Lab Report

Title: Lab1

Notice: Dr. Bryan Runck

Author: Rob Hendrickson

Date: 11/02/2022

Repository: https://github.com/RwHendrickson/GIS5571/tree/main/Lab02/Part_2

Time Spent: 25 hours

Abstract

In this lab, I created a walk-ability cost surface for the area around Minnesota's Whitewater State Park. The problem statement section provides contextual information and requirements for analysis. The input data section will describe the different datasets acquired in this exercise. The methods section includes detailed visual and textual descriptions of the workflow conducted. The results section will present some visualizations of the surfaces that were created. This report concludes with a discussion on the lessons learned and future directions of this work.

1 Problem Statement

The client of this project is a fictional character named Dory who lives on a farm near the borders of Olmsted, Wabasha, and Winona counties. She enjoys walking to the North Picnic area of Whitewater State Park from time to time. Her journey is often quite enjoyable, however, it can also be quite treacherous if she chooses the wrong route at the wrong time of year. Her largest concerns are muddy farm fields in the spring and water bodies if there isn't a bridge (or she has her waders). Otherwise, she just prefers the most gradual slope. On the following page is a map of her journey and a table of the requirements for the cost surface construction.

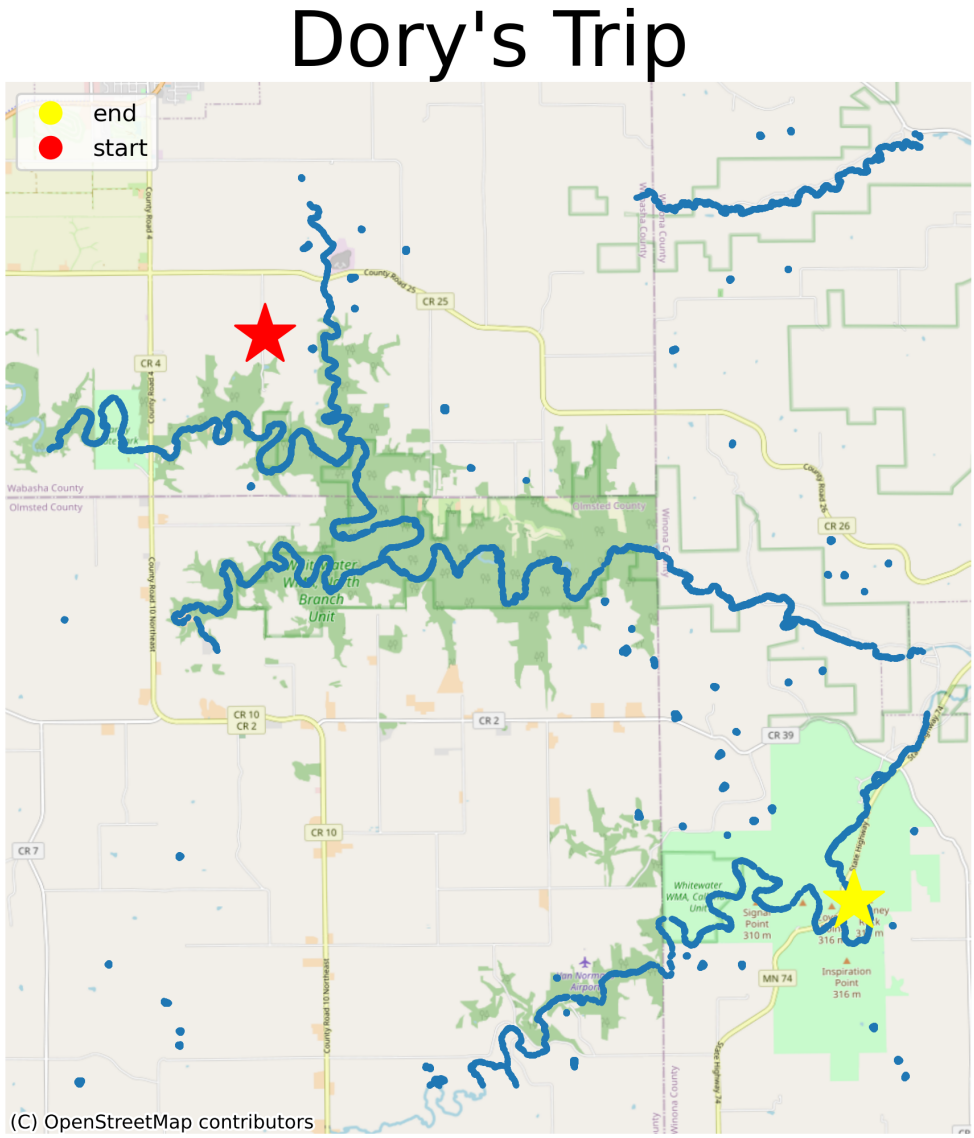

Figure 1: A map of the study area with trip origin, destination, and water highlighted.

	Requirement	Defined As	(Spatial) Data	Attribute Data	Dataset	Preparation
1	Elevation, Water, and Bridges	Acquire and process LiDAR data	Points	Elevation & Category (Water, ground, bridges, etc.)	Minnesota Dept. of Natural Resources	Navigated the API tree
2	Fields	Acquire and select relevant land classification data	Polygons	Landcover Classification	Minnesota Land Cover Classification System	
3	Rasterize	Convert all vectors into a common raster format	Points and Polygons	elevation (integer), is_water_no_bridge (boolean), is_field (boolean)	Elevation points, Water and no bridge points, field polygons	
4	Slope	Calculate slope from elevation	Raster	Elevation (integer)	Rasterized elevation	
5	Standardize	Ensure that all values are in a similar range with appropriate sign	Rasters	slope (float), is_water_no_bridge (boolean), is_field (boolean)	Rasterized slope, is_water_no_bridge, and is_field	Explored distribution of values
6	Weight	Compose weights combinations between the 3 rasters that sum up to 1				
7	Cost Surface	Utilize map algebra to construct a walk-ability surface using all weight combinations	Rasters	standardized slope, standardized is_water_no_bridge, standardized is_field		
8	Uncertainty Analysis	Compare cost surfaces to Dory's preferences	Rasters	Walk-ability Cost		

Table 1: Project requirements

2 Input Data

The data acquired from the Minnesota Department of Natural Resources (MnDNR) were 20 LiDAR tiles from Wabasha, Winona, and Olmsted counties. The full list of tiles is in figure 2 and were determined using each county's respective file title, `tile_index_map.pdf`. These tiles contain geographic information, classification (water, ground, bridge, vegetation, etc), and elevation. Information acquired from Minnesota Land Cover Classification System (MLCCS) contained polygons with classifications based on the MLCCS coding scheme. Their API links can be found in table 2.

```
[('winona', '4342-28-62'),
 ('winona', '4342-28-63'),
 ('winona', '4342-29-62'),
 ('winona', '4342-29-63'),
 ('winona', '4342-30-62'),
 ('winona', '4342-30-63'),
 ('winona', '4342-31-62'),
 ('winona', '4342-31-63'),
 ('wabasha', '4342-28-59'),
 ('wabasha', '4342-28-60'),
 ('wabasha', '4342-28-61'),
 ('olmsted', '4342-29-59'),
 ('olmsted', '4342-29-60'),
 ('olmsted', '4342-29-61'),
 ('olmsted', '4342-30-59'),
 ('olmsted', '4342-30-60'),
 ('olmsted', '4342-30-61'),
 ('olmsted', '4342-31-59'),
 ('olmsted', '4342-31-60'),
 ('olmsted', '4342-31-61')]
```

Figure 2: Full list of LiDAR tiles in the format (county, tile id)

	Title	**Purpose in Analysis**	**Link to Source**
1	MnDNR LiDAR Tiles	Determining location of water & bridges, calculating slope	https://resources.gisdata.mn.gov/pub/data/elevation/lidar/county/
2	MLCCS Land Cover Classifications	Determining location of fields	https://resources.gisdata.mn.gov/pub/gdrs/data/pub/us_mn_state_dnr/biota_landcover_mlccs/shp_biota_landcover_mlccs.zip

Table 2: Data Sources

3 Methods

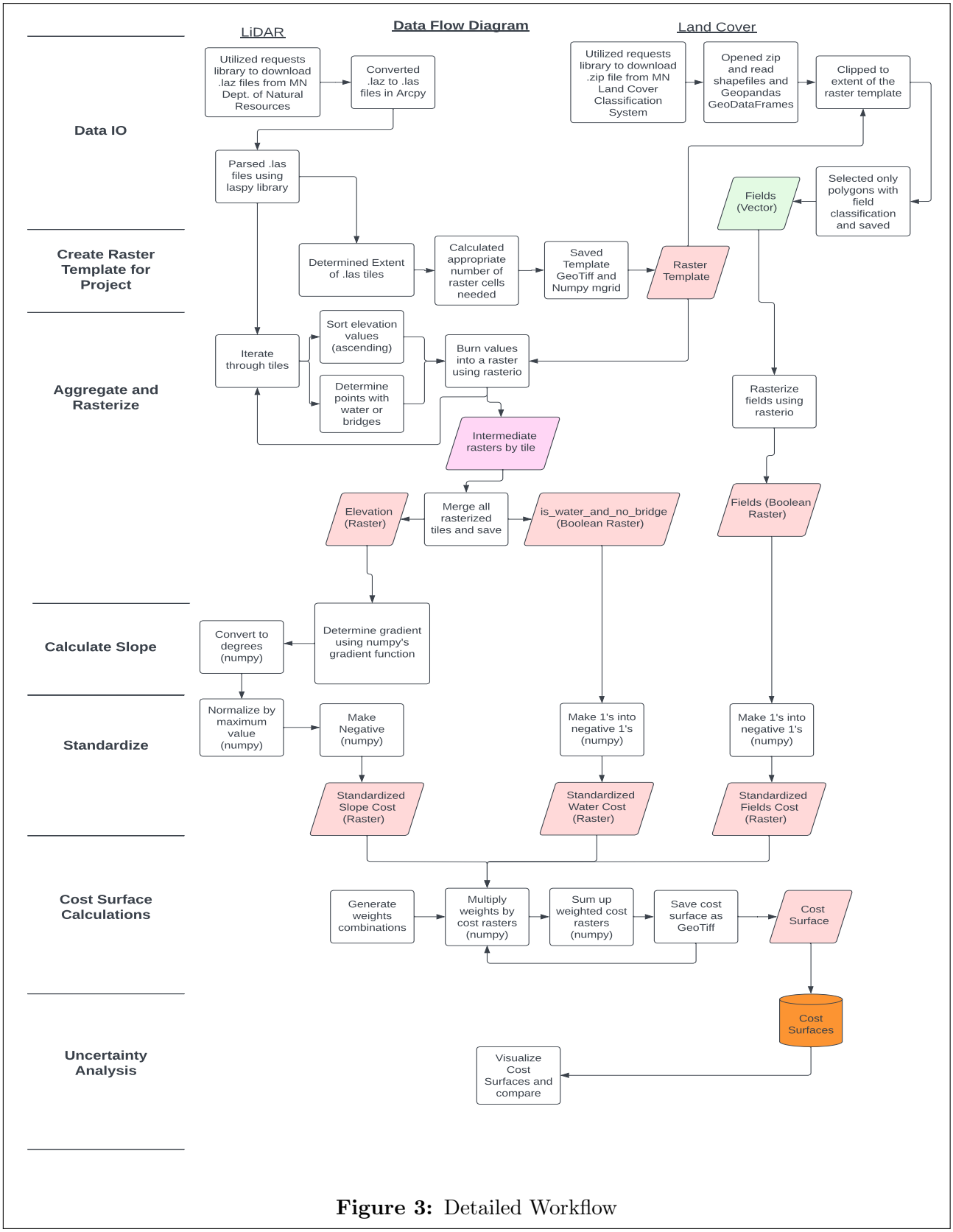

Figure 3: Detailed Workflow

3.1 Data Input/Output

3.1.1 MnDNR

To diagnose which LiDAR tiles were needed from MnDNR, the tile index maps were consulted from each county. An example of this map can be found here: https://resources.gisdata.mn.gov/pub/data/elevation/lidar/county/winona/tile_index_map.pdf. This list of tiles was iterated over and their respective `.laz` files were requested and written to disk. These files were decompressed into `.las` files utilizing Arcpy's ConvertLas function. Code to accomplish these tasks is provided in figures 4 and 5.

```python
# Download Data

downloaded = True

if not(downloaded):
    # Takes like 5 minutes? Maybe?
    # Total size is 458mb

    base_url = 'https://resources.gisdata.mn.gov/pub/data/elevation/lidar/county/'

    for tile_stuff in tile_info:

        county = tile_stuff[0]
        tile_index = tile_stuff[1]

        tile_name = tile_index + '.laz'
        tile_url = base_url + county + '/laz/' + tile_name

        savepath = os.path.join('1_raw_data', 'LazTiles', tile_name)

        if not(os.path.exists(savepath)):

            response = requests.request("GET", tile_url) # Get request

            # Save

            with open(savepath, "wb") as file:
                file.write(response.content)
```

Figure 4: Iteratively interfacing with MnDNR's API

```python
# Convert Laz to Las

converted_to_las = True

if not(converted_to_las):

    # Arcpy Stuff

    import arcpy # Arcpy

    # Set Working Directory (Arcpy)

    arcpy.env.workspace = os.path.join(os.getcwd(), 'Arc1_Lab02_part2.gdb')

    # Convert to las

    arcpy.conversion.ConvertLas('Part_2_LazTiles', os.path.join('1_raw_data','LasTiles'),
                                '1.4', 7, 'NO_COMPRESSION',
                                define_coordinate_system = 'ALL_FILES',
                                in_coordinate_system = arcpy.SpatialReference(26915))
```

Figure 5: Iteratively converting `.laz` files to `.las` files

Once the files were in a `.las` format, they could be accessed with the Python library, `laspy`. The library claims to work with `.laz` files but it didn't work with my version. The extent of all the tiles was ascertained with the code in figure 6. This information was used to create a template raster for analysis (see section 3.2). It should be noted that the `.las` units are in centimeters.

```python
# Find extent of all the tiles

minx = np.inf
miny = np.inf
maxx = np.NINF
maxy = np.NINF

for tile_name in tile_names:

    path = os.path.join('1_raw_data', 'LasTiles', tile_name)

    las = laspy.read(path)

    # Check extent of tile

    if min(las.X) < minx:
        minx = min(las.X)
    if min(las.Y) < miny:
        miny = min(las.Y)
    if max(las.X) > maxx:
        maxx = max(las.X)
    if max(las.Y) > maxy:
        maxy = max(las.Y)
```

Figure 6: Getting the extent of all `.las` tiles

3.1.2 MLCCS

Land cover from MLCCS was accessed through the Minnesota Geospatial Commons's API. This link can be found in table 2. A zip file was downloaded and the shapefile within titled, `landcover_minnesota_land_cover_classification_system.shp`, was read into Geopandas. Here it was clipped to the extent of the template raster (see section 3.2) and selected for the classification, 22 (agricultural land). This selected GeoDataFrame was saved in a `.geojson` format. The code that clips, selects, and saves is provided in figure 7.

```
# Clip to extent

# minx = 56495713 miny = 487566062 maxx = 57761522 maxy = 488967939
# These are in centimeters!!!

# Load template raster bounds

# Get bounds from template
rst = rasterio.open('template.tif') # Open template
bounds = rst.bounds
rst.close()

minx, maxy, maxx, miny = bounds

# Create a custom shapely polygon
extent = Polygon([(minx,miny), (minx, maxy), (maxx, maxy), (maxx, miny), (minx,miny)])
extent_gdf = gpd.GeoDataFrame([1], geometry=[extent], crs='EPSG:26915')

# Clip

landcover_clipped = landcover.clip(extent_gdf)
```

```
# Save Fields (22 classification)

landcover_clipped[landcover_clipped.CARTO == 22].to_file(os.path.join('1_raw_data', 'fields.geojson'))
```

Figure 7: Clipping, selecting, and saving the land cover dataset.

3.2 Creating Raster Template

Upon identifying the extent of the `.las` files, the height and width of the study was computed. These were used to calculate dimensions of a regular grid of 10 meter resolution for the space. This grid was saved in both a Geotiff raster format and numpy mgrid. The code that executes these operations is in figures 8 - 11.

```
# Find the width and height (in meters)

extent = minx_meters, miny_meters, maxx_meters, maxy_meters = (minx/100, miny/100, maxx/100, maxy/100)

height = extent[3] - extent[1]
width = extent[2] - extent[0]

print('height = ', height/1000, 'kilometers, width = ', width/1000, 'kilometers')
```

```
height =  14.018769999999552 kilometers, width =  12.658089999999968 kilometers
```

Figure 8: Calculating width and height of study extent.

```
# Create raster

resolution = 10 # Want maybe 10 x 10m meter cells

x_correction = resolution - np.mod(width, resolution) # It won't be perfect unless we add these
y_correction = resolution - np.mod(height, resolution)

# Split correction amongst min & maxs

new_minx = minx_meters - x_correction/2
new_maxx = maxx_meters + x_correction/2
new_miny = miny_meters - y_correction/2
new_maxy = maxy_meters + y_correction/2

# New heights

new_height = new_maxy - new_miny
new_width = new_maxx - new_minx

print('Corrected Height (for 10m resolution) = ', new_height, 'width = ', new_width)

x_cells = new_width/resolution
y_cells = new_height/resolution

print('x_cells = ', x_cells, '\ny_cells = ', y_cells)
```

```
Corrected Height (for 10m resolution) =  14020.0 width =  12660.0
x_cells =  1266.0
y_cells =  1402.0
```

Figure 9: Computing number of cells in each dimension for raster.

```
# Definitions

def save_geotiff(array, name, crs, resolution, minx, miny):
    '''Saves a numpy array into a geotiff.

    Give name as a string
    crs as int, resolution as int
    minx and miny both as floats
    '''
    transform = Affine.translation(minx - resolution / 2, miny - resolution / 2\
                                   ) * Affine.scale(resolution, resolution)

    with rasterio.open(
        os.path.join(".", name + '.tif'),
        mode="w",
        driver="GTiff",
        height=array.shape[1],
        width=array.shape[0],
        count=1,
        dtype= 'float64',
        crs= rasterio.crs.CRS.from_epsg(crs),
        transform=transform,
    ) as new_dataset:
        new_dataset.write(array, 1)
```

Figure 10: A function for saving a geotiff file.

```
# Save Template as a GeoTiff

array_temp = np.empty([int(x_cells), int(y_cells)])

save_geotiff(array_temp, 'template', 26915, resolution, new_minx, new_miny)

# Save Template as a Numpy Grid (for plotting)
# Must mannually input cells from above.......

raster = np.mgrid[new_minx:new_maxx:1266j,
                  new_miny:new_maxy:1402j]

np.save('template', raster)
```

Figure 11: Saving the template raster.

3.3 Aggregate and Rasterize

3.3.1 Elevation

To get a full elevation raster, the `.las` files were iterated over. Each was read using the `laspy` library and rasterized using Rasterio's `features.rasterize` function. This function does not have an average values option when merging points into a cell, so their elevations were sorted in ascending order and the merge algebra, 'REPLACE' was used. This effectively burned the largest `.las` point value for each cell into the template raster's format and saved it as a geotiff. Once complete, all intermediate rasters were summed up and overlapping cells were averaged. Code that accomplishes these tasks are in figures 12 and 13. This final raster was saved as a geotiff called, `full_elevation.tif`.

```python
# aggregate points into raster
# Takes about 15 minutes

# This version had overlap between the tiles...

for i, tile_name in enumerate(tile_names):

    path = os.path.join('1_raw_data', 'LasTiles', tile_name)

    las = laspy.read(path)

    las_geoms = gpd.points_from_xy(las.X/100, las.Y/100) # Spatialize
    Zs = las.Z # Elevations

    sort_indices = np.argsort(Zs) # Sort ascending

    out_fn = os.path.join('tile_elevations', tile_name[:-4] + '_elevation.tif') # Savepath

    with rasterio.open(out_fn, 'w+', **meta) as out: # Burn features into raster
        out_arr = out.read(1)


        # this is where we create a generator of geom, value pairs to use in rasterizing
        shapes = ((geom,value) for geom, value in zip(las_geoms[sort_indices], Zs[sort_indices]))

        burned = features.rasterize(shapes=shapes,
                                    fill=0,
                                    out=out_arr,
                                    transform=out.transform)#,
                                    # merge_alg=rasterio.enums.MergeAlg.add) # We're not adding them, we're replacing
                                    # This is why we sorted earlier, consistently using the highest value

        out.write_band(1, burned)
```

Figure 12: Iteratively rasterizing the `.las` files.

```python
# Load those rasters!

elevs = np.empty([len(tile_names)], dtype = object) # storage for each tile

for i, tile_name in enumerate(tile_names):

    out_fn = os.path.join('tile_elevations', tile_name[:-4] + '_elevation.tif')

    rast = rasterio.open(out_fn) # Open

    elevs[i] = rast.read(1) # Save band

    rast.close() # Close
```

```python
# Add them up to get elevation raster

full_elev = np.zeros(elevs[0].shape)

for elev in elevs:

    # Do any cells overlap?

    filled = full_elev > 1
    to_fill = elev > 1


    overlaps = np.logical_and(filled, to_fill)

    # Take average if they overlap

    full_elev[overlaps] = (full_elev[overlaps] + elev[overlaps])/2

    # Otherwise

    full_elev[np.invert(overlaps)] += elev[np.invert(overlaps)]
```

Figure 13: Merging intermediate elevation rasters.

3.3.2 Water and Bridges

The `laspy.read()` objects have a classification attribute which yields a numpy array of each point's classification. 2 indicated ground, 9 indicated water, and 14 indicated bridge decks. There is also 8 (model keypoint) and 12 (overlapping points) which could be utilized in future projects. No bridge decks were identified in any of the MnDNR's `.las` files, but it was assumed that breaks in the water points were bridges. After selecting only water classified points, a similar process to that in section 3.3.1 was conducted to rasterize the points to the template raster where the value burned was -1 if water and 0 if not. This final raster was saved as a geotiff called, `full_water.tif`.

3.3.3 Fields

The fields geojson was loaded as a Geopandas Geodataframes and rasterized using the same process as 3.3.1 (without the iteration) and saved as `rasterized_fields.tif`.

3.4 Transform and Normalize Costs

Each of the final rasters, `full_elevation.tif`, `full_water.tif`, and `rasterized_fields.tif` were transformed and/or normalized to align with Dory's preferences. Negative values indicate a cost (which all variables are) and all magnitudes were between 0 and 1. They were saved in the same format as the template using the function in figure 14.

```python
def Save_Geotiff_to_template(array, template_path, save_path):
    '''Saves a numpy array into a geotiff with the same CRS as the template.
    '''

    # Get metadata from template
    rst = rasterio.open(template_path) # Open template
    meta = rst.meta.copy() # Copy template metadata
    # meta.update(compress='lzw') # Good for integers/categorical rasters
    rst.close()

    with rasterio.open(save_path, 'w+', **meta) as out: # Burn features into raster
        out.write_band(1, array)
```

Figure 14: A function to save an array to a template raster.

3.4.1 Slope

To calculate slope, first the gradient was calculated using Numpy's gradient function on the elevation (in meters) and then this was converted into slope degrees (see code in figure 15). The distribution of values was found to be heavily tailed (see figure 16) so it was transformed using log_{10} which produced a multimodal distribution. The logged slope values were further transformed by shifting values by the minimum value and multiplied by -1. Finally they were

normalized by the maximum value to achieve the "standardized" slope cost surface (code in figure 17, distribution in figure 18).

```
# Get the slope for the elevation...

# https://livebook.manning.com/book/geoprocessing-with-python/chapter-11/115
# https://gis.stackexchange.com/questions/361837/calculating-slope-of-numpy-array-using-gdal-demprocessing

# This is like the derivative AKA Gradient

elevation = bands[0]/100 # Elevation is also in centimeters
elevation[elevation == 0] = None # Remove any zeros, those are the "no data"
resolution = 10

px, py = np.gradient(elevation, resolution)

slope = np.sqrt(px ** 2 + py ** 2)

slope_deg = np.degrees(np.arctan(slope))
```

Figure 15: Computing slope degrees.

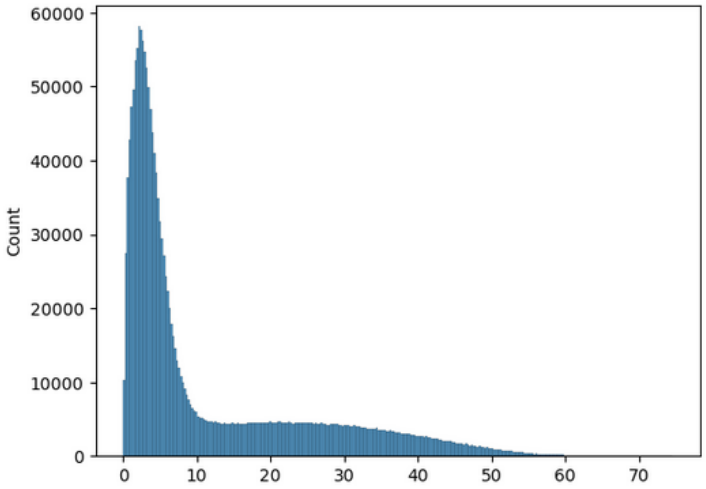

Figure 16: Distribution of slope degree values.

```
# I'd really like to get into this more, but I'm kind of running outta time.

log_slope = np.log(slope_deg)

# I'm going to standardize this by saying:

non_nan = np.invert(np.isnan(log_slope))
non_neg_inf = np.invert(np.isneginf(log_slope))

non_nan_neg_inf = np.logical_and(non_nan,non_neg_inf)

max_ = log_slope[non_nan_neg_inf].max()
min_ = log_slope[non_nan_neg_inf].min()

shifted_log_slope = log_slope - min_ # Shifting so only positive values

std_log_slope_deg = -(shifted_log_slope/(max_ - min_)) # Normalizing to range and making negative

# Slope of zero (where slope_deg = 0) should be no cost (not infinity)

std_log_slope_deg[slope_deg == 0] = 0
```

Figure 17: Transforming and normalizing slope degree.

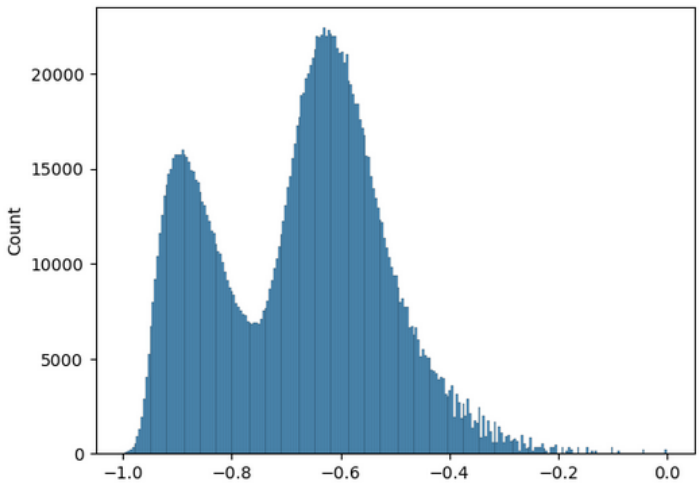

Figure 18: Final distribution of transformed and normalized slope.

3.4.2 Water and Fields

The water and fields boolean rasters were just ensured to be -1 (if field or water) and 0 otherwise.

3.5 Cost Surface Computation

All weights between costs were standardized to be between 0 and 1. Weights combinations (by increments of 0.1) were generated using code in figure 19. These weights combinations were iterated through and multiplied by the respective cost rasters. The resulting summation of the weighted rasters was saved with the title format `__S-__W-__F-Cost_Surface.tif` where the blanks coincide with the percentage that raster was weighted (S = Slope, W = Water, F = Fields). This code can be seen in figure 20.

```
# Constraints
# S_weight + W_weight + F_weight = 1
# 0 < S_w, W_w, F_w < 1

potential_values = np.linspace(0, 1, 11)

weights_tests = []

for S_w in potential_values:
    for W_w in potential_values:
        for F_w in potential_values:
            if int((S_w + W_w + F_w)*100) == 100:
                weights_tests += [[S_w, W_w, F_w]]
```

Figure 19: Creating weights combinations for cost surface creation.

```
weights_tests = np.array(weights_tests)

for weights in weights_tests:

    title = ''

    cost_surface = np.zeros(rasts[0].shape)

    for i, w in enumerate(weights):

        cost_surface += w * rasts[i]

        title += str(int(w*100)) + rast_names[i][0].upper() + '-'

    title += 'Cost_Surface'

    savepath = os.path.join('5_weights_tests', title + '.tif')

    Save_Geotiff_to_template(cost_surface, 'template.tif', savepath)
```

Figure 20: Iterating through weights combinations and saving resulting cost surfaces.

3.6 Uncertainty Analysis

Select cost surfaces were visualized using matplotlib's pcolormesh plot to understand how varying the weights effected the cost surface. An example function that creates this type of visualization can be found in figure 21.

```
def six_plot(rast_names, suptitle):

    # Load necessary data

    raster = np.load('template.npy')

    trip = gpd.read_file('DoryTrip.geojson').to_crs('EPSG:26915')

    f, axs = plt.subplots(2,3, figsize = (24,16))

    norm = mpl.colors.Normalize(vmin=-1, vmax=0) # Norm for coloration

    for i, ax in enumerate(axs.flatten()):

        path = os.path.join('5_weights_tests', rast_names[i])

        # Load that raster

        rast = rasterio.open(path) # Open

        values = rast.read(1) # Save band

        rast.close() # Close

        # Plot it

        art = ax.pcolormesh(raster[0], raster[1], values.T, norm = norm, shading='auto', cmap = 'magma')
        title = rast_names[i][:-17]
        ax.set_title(title, fontsize = 16) # Add title
        ax.set_axis_off()

        trip.plot(categorical = True, cmap = 'Dark2',
            # column = 'type', cmap = 'black',
                  # legend = True,
                  marker = '*', markersize = 200, ax = ax)



    f.colorbar(art, ax=axs[:, :], location='right', shrink=0.85)
    f.suptitle(suptitle, fontsize = 24)
    # f.tight_layout()
    f.savefig(suptitle+'.png', dpi = 300)
```

Figure 21: Function to plot six cost surfaces.

4 Results

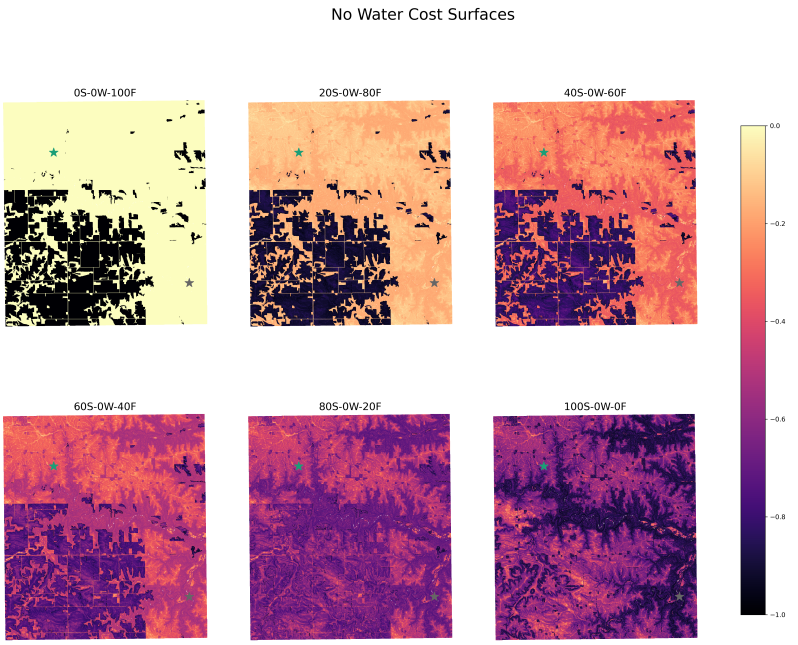

Figure 22: Cost Surfaces with no weight on water cost.

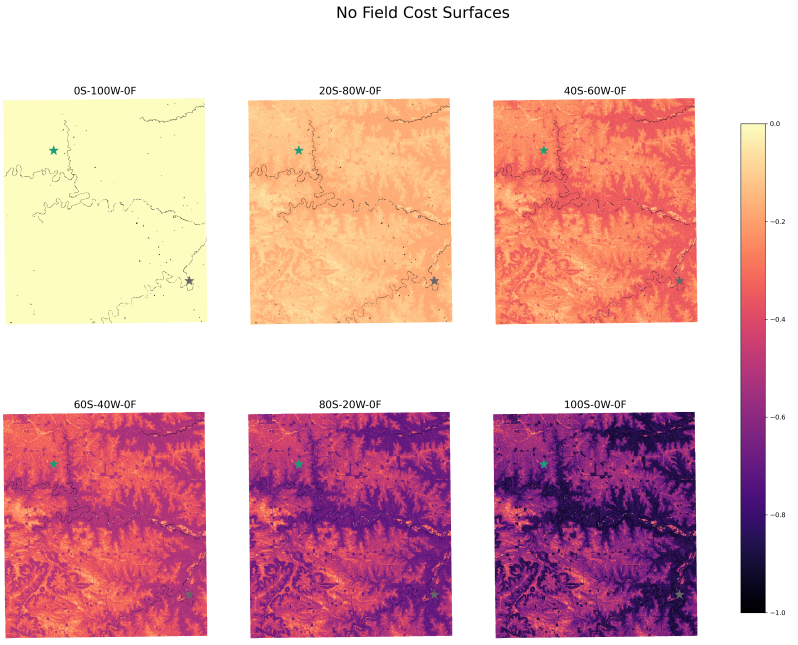

Figure 23: Cost Surfaces with no weight on field cost.

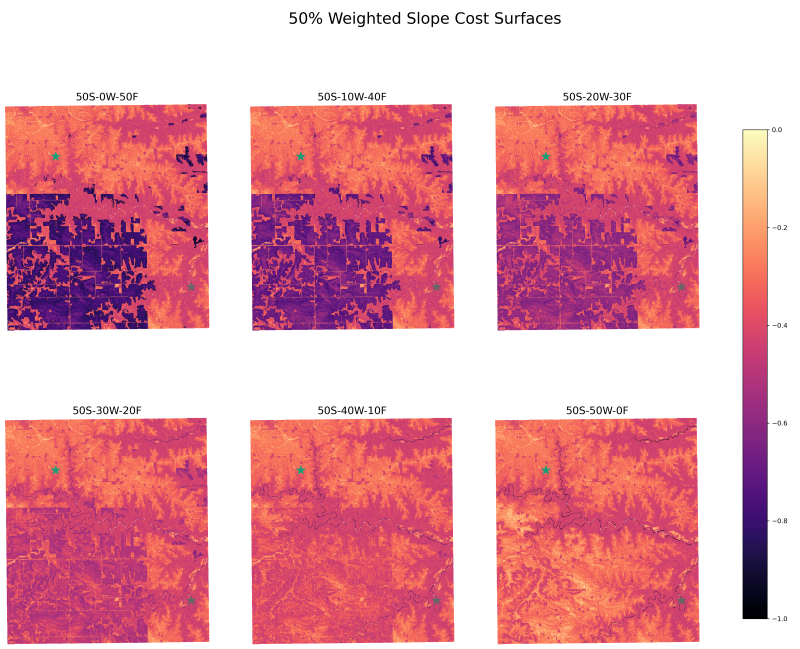

Figure 24: Cost Surfaces with 50% weight on slope cost.

Figure 25: Cost Surfaces with 50% weight on water cost.

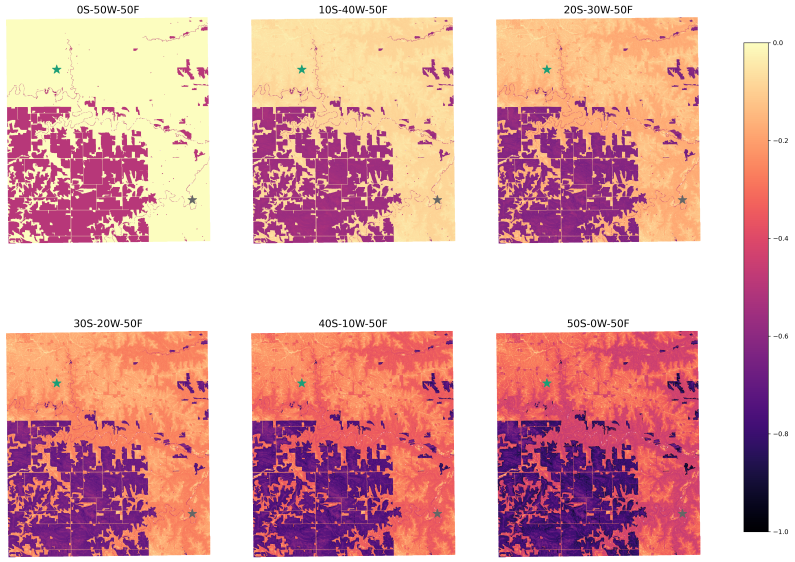

Figure 26: Cost Surfaces with 50% weight on field cost.

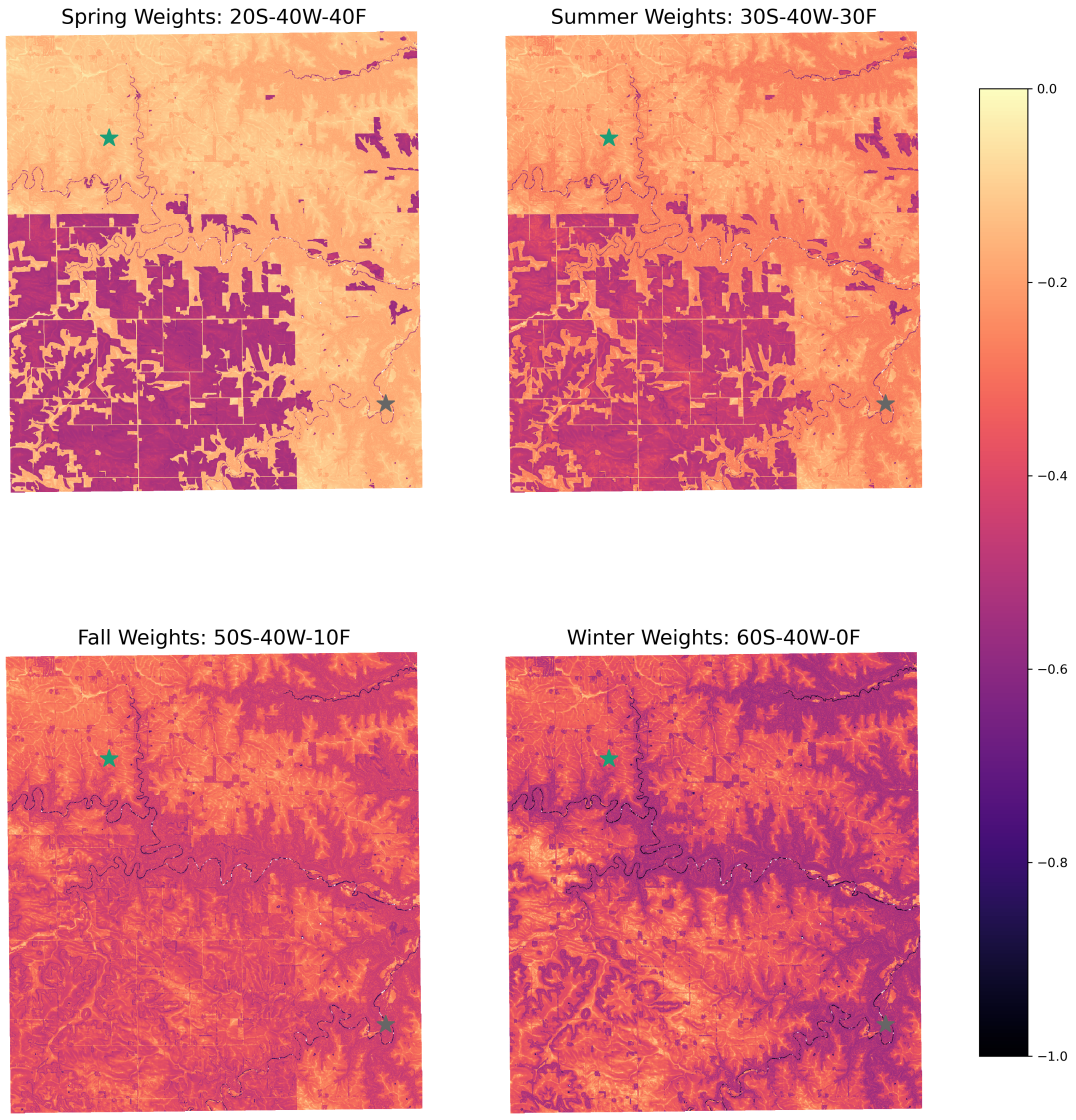

Figure 27: Proposed cost surfaces for each season.

Visually the results appear to be accurate.

5 Discussion and Conclusion

In this lab I explored creating a cost surface using almost all open sourced code.

Interfacing with the API's wasn't too difficult besides dealing with the cross-county `.laz` tiles and the tile index maps. I was a little disappointed that I couldn't get laspy to read the compressed `.laz` files, but I'm sure that this could be resolved with more time.

I have grown quite comfortable over the past year (and certainly more so after this lab) with utilizing Numpy and Rasterio to deal with rasters in Python. The largest concerns in this department were not having an option to average values when merging features into a raster cell and not having the snap to raster feature when rasterizing bridges (though I think this snap feature could be achieved with a little more time potentially using Geopandas' sindex.nearest method). Another insight from this stage in the project was that the classifications of the `.las` points could be used to aid in creating a more accurate surface (such as the model keypoints and overlapping points).

The fun in this lab really began when exploring the slope values across the study area. It was rather curious that the log transform of these values showed a multimodal distribution and it would be cool to categorize the slopes based on this insight (ie. a scale of 0 to 4 in difficulty). The cost surfaces were extremely cool visually (particularly when slope was involved) and the influence the weights had on the surface was another curiosity.

Take the interaction between field and slope (without water) for example. We can see that as we increase the weight of slope, the slope surface slowly materializes underneath the areas designated as fields. Looking at the interaction between two costs at a time led to some knowledge on each that informed the final seasonal cost surfaces. In spring we weighted water and fields higher and can see that this makes much of the southwest of the study area look unappealing as a route to Whitewater Park (except via roads). After the harvest, there may be many good routes through this area! It would be interesting to take these surfaces to the next level and find the most optimal routes. Maybe even do some random walks to get a feeling about the different options Dory has.

Upon completing this lab, I am much more confident with cost surfaces, rasters, and open sourced workflows for dealing with `.las` files and look forward to implementing these

newfound skills into future projects!

Self Score

Category	Description	Points Possible	Score
Structural Elements	All elements of a lab report are included (2 points each): Title, Notice: Dr. Bryan Runck, Author, Project Repository, Date, Abstract, Problem Statement, Input Data w/ tables, Methods w/ Data, Flow Diagrams, Results, Results Verification, Discussion and Conclusion, References in common format, Self-score	28	24
Clarity of Content	Each element above is executed at a professional level so that someone can understand the goal, data, methods, results, and their validity and implications in a 5 minute reading at a cursory-level, and in a 30 minute meeting at a deep level (12 points). There is a clear connection from data to results to discussion and conclusion (12 points).	24	20
Reproducibility	Results are completely reproducible by someone with basic GIS training. There is no ambiguity in data flow or rationale for data operations. Every step is documented and justified.	28	28
Verification	Results are correct in that they have been verified in comparison to some standard. The standard is clearly stated (10 points), the method of comparison is clearly stated (5 points), and the result of verification is clearly stated (5 points).	20	10
		100	82