) to `synthetic_topography.csv`.

> **Note:** This example requires installing `noise`, `plotly`, and `gstools`. You can do so via: `pip install noise plotly gstools`

---

python

```python
import numpy as np
import pandas as pd
```

```python
import matplotlib.pyplot as plt
from noise import pnoise2
import gstools as gs
import plotly.graph_objects as go

# ----------------------------------------
# Generate Perlin Noise terrain
# ----------------------------------------
def generate_perlin_terrain(num_points, params, x_range=(0, 10), y_range=(0, 10)):
    """
    Generates a 2D Perlin noise field over [x_range, y_range], normalized to [-1, 1].

    :param num_points: Number of points in each dimension (int)
    :param params: Dictionary containing noise parameters (scale, octaves, etc.)
    :param x_range: Tuple specifying the x-min and x-max
    :param y_range: Tuple specifying the y-min and y-max
    :return: (X, Y, Z) where X, Y are the meshgrid, Z is normalized in [-1, 1]
    """
    scale = params['scale']
    octaves = params['octaves']
    persistence = params['persistence']
    lacunarity = params['lacunarity']

    # Create a grid
    x = np.linspace(x_range[0], x_range[1], num_points)
    y = np.linspace(y_range[0], y_range[1], num_points)
    X, Y = np.meshgrid(x, y)

    # Generate Perlin noise
    Z = np.zeros((num_points, num_points))
    for i in range(num_points):
        for j in range(num_points):
            # pnoise2 expects float inputs for x, y
            Z[i, j] = pnoise2(
                X[i, j] / scale,
                Y[i, j] / scale,
                octaves=octaves,
                persistence=persistence,
                lacunarity=lacunarity,
                repeatx=999999,   # Large repeats so we don't tile quickly
                repeaty=999999,
```

```
                base=42
            )

    # Perlin noise is typically in [-1, 1] but let's verify/normalize to exactly
[-1, 1]
    z_min, z_max = Z.min(), Z.max()
    if z_max - z_min < 1e-8:
        # Edge case: all noise is the same
        Z_norm = np.zeros_like(Z)
    else:
        Z_norm = 2.0 * (Z - z_min) / (z_max - z_min) - 1.0  # scale to [-1, 1]

    return X, Y, Z_norm

# ----------------------------------------
# Generate Gaussian Random Field terrain
# ----------------------------------------
def generate_grf_terrain(num_points, x_range=(0, 10), y_range=(0, 10),
len_scale=1.0):
    """
    Generates a 2D Gaussian Random Field using gstools, normalized to [-1, 1].

    :param num_points: Number of points in each dimension
    :param x_range: Tuple specifying the x-min and x-max
    :param y_range: Tuple specifying the y-min and y-max
    :param len_scale: Correlation length scale for the GRF
    :return: (X, Y, Z) with Z in [-1, 1]
    """
    x = np.linspace(x_range[0], x_range[1], num_points)
    y = np.linspace(y_range[0], y_range[1], num_points)
    X, Y = np.meshgrid(x, y)

    # Define a variogram model; we choose Exponential for some roughness
    model = gs.Exponential(dim=2, var=1.0, len_scale=len_scale)
    srf = gs.SRF(model, seed=42)  # for reproducibility

    # Generate the GRF on a structured grid
    Z = srf.structured([x, y])  # shape = (ny, nx) => (num_points, num_points)

    # Normalize Z to [-1, 1]
    z_min, z_max = Z.min(), Z.max()
    if z_max - z_min < 1e-8:
```

```
        Z_norm = np.zeros_like(Z)
    else:
        Z_norm = 2.0 * (Z - z_min) / (z_max - z_min) - 1.0

    return X, Y, Z_norm

# ----------------------------------------------------------------------
# STEP 1: Choose Topography Type and Grid Size
# ----------------------------------------------------------------------
print("\nDefine Grid Dimensions:")
grid_size = float(input("Enter the grid size in meters (e.g., 1000 for 1km x 1km):
"))
resolution = int(input("Enter the grid resolution (number of points per side, e.g.,
100): "))

topography_options = {
    '1': 'Hill',
    '2': 'Valley',
    '3': 'Plateau',
    '4': 'Mountain Range',
    '5': 'Depression'
}

print("\nSelect the type of topography to generate:")
for key, value in topography_options.items():
    print(f"{key}. {value}")

choice = input("Enter the number corresponding to your choice: ")
selected_topography = topography_options.get(choice, 'Hill')  # Default to 'Hill' if
invalid input

# ----------------------------------------------------------------------
# STEP 2: Define Terrain Generation Parameters Based on Selection
# ----------------------------------------------------------------------
def set_parameters(topography_type):
    """
    Returns a dictionary of parameters for Perlin noise (scale, octaves, etc.)
    and a length scale for the Gaussian Random Field (GRF).
    """
    if topography_type == 'Hill':
        return {
            'scale': 50.0,
```

```
            'octaves': 4,
            'persistence': 0.6,
            'lacunarity': 2.5,
            'grf_len_scale': 2.0
        }
    elif topography_type == 'Valley':
        return {
            'scale': 80.0,
            'octaves': 5,
            'persistence': 0.5,
            'lacunarity': 2.0,
            'grf_len_scale': 3.0
        }
    elif topography_type == 'Plateau':
        return {
            'scale': 100.0,
            'octaves': 3,
            'persistence': 0.4,
            'lacunarity': 2.2,
            'grf_len_scale': 1.5
        }
    elif topography_type == 'Mountain Range':
        return {
            'scale': 40.0,
            'octaves': 7,
            'persistence': 0.7,
            'lacunarity': 2.8,
            'grf_len_scale': 1.0
        }
    elif topography_type == 'Depression':
        return {
            'scale': 90.0,
            'octaves': 6,
            'persistence': 0.5,
            'lacunarity': 2.3,
            'grf_len_scale': 2.5
        }

params = set_parameters(selected_topography)

# ----------------------------------------------------------------------
# STEP 3: Define the grid
```

```
# ----------------------------------------------------------------------
num_points = resolution
x = np.linspace(0, grid_size, num_points)
y = np.linspace(0, grid_size, num_points)

# Create a mesh for evaluation
X_synth, Y_synth = np.meshgrid(x, y)
X_flat = X_synth.flatten()
Y_flat = Y_synth.flatten()

# ----------------------------------------------------------------------
# STEP 4: Generate base terrain
# ----------------------------------------------------------------------
# Generate Perlin noise component
X_perlin, Y_perlin, z_perlin = generate_perlin_terrain(
    num_points,
    params,
    x_range=(0, grid_size),
    y_range=(0, grid_size)
)

# Generate GRF component
z_grf = generate_grf_terrain(
    num_points,
    x_range=(0, grid_size),
    y_range=(0, grid_size),
    len_scale=params['grf_len_scale']
)[2]  # we only need the Z values

# Combine Perlin and GRF with weights depending on topography
if selected_topography in ['Mountain Range', 'Hill']:
    perlin_weight = 0.7
    grf_weight = 0.3
elif selected_topography in ['Valley', 'Depression']:
    perlin_weight = 0.4
    grf_weight = 0.6
else:  # Plateau
    perlin_weight = 0.5
    grf_weight = 0.5

z_base = perlin_weight * z_perlin + grf_weight * z_grf
```

```
# Add small-scale random noise for additional variation
noise_strength = 0.05
z_synth_final = z_base + noise_strength * np.random.randn(*z_base.shape)

# Scale the heights to a desired range (meters)
height_scales = {
    'Hill': 100,           # ~100m high hills
    'Valley': -150,        # ~150m deep
    'Plateau': 500,        # ~500m elevated plateau
    'Mountain Range': 2000,# ~2000m tall mountains
    'Depression': -300     # ~300m below surrounding region
}

# Safeguard against division by zero if z_synth_final is near-constant
max_abs_val = np.max(np.abs(z_synth_final))
if max_abs_val < 1e-8:
    # If the variation is too small, just keep it zero
    z_synth_final[:] = 0
else:
    z_synth_final *= (height_scales[selected_topography] / max_abs_val)

# ---------------------------------------------------------------------
# STEP 5: Create and save the synthetic dataset
# ---------------------------------------------------------------------
df_synth = pd.DataFrame({
    'X': X_flat,
    'Y': Y_flat,
    'Z (Synthetic)': z_synth_final.flatten()
})

output_filename = 'synthetic_topography.csv'
df_synth.to_csv(output_filename, index=False)
print(f"\nSynthetic {selected_topography} topography saved to '{output_filename}'")

# ---------------------------------------------------------------------
# STEP 6: Visualization with Plotly
# ---------------------------------------------------------------------
fig = go.Figure(data=[
    go.Surface(
        x=X_synth,
        y=Y_synth,
        z=z_synth_final.reshape(X_synth.shape),
```

```
        colorscale='terrain'
    )
])

fig.update_layout(
    title=f'Synthetic {selected_topography} Topography',
    scene=dict(
        xaxis_title='X (meters)',
        yaxis_title='Y (meters)',
        zaxis_title='Elevation (meters)',
        camera=dict(
            eye=dict(x=1.5, y=1.5, z=1.2)
        )
    ),
    width=1000,
    height=800
)

fig.show()
```

How It Works

1. **User Input**
   - **Grid size (in meters)**: e.g., `1000` for a 1 km × 1 km region.
   - **Resolution**: Number of points in each dimension. If `resolution = 100`, you get a 100 × 100 = 10,000-point mesh.
   - **Topography Type**: Choose from *Hill*, *Valley*, *Plateau*, *Mountain Range*, or *Depression*.
2. **Perlin Noise (`generate_perlin_terrain`)**
   - We produce fractal noise using the `noise` library.
   - The function normalizes the resulting 2D array to the range $[-1, 1]$.
3. **Gaussian Random Field (`generate_grf_terrain`)**
   - Uses `gstools` to generate a random field based on an **Exponential** variogram model.
   - Also normalized to $[-1, 1]$.

- `len_scale` controls how quickly the terrain correlates spatially (smaller = more rugged, larger = smoother).

4. **Combining Perlin + GRF**
   - The code blends them with different weights (70%-30%, 40%-60%, etc.) depending on the selected topography type.
   - You can tweak these weights to get more or less fractal vs. correlated terrain.
5. **Adding Noise & Scaling**
   - A small random noise term ($\pm 5\%$) is added to break up uniform patterns.
   - The final `z_synth_final` is then **scaled** to a typical height range:
     - **Hill**: ± 100 meters
     - **Valley**: ± 150 meters (negative to represent going below the surrounding region)
     - **Plateau**: up to 500 meters high
     - **Mountain Range**: up to 2000 meters
     - **Depression**: down to -300 meters
6. **Saving & Visualizing**
   - Coordinates and elevations are flattened into columns: (X, Y, Z).
   - Saved in **CSV** format: `synthetic_topography.csv`.
   - Plotly's **3D Surface** (`go.Surface`) provides an interactive 3D plot in a web browser.

Customization Tips

- **Adjust Variogram**: In `generate_grf_terrain()`, switch from `Exponential` to other models (`Gaussian`, `Spherical`, etc.) or alter `var` and `len_scale` for different correlation lengths.
- **Change Weights**: Modify how Perlin vs. GRF is blended to get different "textures."
- **Height Range**: Tweak the `height_scales` dictionary to produce steeper or more gentle relief.

- **Performance**: If you need bigger grids (e.g., 1000×1000), consider your system's memory constraints.

With this code, you can **quickly generate** plausible synthetic topography **without** using any real-world data, yet still produce surfaces that exhibit features reminiscent of natural terrains.

Update the generation of plotly to be accroding to topograph rather than just points

Below is an **updated Python script** that uses **different color scales** and **camera angles** in Plotly, depending on which topography type the user selects. This way, the *visual style* of the 3D surface better reflects the chosen *Hill, Valley, Plateau, Mountain Range,* or *Depression* topography—rather than using a single default color scale.

python

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from noise import pnoise2
import gstools as gs
import plotly.graph_objects as go

# ----------------------------------------
# Generate Perlin Noise terrain
# ----------------------------------------
def generate_perlin_terrain(num_points, params, x_range=(0, 10), y_range=(0, 10)):
    """
    Generates a 2D Perlin noise field over [x_range, y_range], normalized to [-1, 1].

    :param num_points: Number of points in each dimension (int)
    :param params: Dictionary containing noise parameters (scale, octaves, etc.)
    :param x_range: Tuple specifying the x-min and x-max
    :param y_range: Tuple specifying the y-min and y-max
    :return: (X, Y, Z) where X, Y are the meshgrid, Z is normalized in [-1, 1]
    """
    scale = params['scale']
    octaves = params['octaves']
```

```
    persistence = params['persistence']
    lacunarity = params['lacunarity']

    # Create a grid
    x = np.linspace(x_range[0], x_range[1], num_points)
    y = np.linspace(y_range[0], y_range[1], num_points)
    X, Y = np.meshgrid(x, y)

    # Generate Perlin noise
    Z = np.zeros((num_points, num_points))
    for i in range(num_points):
        for j in range(num_points):
            Z[i, j] = pnoise2(
                X[i, j] / scale,
                Y[i, j] / scale,
                octaves=octaves,
                persistence=persistence,
                lacunarity=lacunarity,
                repeatx=999999,
                repeaty=999999,
                base=42
            )

    # Normalize to [-1, 1]
    z_min, z_max = Z.min(), Z.max()
    if z_max - z_min < 1e-8:
        Z_norm = np.zeros_like(Z)
    else:
        Z_norm = 2.0 * (Z - z_min) / (z_max - z_min) - 1.0

    return X, Y, Z_norm

# ----------------------------------------
# Generate Gaussian Random Field terrain
# ----------------------------------------
def generate_grf_terrain(num_points, x_range=(0, 10), y_range=(0, 10),
len_scale=1.0):
    """
    Generates a 2D Gaussian Random Field using gstools, normalized to [-1, 1].

    :param num_points: Number of points in each dimension
    :param x_range: Tuple specifying the x-min and x-max
```

```
    :param y_range: Tuple specifying the y-min and y-max
    :param len_scale: Correlation length scale for the GRF
    :return: (X, Y, Z) with Z in [-1, 1]
    """
    x = np.linspace(x_range[0], x_range[1], num_points)
    y = np.linspace(y_range[0], y_range[1], num_points)
    X, Y = np.meshgrid(x, y)

    # Define a variogram model (Exponential as an example)
    model = gs.Exponential(dim=2, var=1.0, len_scale=len_scale)
    srf = gs.SRF(model, seed=42)

    # Generate the GRF on the structured grid
    Z = srf.structured([x, y])  # shape = (ny, nx)

    # Normalize Z to [-1, 1]
    z_min, z_max = Z.min(), Z.max()
    if z_max - z_min < 1e-8:
        Z_norm = np.zeros_like(Z)
    else:
        Z_norm = 2.0 * (Z - z_min) / (z_max - z_min) - 1.0

    return X, Y, Z_norm

# ---------------------------------------------------------------------
# STEP 1: Choose Topography Type and Grid Size
# ---------------------------------------------------------------------
print("\nDefine Grid Dimensions:")
grid_size = float(input("Enter the grid size in meters (e.g., 1000 for 1km x 1km):
"))
resolution = int(input("Enter the grid resolution (number of points per side, e.g.,
100): "))

topography_options = {
    '1': 'Hill',
    '2': 'Valley',
    '3': 'Plateau',
    '4': 'Mountain Range',
    '5': 'Depression'
}

print("\nSelect the type of topography to generate:")
```

```python
for key, value in topography_options.items():
    print(f"{key}. {value}")

choice = input("Enter the number corresponding to your choice: ")
selected_topography = topography_options.get(choice, 'Hill')  # Default to 'Hill' if invalid input

# ---------------------------------------------------------------------
# STEP 2: Define Terrain Generation Parameters Based on Selection
# ---------------------------------------------------------------------
def set_parameters(topography_type):
    """
    Returns a dictionary of parameters for Perlin noise (scale, octaves, etc.)
    and a length scale for the Gaussian Random Field (GRF).
    """
    if topography_type == 'Hill':
        return {
            'scale': 50.0,
            'octaves': 4,
            'persistence': 0.6,
            'lacunarity': 2.5,
            'grf_len_scale': 2.0
        }
    elif topography_type == 'Valley':
        return {
            'scale': 80.0,
            'octaves': 5,
            'persistence': 0.5,
            'lacunarity': 2.0,
            'grf_len_scale': 3.0
        }
    elif topography_type == 'Plateau':
        return {
            'scale': 100.0,
            'octaves': 3,
            'persistence': 0.4,
            'lacunarity': 2.2,
            'grf_len_scale': 1.5
        }
    elif topography_type == 'Mountain Range':
        return {
            'scale': 40.0,
```

```
            'octaves': 7,
            'persistence': 0.7,
            'lacunarity': 2.8,
            'grf_len_scale': 1.0
        }
    elif topography_type == 'Depression':
        return {
            'scale': 90.0,
            'octaves': 6,
            'persistence': 0.5,
            'lacunarity': 2.3,
            'grf_len_scale': 2.5
        }

params = set_parameters(selected_topography)

# ----------------------------------------------------------------------
# STEP 3: Define the grid
# ----------------------------------------------------------------------
num_points = resolution
x = np.linspace(0, grid_size, num_points)
y = np.linspace(0, grid_size, num_points)

# Create a mesh for evaluation
X_synth, Y_synth = np.meshgrid(x, y)
X_flat = X_synth.flatten()
Y_flat = Y_synth.flatten()

# ----------------------------------------------------------------------
# STEP 4: Generate base terrain
# ----------------------------------------------------------------------
# Perlin noise component
X_perlin, Y_perlin, z_perlin = generate_perlin_terrain(
    num_points,
    params,
    x_range=(0, grid_size),
    y_range=(0, grid_size)
)

# GRF component
z_grf = generate_grf_terrain(
    num_points,
```

```
    x_range=(0, grid_size),
    y_range=(0, grid_size),
    len_scale=params['grf_len_scale']
)[2]  # Only the Z values

# Blend Perlin and GRF by topography
if selected_topography in ['Mountain Range', 'Hill']:
    perlin_weight = 0.7
    grf_weight = 0.3
elif selected_topography in ['Valley', 'Depression']:
    perlin_weight = 0.4
    grf_weight = 0.6
else:  # Plateau
    perlin_weight = 0.5
    grf_weight = 0.5

z_base = perlin_weight * z_perlin + grf_weight * z_grf

# Add small random noise for detail
noise_strength = 0.05
z_synth_final = z_base + noise_strength * np.random.randn(*z_base.shape)

# Scale the heights
height_scales = {
    'Hill': 100,            # ~100m high hills
    'Valley': -150,         # ~150m deep valley
    'Plateau': 500,         # ~500m high plateau
    'Mountain Range': 2000, # ~2000m high mountains
    'Depression': -300      # ~300m deep depression
}

max_abs_val = np.max(np.abs(z_synth_final))
if max_abs_val < 1e-8:
    z_synth_final[:] = 0
else:
    z_synth_final *= (height_scales[selected_topography] / max_abs_val)

# ---------------------------------------------------------------------
# STEP 5: Create and save the synthetic dataset
# ---------------------------------------------------------------------
df_synth = pd.DataFrame({
    'X': X_flat,
```

```
    'Y': Y_flat,
    'Z (Synthetic)': z_synth_final.flatten()
})

output_filename = 'synthetic_topography.csv'
df_synth.to_csv(output_filename, index=False)
print(f"\nSynthetic {selected_topography} topography saved to '{output_filename}'")

# ----------------------------------------------------------------------
# STEP 6: Visualization with Plotly
#      - Custom color scales / camera angles for each topography
# ----------------------------------------------------------------------
# Define custom color scales (Plotly built-ins or named ones)
color_scales = {
    'Hill': 'Greens',
    'Valley': 'Blues',
    'Plateau': 'Earth',
    'Mountain Range': 'Picnic',
    'Depression': 'Viridis'
}
chosen_colorscale = color_scales.get(selected_topography, 'terrain')  # fallback

# Define custom camera positions (just examples)
camera_settings = {
    'Hill': dict(eye=dict(x=1.5, y=1.5, z=1.2)),
    'Valley': dict(eye=dict(x=-1.2, y=1.2, z=1.4)),
    'Plateau': dict(eye=dict(x=2.0, y=2.0, z=2.0)),
    'Mountain Range': dict(eye=dict(x=1.8, y=1.8, z=1.8)),
    'Depression': dict(eye=dict(x=1.0, y=-1.4, z=1.2))
}
chosen_camera = camera_settings.get(selected_topography, dict(x=1.5, y=1.5, z=1.2))

fig = go.Figure(data=[
    go.Surface(
        x=X_synth,
        y=Y_synth,
        z=z_synth_final.reshape(X_synth.shape),
        colorscale=chosen_colorscale,
        contours={
            "z": {
                "show": True,
                "start": np.min(z_synth_final),
```

```
                "end": np.max(z_synth_final),
                "size": (np.max(z_synth_final) - np.min(z_synth_final)) / 8,
                "color":"black"
            }
        }
    )
])

fig.update_layout(
    title=f'Synthetic {selected_topography} Topography',
    scene=dict(
        xaxis_title='X (meters)',
        yaxis_title='Y (meters)',
        zaxis_title='Elevation (meters)',
        camera=chosen_camera
    ),
    width=1000,
    height=800
)

fig.show()
```

What Changed?

1. **Color Scales**
   - We define a dictionary `color_scales` keyed by topography name. For instance, “Hill” uses Plotly’s `'Greens'`, “Valley” uses `'Blues'`, etc.
   - During plotting, we fetch the color scale:

     ```python
     chosen_colorscale = color_scales.get(selected_topography, 'terrain')
     ```

2. **Camera Angles**
   - Another dictionary `camera_settings` maps each topography to a different 3D camera position (defined by `eye = dict(x=..., y=..., z=...)`).
   - Adjust these values for more dramatic (or subtle) views:

```python
chosen_camera = camera_settings.get(selected_topography, dict(x=1.5, y=1.5, z=1.2))
```

3. **Contour Lines**
   - We added a small snippet in the `go.Surface` definition:

```python
contours={
    "z": {
        "show": True,
        "start": np.min(z_synth_final),
        "end": np.max(z_synth_final),
        "size": (np.max(z_synth_final) - np.min(z_synth_final)) / 8,
        "color":"black"
    }
}
```

   - This draws **horizontal contour lines** on the 3D surface, giving it more of a "topographic map" feel.
4. **Everything Else**
   - The rest of the steps (Perlin + GRF generation, scaling, saving CSV) remain the same. This script still produces a fully synthetic topography *without* needing real-world data.

Further Tweaks

- **Refine Color Scale**
  - Plotly has many built-in color scales (e.g., `'Viridis'`, `'Plasma'`, `'RdBu'`, `'Cividis'`, `'Portland'`, etc.). Choose ones that best represent each landform.
- **Adjust Camera**
  - A more "top-down" view for Plateaus or a more "angled" view for Mountain Ranges can be set by modifying the `eye` parameters.

- **Labeling**
  - You can dynamically change axis labels or add subtitles referencing altitude ranges, e.g., “0–2000 m” or “-300 to 0 m” for a depression.
- **Contours**
  - Currently, we sample 8 contour intervals. You can increase or decrease `size` or specify an exact list of contour levels.

With these changes, the **Plotly visualization** now *adapts* to each topography type—offering **unique color palettes** and **camera perspectives** for a more intuitive and engaging presentation.