Contents

Maryamsadat Hosseini	S1111262

Reason for Topic Selection & AI used

I chose to work with Python and notebooks because they were new to me and I wanted to challenge myself. Since I had already worked with web maps, I decided to explore viewshed analysis instead. I only had a basic understanding of it, and I wanted to develop a deeper knowledge of the topic. For this assignment I used https://chatgpt.com/ for assissting and better learning.

Project Overview

This notebook was developed in the ArcGIS Pro Notebook environment, which is based on Jupyter Notebook and integrates ArcPy for geospatial automation.

For this course project, I developed an automated viewshed analysis notebook using ArcGIS Pro and ArcPy. The main objective was to build a structured, reusable, and flexible workflow that allows users to perform visibility analysis under different scenarios without manually changing the core code.

In this notebook, the workflow starts by preparing the working environment and allowing the user to select a Digital Elevation Model (DEM). Based on the spatial extent of the DEM, a fishnet grid is automatically generated. From this grid, observer points are created, and the observer height is defined as a user-controlled parameter. This makes the analysis adaptable to different study areas and conditions.

To improve flexibility and performance, I included optional steps such as DEM resampling, which can be used to reduce computational time, and creating a smaller subset of observer points for testing purposes. These steps are not mandatory but help optimize the workflow when working with high-resolution data or large datasets.

The core of the notebook is the Viewshed2 analysis using the frequency option. This allows the model to calculate how many observer locations can see each raster cell, providing a more informative visibility output compared to a simple binary viewshed.

In addition, the notebook supports multi-height input, enabling the user to automatically run the analysis for several observer heights in one execution. Each scenario is saved separately, and basic visibility statistics are calculated to help interpret and compare the results.

Overall, this project demonstrates how a spatial analysis workflow can be automated, parameterized, and structured in a way that supports reuse, scenario comparison, and practical decision-making applications.

Project purpose

The main objective of this project was to develop an automated and structured viewshed analysis workflow using ArcPy in ArcGIS Pro. Through this notebook, I aimed to reduce manual geoprocessing steps and create a reusable tool that allows multi-scenario visibility analysis based on different observer heights. Although this project is a small academic example, it demonstrates the core logic and structure that can be applied to much larger and more complex spatial analysis projects in real-world urban planning and decision-support systems.

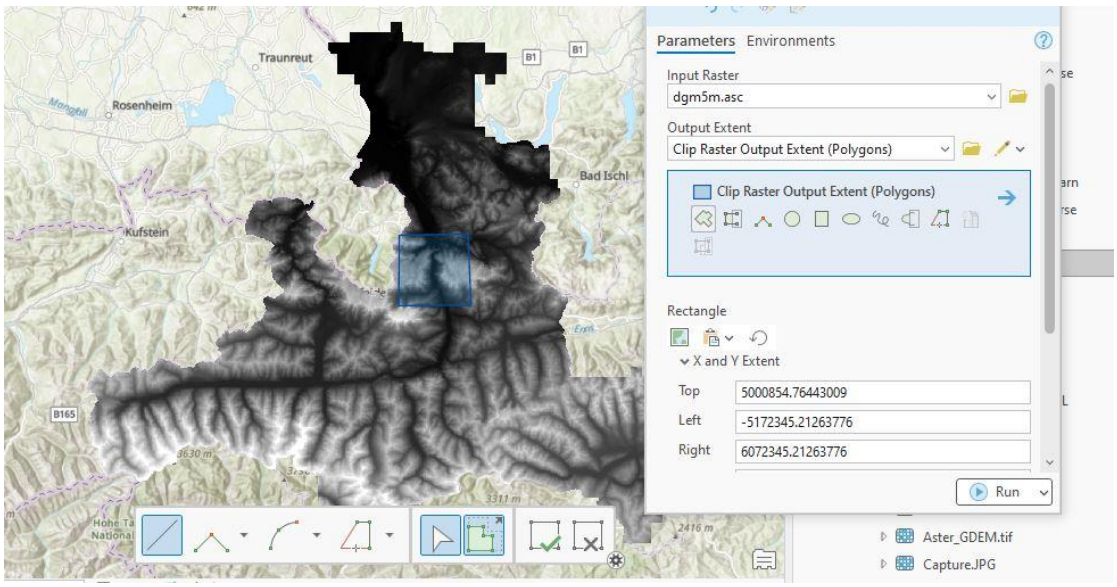
Practical steps

Section 1: Data prepration

Step 1 – Preparing the DEM

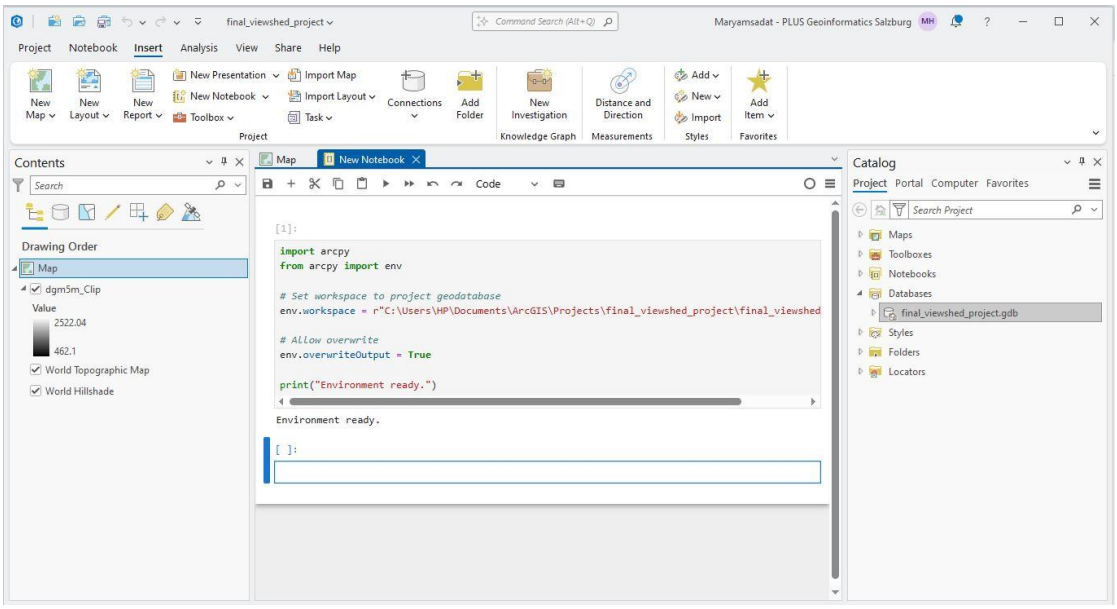
In the first step, I prepared the elevation data. The original DEM (5 meter resolution) was very large (around 2.8 GB), so I clipped it to a smaller study area for development and testing purposes. I stored the clipped DEM inside the project geodatabase.

*I reduced the size significantly because the ArcGIS Pro crashed so many times.



Step 2 – Setting the ArcPy Environment

In the second step, I created an ArcGIS Notebook and set up the ArcPy environment. I defined the workspace (project geodatabase)

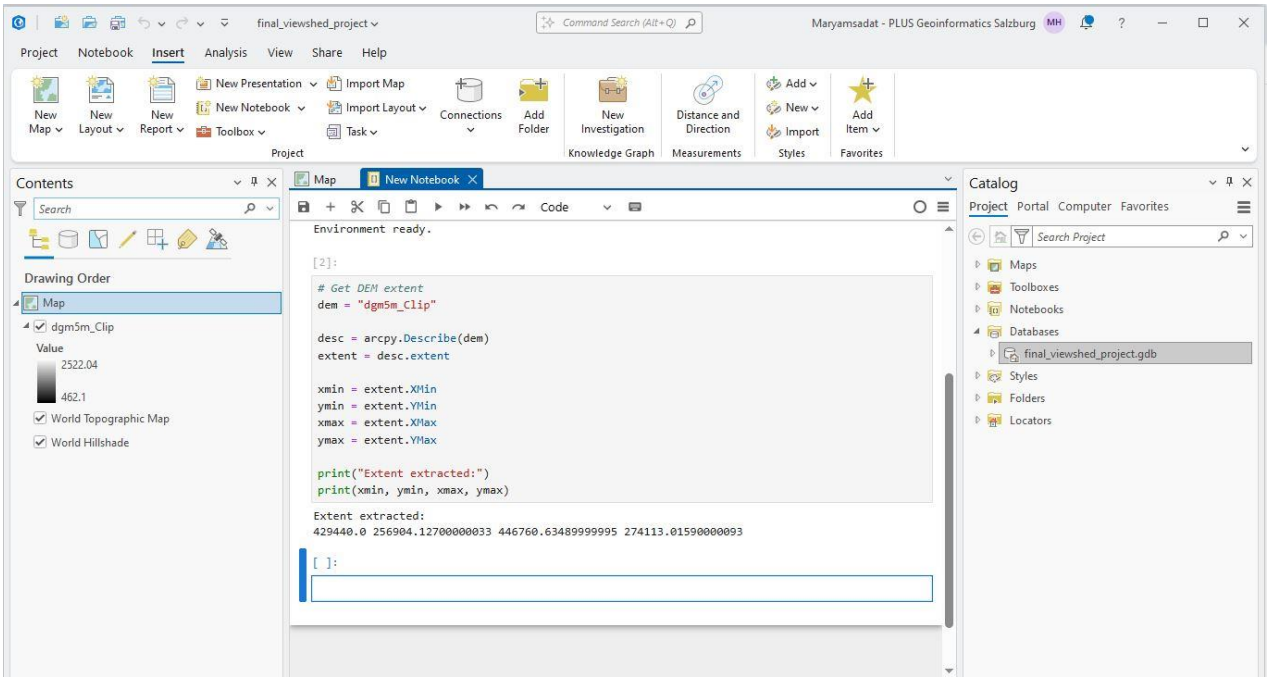


Step 3 – Extracting the DEM Extent

Next, I extracted the spatial extent (Xmin, Ymin, Xmax, Ymax) of the DEM.

This is important because:

- The grid must exactly match the DEM area
- The analysis should not exceed the study region.
- This makes the workflow fully automatic and reusable.

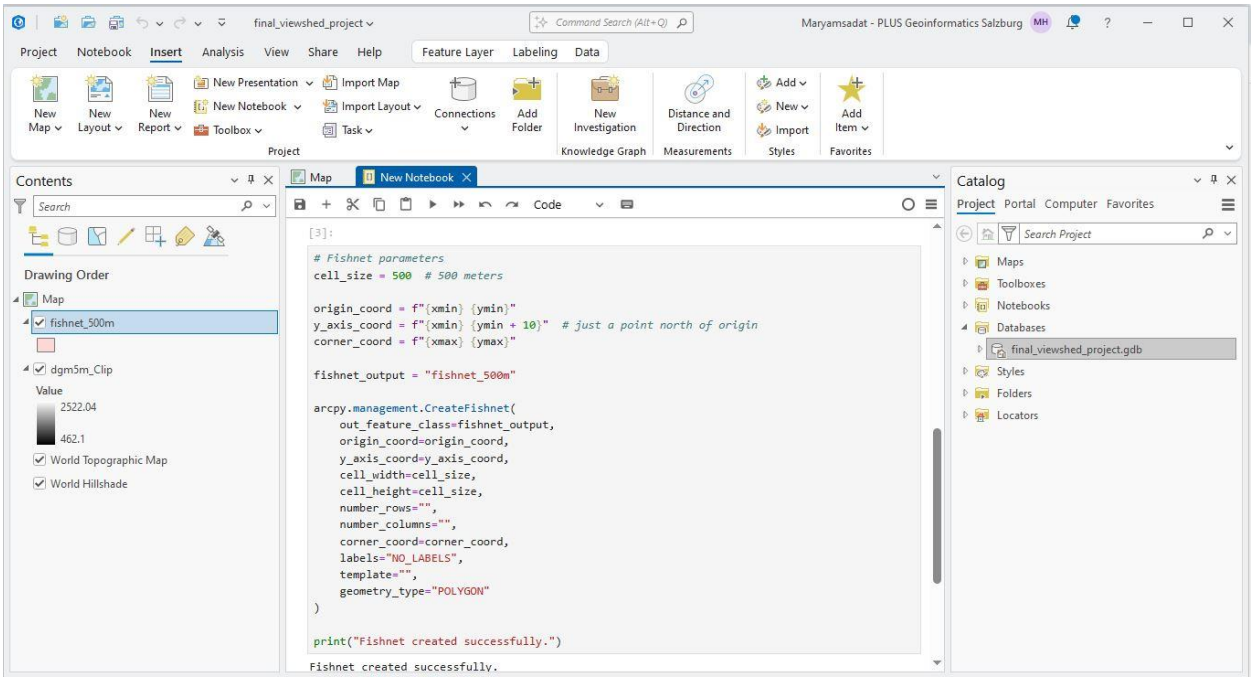


Step 4 – Creating a Fishnet Grid (500m)

Then, I created a regular fishnet grid with 500-meter spacing over the DEM extent. Each grid cell is 500 × 500 meters.

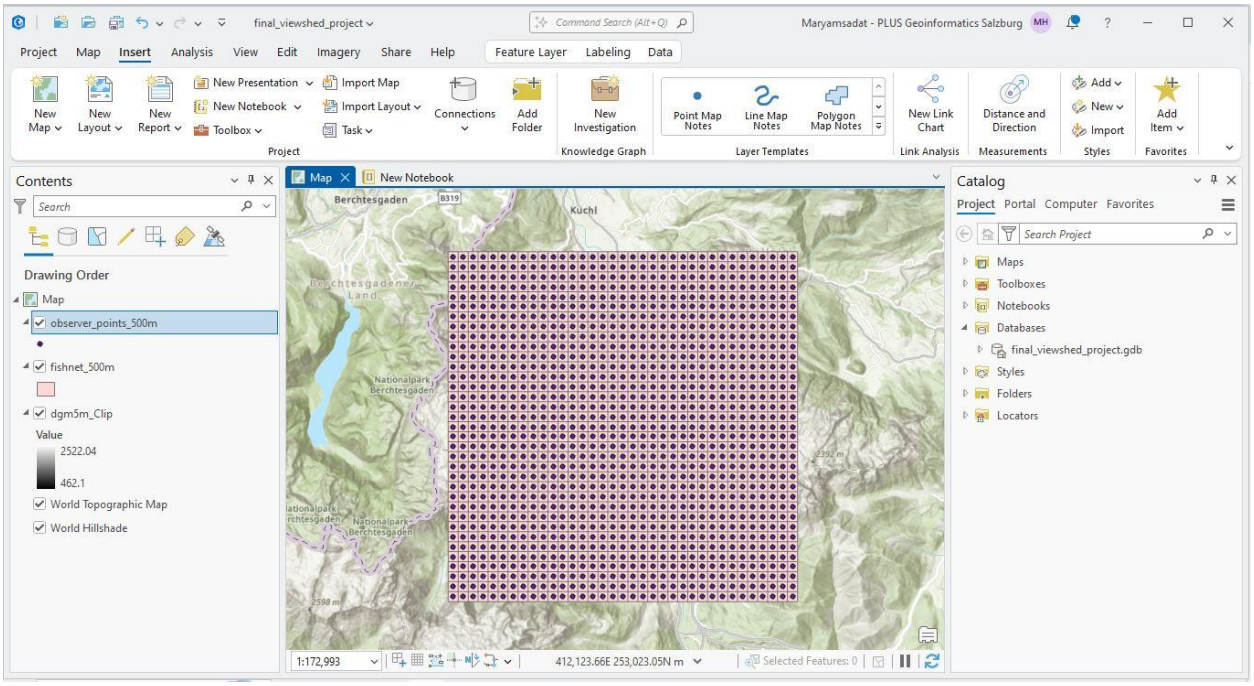
The purpose of this grid is to:

- Automatically generate systematic observation locations
- Avoid manual point selection



Step 5 – Generating Observer Points

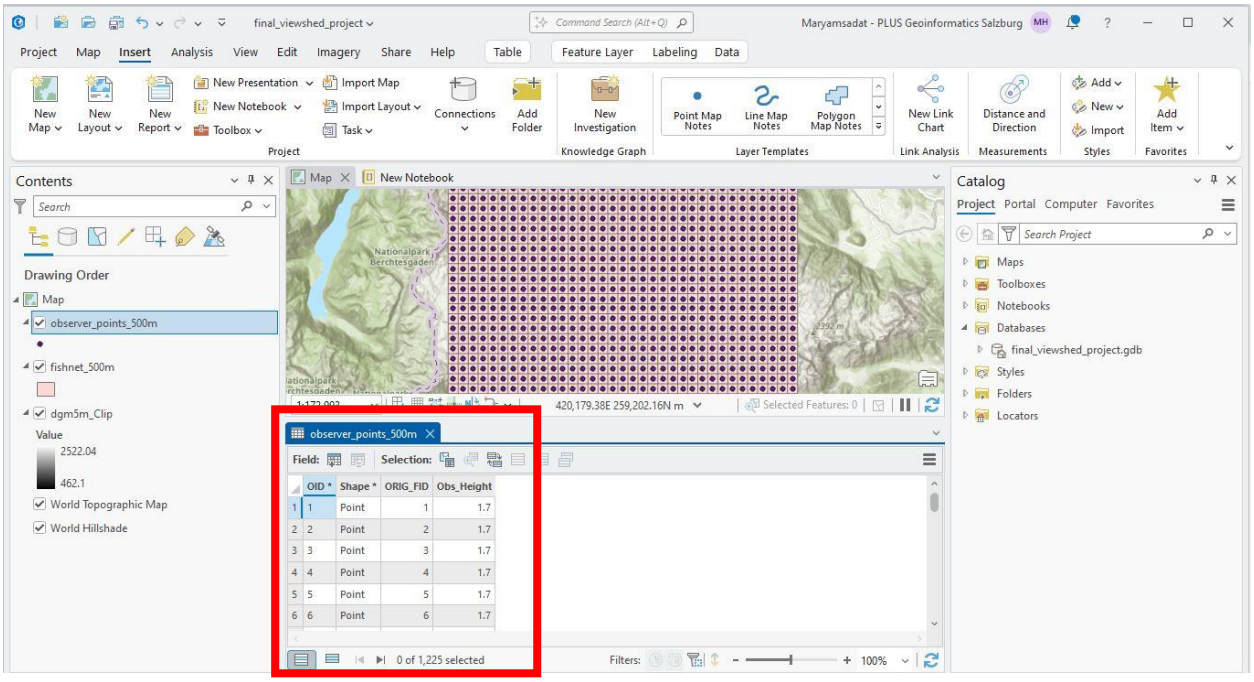
After creating the grid polygons, I converted each cell into a centroid point. These centroid points represent potential observer locations. In total, 1225 observer points were generated. I wanted to ensure that I cover the whole study area.



Step 6 – Assigning Observer Height

In this step, I added an **observer height** to all generated points. Since visibility analysis depends not only on terrain elevation but also on the height of the observer, it is necessary to define this value. I created a new field called **Obs_Height** and assigned a constant value of **1.7 meters**, representing the average human eye level. I wanted to make sure that the viewshed analysis reflects realistic observation conditions.

*At this stage, the observer points are fully prepared for **visibility analysis**.



During the first attempt, I created a fishnet grid with 500-meter spacing, which generated 1225 observer points. However, when I ran the viewshed analysis, the processing time became extremely long. So I decided to adjust the grid spacing to 1000 meters. This reduced the number of observer points to 324, which is more appropriate for testing and development. The 324 points still provide full spatial coverage of the study area while significantly improving performance and reducing processing time. This adjustment allows the workflow to remain realistic, reusable, and computationally manageable.

Since the viewshed analysis was taking too long with the 5-meter DEM, I resampled the elevation data to 20 meters. This significantly reduced the number of raster cells and improved processing performance while still maintaining sufficient detail for development and testing purposes.

Step 7- Viewshed completion

[6]:

```
import arcpy

# Total cell count
total_cells = 0
visible_cells = 0

with arcpy.da.SearchCursor("out_viewshed", ["Value", "Count"]) as cursor:
    for row in cursor:
        value = row[0]
        count = row[1]
        total_cells += count
        if value > 0:
            visible_cells += count

percentage_visible = (visible_cells / total_cells) * 100

print(f"Visible area percentage: {percentage_visible:.2f}%")
```

```
Visible area percentage: 100.00%
```

[7]:

```
high_cells = 0

with arcpy.da.SearchCursor("out_viewshed", ["Value", "Count"]) as cursor:
    for row in cursor:
        value = row[0]
        count = row[1]
        if value >= 10:
            high_cells += count

high_percentage = (high_cells / total_cells) * 100

print(f"High visibility area percentage: {high_percentage:.2f}%")
```

```
High visibility area percentage: 14.67%
```

The analysis shows that while 100% of the study area is visible from at least one observer, only 14.67% of the area is visible from ten or more observer locations. This indicates that although overall terrain coverage is complete, high visibility intensity is spatially concentrated in specific areas.

Section2- parameterization

Step 8

In this section, I transformed three key components of the workflow into **user-defined parameters**: the DEM input, the grid spacing, and the observer height. Instead of using fixed values in the script, I implemented dynamic input prompts that allow users to define these

parameters interactively. This makes the workflow flexible, reusable, and adaptable to different study areas and visibility scenarios without modifying the core code.

For example dor observer height, I implemented a dynamic observer height parameter with input validation. Instead of using a fixed value (e.g., 1.7 meters), the user can now enter any positive numeric height. The script also checks for invalid or negative values to prevent errors and ensure realistic input.

This modification improves flexibility and robustness of the workflow, making the notebook more reusable and user-friendly. I replaced a hard-coded value with a validated user-defined parameter to improve flexibility and reusability.

```
# --- Get observer height safely from user ---
while True:
    try:
        observer_height = float(input("Enter observer height in meters (e.g., 1.7, 10, 25): "
        if observer_height <= 0:
            print("Height must be a positive number.")
            continue
        break
    except ValueError:
        print("Please enter a valid numeric value.")

print(f"Observer height set to: {observer_height} meters")

# --- Add field if not exists ---
fields = [f.name for f in arcpy.ListFields("observer_points_1000m")]

if "Obs_Height" not in fields:
    arcpy.management.AddField(
        in_table="observer_points_1000m",
        field_name="Obs_Height",
        field_type="DOUBLE"
    )
    print("Obs_Height field created.")
else:
    print("Obs_Height field already exists.")
```

```
else:
    print("Obs_Height field already exists.")

# --- Update height value ---
arcpy.management.CalculateField(
    in_table="observer_points_1000m",
    field="Obs_Height",
    expression=str(observer_height),
    expression_type="PYTHON3"
)

print("Observer height successfully updated.")
```

```
Enter observer height in meters (e.g., 1.7, 10, 25):  -5
Height must be a positive number.
Enter observer height in meters (e.g., 1.7, 10, 25):  2
Observer height set to: 2.0 meters
Obs_Height field already exists.
Observer height successfully updated.
```

```
# --- Create dynamic fishnet name ---
fishnet_name = f"fishnet_{cell_size}m"

# --- Define coordinates ---
origin_coord = f"{xmin} {ymin}"
y_axis_coord = f"{xmin} {ymin + 10}"
corner_coord = f"{xmax} {ymax}"

# --- Create Fishnet ---
arcpy.management.CreateFishnet(
    out_feature_class=fishnet_name,
    origin_coord=origin_coord,
    y_axis_coord=y_axis_coord,
    cell_width=cell_size,
    cell_height=cell_size,
    number_rows="",
    number_columns="",
    corner_coord=corner_coord,
    labels="NO_LABELS",
    template="",
    geometry_type="POLYGON"
)

print(f"{fishnet_name} created successfully.")
```

```
Enter grid spacing in meters (e.g., 800, 1000):
```

Section 3: Multi-Scenario Viewshed Execution

Step 9: Multi-Height Scenario

This section implements a multi-scenario loop that automatically executes the viewshed analysis for multiple user-defined observer heights. Each scenario is processed sequentially, and results are stored as separate output rasters for comparison and further analysis.

```
    # Create dynamic output name
    output_name = f"viewshed_{str(h).replace('.', '_')}m"

    # Run Viewshed
    out_viewshed = Viewshed2(
        in_raster=dem,
        in_observer_features="observer_points_1000m",
        analysis_type="FREQUENCY",
        vertical_error="0 Meters",
        refractivity_coefficient=0.13,
        surface_offset="Obs_Height"
    )

    out_viewshed.save(output_name)

    print(f"{output_name} created successfully.")

print("\nAll scenarios completed.")
```

```
Enter observer heights separated by commas (e.g., 6,15,30):
```

[26]:

Notebook codes explanation:

Step 1:

In this section, I set up the working environment. I imported the necessary ArcPy libraries, activated the Spatial Analyst extension, and defined the workspace where the data is stored.

```
import arcpy
from arcpy import env
from arcpy.sa import *

arcpy.CheckOutExtension("Spatial")
env.workspace = r"C:\Users\HP\Documents\ArcGIS\Projects\final_viewshed_project\final_viewshed

env.overwriteOutput = True

print("Environment ready.")
```

```
Environment ready.
```

Step 2:

In this section, I asked the user to enter the DEM name and checked whether it exists in the workspace. If the DEM does not exist, the script stops to prevent further errors. This ensures that only valid input data is used in the analysis.

```
dem = input("Enter DEM name (e.g., DEM_20m): ")

if not arcpy.Exists(dem):
    raise ValueError("DEM does not exist in workspace.")

print(f"Using DEM: {dem}")
```

```
Enter DEM name (e.g., DEM_20m):  DEM_20m
Using DEM: DEM_20m
```

Complementry step: If the user wants to adjust the DEM resolution—either to speed up the analysis or to increase accuracy—they can use these two resampling steps accordingly. You can use the second function to make sure the cell size of the resampled DEM has change accoringly.

[ ]:

```
#not compulsory to run
#If needed, a lower-resolution DEM can be used to reduce computational cost
resampled_dem = "DEM_20m"

arcpy.management.Resample(
    in_raster=dem,
    out_raster=resampled_dem,
    cell_size=20,
    resampling_type="BILINEAR"
)

print("DEM resampled to 20m.")
```

[ ]:

```
#Testing the resampled DEM
desc = arcpy.Describe("DEM_20m")
print("Cell size:", desc.meanCellWidth)
```

Step 3:

In this section, I used the Describe function to extract the spatial extent of the DEM. I stored the minimum and maximum X and Y coordinates, which will be used later to define the boundaries of the fishnet grid.

```
desc = arcpy.Describe(dem)
extent = desc.extent

xmin = extent.XMin
ymin = extent.YMin
xmax = extent.XMax
ymax = extent.YMax

print("Extent loaded.")
```

```
Extent loaded.
```

Step 4:

In this section, I converted the fishnet polygons into observer points using the FeatureToPoint tool. Each grid cell is transformed into a single point located inside the polygon. These points will be used as observer locations for the viewshed analysis.

[8]:

```
observer_points = f"observer_points_{cell_size}m"

arcpy.management.FeatureToPoint(
    in_features=fishnet_name,
    out_feature_class=observer_points,
    point_location="INSIDE"
)

print(f"{observer_points} created.")
```

```
observer_points_3000m created.
```

Step 5:

In this section, I generated a fishnet grid based on the DEM extent and user-defined grid spacing. The grid size is dynamic, allowing flexible spatial resolution. This grid will later be used to create observer points for the viewshed analysis.

```
# --- Get grid spacing from user ---
cell_size = int(input("Enter grid spacing in meters (for testing try 2000): "))
print(f"Grid spacing set to: {cell_size} meters")

# --- Create dynamic fishnet name ---
fishnet_name = f"fishnet_{cell_size}m"

# --- Define coordinates ---
origin_coord = f"{xmin} {ymin}"
y_axis_coord = f"{xmin} {ymin + 10}"
corner_coord = f"{xmax} {ymax}"

# --- Create Fishnet ---
arcpy.management.CreateFishnet(
    out_feature_class=fishnet_name,
    origin_coord=origin_coord,
    y_axis_coord=y_axis_coord,
    cell_width=cell_size,
    cell_height=cell_size,
    corner_coord=corner_coord,
    labels="NO_LABELS",
    geometry_type="POLYGON"
)

print(f"{fishnet_name} created successfully.")
```

Step 6:

In this section, I counted the number of observer points created from the fishnet grid. The result shows how many observer locations will be used in the viewshed analysis, which also affects the computational load of the model.

```
count = int(arcpy.management.GetCount("observer_points_3000m")[0])
print("Number of observers:", count)
```

```
Number of observers: 36
```

Step 7:

In this section, I performed the Viewshed2 analysis using the DEM and the generated observer points. The analysis type was set to “FREQUENCY,” meaning the output raster shows how many observer points can see each location. The observer height was applied using the previously created field. Finally, the viewshed result was saved to the geodatabase.

```
from arcpy.sa import *

# Enable Spatial Analyst
arcpy.CheckOutExtension("Spatial")

dem = "dgm5m_Clip"
observers = "observer_points_3000m"

viewshed_output = "viewshed_3000m_result"

out_viewshed = Viewshed2(
    in_raster=dem,
    in_observer_features=observers,
    analysis_type="FREQUENCY",
    vertical_error="0 Meters",
    refractivity_coefficient=0.13,
    surface_offset="Obs_Height"
)

out_viewshed.save(viewshed_output)

print("Viewshed 3000m analysis completed successfully.")
```

Complementry step: If the user wants to test the workflow more quickly, they can create a small subset of observer points (e.g., 30 points) instead of using the full dataset. This significantly reduces processing time and allows you to verify that the analysis runs correctly. Once testing is complete, you can run the full analysis using all observer points for final results.

[3]:

```
# Create a small subset of observers for testing
arcpy.analysis.Select(
    in_features="observer_points_1000m",
    out_feature_class="observer_subset_30",
    where_clause="OBJECTID <= 30"
)

print("Subset of 30 observers created.")
```

```
Subset of 30 observers created.
```

[4]:

```
count = int(arcpy.management.GetCount("observer_subset_30")[0])
print("Number of test observers:", count)
```

```
Number of test observers: 30
```

Step 8: Optional Testing Step

This test viewshed step is optional and intended for quick validation of the workflow. It is especially useful when working with high-resolution DEMs or a large number of observer points, but it is not required for the final analysis.

```
#optional testing step
from arcpy.sa import *

arcpy.CheckOutExtension("Spatial")

out_viewshed = Viewshed2(
    in_raster="DEM_20m",
    in_observer_features="observer_subset_30",
    analysis_type="FREQUENCY",
    vertical_error="0 Meters",
    refractivity_coefficient=0.13,
    surface_offset="Obs_Height"
)

out_viewshed.save("viewshed_test_30")

print("Test Viewshed completed.")
```

Step 9:

In these sections, I analyzed the viewshed output raster to calculate visibility statistics. First, I computed the percentage of the total area that is visible from at least one observer. Then, I calculated the percentage of areas with high visibility (visible from 10 or more observers). These metrics help quantify and interpret the visibility results beyond visual inspection.

[6]:

```
import arcpy

# Total cell count
total_cells = 0
visible_cells = 0

with arcpy.da.SearchCursor("out_viewshed", ["Value", "Count"]) as cursor:
    for row in cursor:
        value = row[0]
        count = row[1]
        total_cells += count
        if value > 0:
            visible_cells += count

percentage_visible = (visible_cells / total_cells) * 100

print(f"Visible area percentage: {percentage_visible:.2f}%")
```

```
Visible area percentage: 100.00%
```

[7]:

```
high_cells = 0

with arcpy.da.SearchCursor("out_viewshed", ["Value", "Count"]) as cursor:
    for row in cursor:
        value = row[0]
        count = row[1]
        if value >= 10:
            high_cells += count

high_percentage = (high_cells / total_cells) * 100

print(f"High visibility area percentage: {high_percentage:.2f}%")
```

```
High visibility area percentage: 14.67%
```

Step 10:

In this section, I automated the viewshed analysis for multiple observer heights. The user can enter several height values, and the script loops through each height, updates the observer field, runs the Viewshed2 analysis, and saves each result with a dynamic output name. This enables multi-scenario visibility assessment in a fully automated way.

```
from arcpy.sa import *

arcpy.CheckOutExtension("Spatial")

# --- Get multiple heights from user ---
height_input = input("Enter observer heights separated by commas (e.g., 6,15,30): ")
height_list = [float(h.strip()) for h in height_input.split(",")]

print("Heights to analyze:", height_list)

# --- Loop through each height ---
for h in height_list:

    print(f"\nProcessing height: {h} meters")

    # Update observer height
    arcpy.management.CalculateField(
        in_table="observer_points_1000m",
        field="Obs_Height",
        expression=str(h),
        expression_type="PYTHON3"
    )

    # Create dynamic output name
    output_name = f"viewshed_{str(h).replace('.', '_')}m"
```

```
    # Create dynamic output name
    output_name = f"viewshed_{str(h).replace('.', '_')}m"

    # Run Viewshed
    out_viewshed = Viewshed2(
        in_raster=dem,
        in_observer_features="observer_points_1000m",
        analysis_type="FREQUENCY",
        vertical_error="0 Meters",
        refractivity_coefficient=0.13,
        surface_offset="Obs_Height"
    )

    out_viewshed.save(output_name)

    print(f"{output_name} created successfully.")

print("\nAll scenarios completed.")
```

Step 11:

In this step, I printed the current workspace to verify that all inputs and outputs are being accessed from the correct geodatabase. This helps prevent errors related to missing datasets.

```
print(env.workspace)
```