Flooding on Tribal Lands

Authors: Bri Stone, Shana Larwrence, Jesse Joseph

A Common Issue

Spring of 2019 Tribal Nations in South Dakota suffered from extreme flooding that resulted in road loss, land degradation, and property damage. Most reservations do not have funding or resources to replace or rebuild. In the case of road loss, sometimes that road may be the only route causing the possibility of communities or families to be isolated from resources. Property damage for farmers and ranchers can take years to recover, sometimes resulting in total loss.

Creating a reservation wide map that contains high to low risk levels will help the tribe identify areas to design preventative measures rather than reacting to the situation. Additionally, landowners can see what parts of their property are at risk, allowing them to use different mitigation strategies.

These reasons indicate why it is important for our team to create a universal code that results in a flooding risk map for utilization by all Tribal Entities. We hope the risk map created by the code will help to minimize the issues of land erosion, road and land loss, property damage, land degradation and loss of sacred sites within each reservation.

Prior Work

There are several organizations that have created where flood maps for most places of the United States, but their intended use is for insurance. One such organization is FEMA (https://www.fema.gov/flood-maps), who creates these maps as part of the National Flood Insurance Program (NFIP). Unfortunately when looking at the maps, the information is not very useful to the average person.

Another option was with the US Geological Survey (USGS) when they created flood maps that showed the extent of flooding for different flood severities. Although these maps are more intuitive to understand, they are also very hard to find and, in many cases, hard copies are only located in libraries of Universities. The USGS does have some very useful flood information (https://www.usgs.gov/faqs/where-can-i-find-flood-maps?qt-news_science_products=0#qt-news_science_products) online, but it is limited in scope.

The Method

Our team derived it's method from a similar analysis that was used to create an island wide inundation vulnerability map for the island of St. Lucia, located on the edge of the Caribbean Sea north of Venezuela. The distinctive variables (Land use, Soils, Precipitation, and a Digital Elevation Model) implemented in the document generate risk levels associated with flooding.

To determine the infiltration rate, soil type and land-use identify the Runoff Curve Numbers (CN). These numbers range from 30 to 100, with 30 being the lowest and 100 having the highest potential for runoff. Land-use categorizes CNs by soil hydrologic units (A-D) so that intersecting soil and land-use layers identify the correct CN. Using precipitation data, daily amounts for mean annual rainfall uses only data for extreme precipitation events because they result in flooding.

Finally, horizontal slopes have the capacity for pooling, thus contributing more to flooding. To compensate, slope has a risk levels of one, two, and six instead of three for horizontal gradients are. After determining risk levels of each variable, the summing of the overlapping values to identify the final risk level results in one through six being low risk and anything greater than nine being high risk.

(insert figure 1)

Flow Chart from St. Lucia Document

Conclusion

As is everyone's goal when setting out to complete a project, not all come to fruition. In the case of our code, it is about 80% complete due to time constraints resulting in the exclusion of precipitation data. One of the big reasons why precipitation data was not included at this time is due to identifying the correct interpolation method and then implementing. Many reservations have very few rain gauges that continuously collect data. For Standing Rock Reservation in particular there are only three gauges on the reservation.

Despite precipitation's absence, the results with just Slope, Soil, and Land-use are rather interesting. When looking at the intermediate layer of CN Risk, calculated from Soil and Land-use, it would appear that the results are the opposite of expectations. With one indicating low risk and three as high risk, the resulting map was inverse of the expected results.

(insert Figure 2)

The most likely cause stems from CN value assignments made to the Land-use layer. The information that we were working with did not contain a complete array of CN values for all Land-use types. Instead, the data's focus was agricultural lands. Additionally the St. Lucia document's study sight was an island so there is also reason to believe that the reassignment of CN value to Risk Level may need alterations for an inland location. Instead of CN values of less than 40 qualifying for low risk, inland might receive more accurate results with CN values less than 50 qualifying for low risk. We believe the same issue exists for slope data, thus we increased the ranges used in the St. Lucia paper by a multiple of 10.

Overall, we are proud of the code that we created as an initial base for the analysis. We believe that the foundation created will enable the continuation of building a code that will be able to take user inputs and export a final map with eases. We cannot guarantee that those who will use the code will have a python background, which is why one of our goals was user ease. Down the line, the incorporation of a simple user interface for the code would be possible.

(insert Figure 3)

Moving Forward

The project will require the following work:

- Inclusion of precipitation data using interpolation
- Identifying a way to improve the Land-use portion of CN values
- Verifying that risk ranges for each variable are appropriate
- User interface?

Code Orginization

The code is orginized in the following mannor:

Preperation go

1. Packages
2. Vairables
3. Defined Code Blocks

Analysis go

4. Reservation Boundary Identification
5. Land-use Layer Prep
6. Soil Layer Prep
7. CN Risk Layer Creation
8. Slope Layer Prep
9. Flood Risk Layer Creation

Document Creation go

10. Final Map

Documents

1. Flood Hazard Mapping of St. Lucia (Document)
   - By: Vincent Cooper and Jacob Opadeyi (February 2006)
     - Chukwuka, Azubuike. (2013). Re: Methodology to "generate flood risk maps".. Retrieved from:
       - https://www.researchgate.net/post/Methodology_to_generate_flood_risk_maps/51abca3ed3df3e (https://www.researchgate.net/post/Methodology_to_generate_flood_risk_maps/51abca3ed3df3e
2. Hydrologic Soil-Cover Complexes (Document)
   - USDA-NRCS
   - NEH, Part 650, (EFH), Amend, IA50, NOV, 2007
     - https://www.nrcs.usda.gov/Internet/FSE_DOCUMENTS/nrcs142p2_011485.pdf (https://www.nrcs.usda.gov/Internet/FSE_DOCUMENTS/nrcs142p2_011485.pdf)
3. Additional Runoff Curve Number Information (Website)
   - https://en.wikipedia.org/wiki/Runoff_curve_number (https://en.wikipedia.org/wiki/Runoff_curve_number)

Data Sources

1. Reservation Boundary
   - Originator: Branch of Geospatial Support
   - Data Name: Land Area Representations (LAR)
   - Download format: Zip File with Shapefile(s) (.shp)
   - Website: US Department of the Interior - Indian Affairs (biamaps.doi.gov/bogs/datadownload.html)
   - Publication Date: 2018-12-10
2. Land-Use Layer
   - Originator: U.S. Geological Survey Gap Analysis Program
   - Data Name: GAP/LANDFIRE National Terrestrial Ecosystems 2011
   - Download format: Zip File with Raster(s) (.tif)
   - Website: USGS - Gap Analysis Project (https://www.usgs.gov/core-science-systems/science-analytics-and-synthesis/gap/science/land-cover-data-download?qt-science_center_objects=0#qt-science_center_objects)
   - Publication Date: 2016-05-13
3. Soil Layer
   - Originator: U.S. Department of Agriculture, Natural Resources Conservation Service
   - Data Name: Soil Survey Geographic (SSURGO) database
   - Download format: Zip File with Shapefile(s) (.shp)
   - Website: USDA NRCS Web Soil Survey (https://websoilsurvey.sc.egov.usda.gov/)
   - Publication Date: 2019-09-16
4. Slope Layer
   - Originator: USDA/NRCS - National Geospatial Center of Excellence
   - Data Name: National Elevation Dataset 30 meter
   - Download format: Zip File with Raster(s) (.tif)
   - Website: USDA NRCS Geospatial Data Gateway (https://gdg.sc.egov.usda.gov/)
   - Publication Date: 2000-Present

Preperation

return to Code Orginization

The following areas consist of the information required for the entire code to run.

1. Packages go
2. Vairables go
3. Defined Code Blocks go

Note: Vairables require user input or adjustment

1. Packages

return to Preperation

These are the different tools to import for the code to run. If you do not already have to package please install. Much of the code can be found at the GitHub Earth Lab Repository (https://github.com/earthlab/earth-analytics-python-env). The list of packages associated from the repository can be found here (https://github.com/earthlab/earth-analytics-python-env/blob/master/environment.yml).

The only package used and not found in the Earth Lab Repository is RichDEM (https://richdem.readthedocs.io/en/latest/). This package is used to create a slope layer from the DEM files.

In [ ]:
```
# Imported Packages
# Standard Packages
import os  # related to file directory
from glob import glob  # identifying files within path
import sys  # gdal error message
from datetime import datetime
import time

from datetime import date
import matplotlib as mpl  # plotting data
import matplotlib.lines as mlines  # dissolve
import matplotlib.pyplot as plt  # plotting data
import numpy as np  # used for masking rasters
import numpy.ma as ma  # used for masking rasters
import pathlib  # related to file directory
from pathlib import Path  # related to file directory
import pandas as pd  # read CSVs
import seaborn as sns  # plotting set-up

import earthpy as et  # use
import earthpy.plot as ep  # plotting histogram
import earthpy.spatial as es  # used for cropping raster
import geopandas as gpd  # reading files - vector
from osgeo import gdal, ogr  # used to convert raster to shp
import rasterio as rio  # reading files - raster
from rasterio.features import shapes  # create vector from raster
from rasterio.merge import merge  # mosaic rasters
from rasterio.plot import plotting_extent  # set extent after cropping
from rasterio.plot import show  # plotting mosaic raster
# projecting rasters
from rasterio.warp import calculate_default_transform, reproject, Resampling
import richdem as rd  # open and run DEMs through slope calculations
from shapely.geometry import box  # Map Legend
from shapely.geometry import Polygon  # clip vector
from shapely.geometry import shape  # vector from raster process
from shapely.geometry import shape, mapping  # fixing file for overlay
```

2. Vairables

return to Preperation

User should confirm that the vairable's folder and file names match those on their system. Vairables are grouped by the following:

- Folders go
- Specified Original Data go
- Saved Data Variables go
- Set Values go
- Folder Check Codes go

Folder Vairables

return to 2. Vairables

These vairables contain the pathways for the code to navigate to folders where either data will be collected or saved. Please only change the red text to fit your folder names. In general each layer type has two folders, an original data folder and a results folder. Only layers that are rasters also have a projection folder. This is to help reduce the risk of confusion for the code.

The final layer created of each vairable is then saved in the Risk Layer folder's input folder. This way when looking back on files it is easier to identify which layers were inputs vs. outputs. Any layer that is the result of layers being combined is saved in the Risk Layer folder's output folder.

Next there are three additional folders:

- preped_csv
  - The location for CSV tables to be saved for joinig with decitated vairable. Go to Specific Originial Data for the particulars of setting up the CSVs.
- map_layers
  - These layers are cities, road, streams, and rivers that contains the area which is used for the creation of the final map.
- saved_figs
  - This is where all eight figures will be saved.

For each group the first line is the highest folder to find the remaining data.

- EX: original_data_folders is the folder that contains each layer's folder that contains initial data.

Please make sure that all data is in one folder and not split in Original Data Folders.

- EX: Data downloaded as county1 and count2, all files copied over to county folder

Finally, folders do not need to be created in advance (see Folder Check Codes). As long as the parent folder is set to where you want everything to be pulled from and saved to it will create the folders. The code will end if it creates any of the folders that require original data to be held in the folder. If it does please move the approperate data to the folders listed as created.

Note: The Directory is the highest level folder. It should contain everything all files and folders.

In [ ]:
```
#Starting timer
start = time.process_time()
initial_timestamp = datetime.now().strftime("%d-%b-%Y (%H:%M:%S.%f)")
```

In [ ]:
```
# All folder to exist within parent folder

# Directory
parent_folder = 'P:\\Personal Files\\Education\\FRCC\\NSF_Internship\\NSF_Proj
ect_Files\\Data'

# Original Data Folder
original_data_folders = os.path.join("Original_Data")
landuse_original_folder = os.path.join(
    original_data_folders, "Landuse_Original_Data")
soil_original_folder = os.path.join(
    original_data_folders, "Soil_Original_Data")
slope_original_folder = os.path.join(
    original_data_folders, "Slope_Original_Data")
reservation_boundry_original_folder = os.path.join(
    original_data_folders, "Reservation_Boundary_Layer")

# Layer Prep Results
landuse_result_folder = os.path.join("Landuse_Results")
soil_result_folder = os.path.join("Soil_Results")
slope_result_folder = os.path.join("Slope_Results")

# Risk Layers Results
risk_folder = os.path.join("Final_Results")
risk_input_folder = os.path.join(risk_folder, "Input_Layers")
risk_output_folder = os.path.join(risk_folder, "Output_Layers")

# Reprojections
landuse_reproject_folder = os.path.join(
    landuse_result_folder, "Landuse_Projection")
slope_reproject_folder = os.path.join(slope_result_folder, "Slope_Projection")

# Other
preped_csv = os.path.join(original_data_folders,"Preped_CSVs")
map_layers = os.path.join(original_data_folders,"Map_Layers")
saved_figs = os.path.join("Figures")
```

Specific Originial Data Files

return to 2. Vairables

These vairables are for specific files to access. Please only change full file names. If selecting a specific shape file the extension is ".shp". **Make sure file extensions are Present!**

Special instructions for CSV files

Soil Table:

When opening the Muaggatt table in a spreadsheet program it will be a text file that is | seperated (that symbol is on the same keyborad buttion as \ ). All of the headers are missing but if you go to the SSURGO Table Column Descriptions (https://data.nal.usda.gov/system/files/SSURGO_Metadata_-_Table_Column_Descriptions.pdf) and look for Muaggatt, all of the headers are there in order. Just add a row at the top of the sheet and copy/paste the Column Physical Name into the first cell of each column. **Do not use the Column Label.** Make sure to save the file as a CSV.

Land-use Table:

To prepare the Land-use table there are three steps (starting on step 3 the formulas are for use in excel):

1. First create a spreadsheet with headders of CN_Join, Description, Sub_Description, Hydrologic_Condition, CN-A, CN-B, CN-C, CN-D. To fill in each column all fields, except the first, are derrived from the 2nd and 3rd item found in Documnents. The first column just runs throught the alphabet and will be used later.
2. Each Land-use layer contains two necessary files to create a new one.
    - The first is a text file that ends in Attributes. Save a copy of this file for manipulation.
    - The second is one that ends with .tif.vat.dbf. The data from this file must be coppied into a new spreadsheet. If more than one Land-Use layer is to be used copy the data from each into one new spreadsheet.
3. By now there should be a total of 3 spreadsheets. (CN values, Attributes, coppied dbf) In the attribute file in the very last column give it a headder of "Present" and then insert the following code in to all of the cells:
    - `=IF(IFERROR(VLOOKUP($A2,dbf_file.csv!$A$1:$C$(last_row_number),1,FALSE),"No")="No","`
4. Delete every row that has a "No" under Present. This can be acceved easily if the table is sorted by the Present column and scroll down until the boundry between "Yes" and "No" is found or filter out the "Yes". Then highlight the group of cells that say "No" and delete.
5. This next part is a bit tedious but create a new column and give each description a CN_Join letter from the first table created. Hopefully in future versions of this code, this part can be automated.
6. Add for new columns (CN-A, CN-B, CN-C, CN-D) and in each column us the code below updating the *number* with the correct number for it's column name.
    - `=VLOOKUP($V2,Hydrologic_SoilCover_Complexes_CN.csv!$A$1:$H$(last_row_number),*number`
    - Replace the *number* with one of the following numbers that matches it's column header:
        - 5 = CN-A
        - 6 = CN-B
        - 7 = CN-C
        - 8 = CN-D
7. Copy and paste all values to remove formulas and save as a .csv

These two CSV files should be saved in the folder assigned by the preped_csv vairable.

In [ ]:
```
# Specific Originial Data file - file name goes between the ""
# reservation shapefile name
reservation_path = os.path.join(reservation_boundry_original_folder, "BIA_Nati
onal_LAR.shp")

all_landuse = glob(os.path.join(landuse_original_folder, "*.tif"))

all_soil = glob(os.path.join(soil_original_folder, "*.shp"))

all_dems = glob(os.path.join(slope_original_folder, "*.tif"))

all_soil_ND = os.path.join(soil_original_folder, "soilmu_a_nd085.shp")
all_soil_SD = os.path.join(soil_original_folder, "soilmu_a_sd031.shp")

soil_table = os.path.join(preped_csv, "muaggatt.csv")
landuse_table = os.path.join(preped_csv, "GAP_LANDFIRE_Attributes_with_CN.csv"
)

road_layer = os.path.join(map_layers, "road100k_l_extract.shp")
city_layer = os.path.join(map_layers, "gnispop_p_extract.shp")
river_line_layer = os.path.join(map_layers, "nhd24kst_l_extract.shp")
river_polygon_layer = os.path.join(map_layers, "nhd24kar_a_extract.shp")
```

Saved Data Variables

return to 2. Vairables

These are layers that will be created by the program and then saved in one of the following folder types:

- Layer Prep Results
- Risk Layers Results
- Reprojections

Suggested names are present in red, but user may change to suit their needs.

In [ ]:
```
# Land-use
landuse_mosaic_outpath = os.path.join(landuse_result_folder, "landuse_mosaic.t
if")
landuse_crop_outpath = os.path.join(landuse_result_folder, "landuse_mosaic_cro
p.tif")
landuse_polygon = os.path.join(landuse_result_folder, "landuse_reclass_poly.sh
p")
landuse_post_dissolve_clip = os.path.join(landuse_result_folder, "landuse_poly
_dissolve.shp")
final_landuse_layer = os.path.join(risk_input_folder, "Landuse_CN_Groups.shp")

# Soil
concat_soil = os.path.join(soil_result_folder, "merged_soil.shp")
soil_join_file = os.path.join(soil_result_folder, "hydro_group_join_soil.shp")
final_soil_layer = os.path.join(risk_input_folder, "Soil_Hydro_Group.shp")

# Slope
dem_mosaic_outpath = os.path.join(slope_result_folder, "dem_mosaic.tif")
dem_crop_outpath = os.path.join(slope_result_folder, "dem_mosaic_crop.tif")
slope_outpath = os.path.join(slope_result_folder, "slope.tif")
slope_reclass_outpath = os.path.join(slope_result_folder, "slope_reclass.tif")
slope_reclass_poly = os.path.join(slope_result_folder, "slope_reclass_poly.sh
p")
slope_post_dissolve = os.path.join(slope_result_folder, "slope_poly_dissolve.s
hp")
final_slope_layer = os.path.join(risk_input_folder, "Slope_Risk.shp")

# Risk Layers
CN_soil_land_risk = os.path.join(risk_output_folder, "CN_Risk.shp")
total_risk = os.path.join(risk_output_folder, "Total_Risk.shp")
flood_risk = os.path.join(risk_output_folder, "Flood_Risk.shp")

# Figures
landuse_fig_check = os.path.join(saved_figs, "Fig1_Landuse_vs_Reservation_Boun
dry_Check.png")
landuse_fig_result = os.path.join(saved_figs, "Fig2_Reservation_Landuse.png")
soil_fig_check = os.path.join(saved_figs, "Fig3_Soil_vs_Reservation_Boundry_Ch
eck.png")
soil_fig_result = os.path.join(saved_figs, "Fig4_Soil_Hydro_Group.png")
slope_fig_check = os.path.join(saved_figs, "Fig6_Slope_Raster_vs_Reservation_B
oundry_Check.png")
slope_fig_result = os.path.join(saved_figs, "Fig7_Reservation_Slope_by_Risk_Va
lue.png")
CN_fig_result = os.path.join(saved_figs, "Fig5_Soil_Landuse_CN_Risk.png")
final_fig = os.path.join(saved_figs, "Fig8_Final_Map.png")
```

Set Values

return to 2. Vairables

These variables are constant values either found within a layer or as a predefined value.

Note: The projection vairable is set to use ESPG values.Insert the correct number directly after the colon. Use this site to determin ESPG number: https://spatialreference.org/ (https://spatialreference.org/)

In [ ]:
```
#Used in Analysis
# code is set to use espg values use use this site to ID espg number: https://
spatialreference.org/
site_projection = "epsg:26914"
reservation_field = "LARName"  # field name where reservation name is found
reservation_name = "Standing Rock LAR"  # name of the reservation

# Land-use
# Column Headders to keep moving forward
landuse_join_head = ['raster_val', 'CN-A', 'CN-B', 'CN-C', 'CN-D', 'geometry']

# Soil
soil_dissolve_field = 'hydgrpdcd'  # Column Headder to dissolve by
# Column Headders to keep during join
soil_join_head = ['AREASYMBOL', 'SPATIALVER', 'musym',
                  'muname', 'MUKEY', soil_dissolve_field, 'geometry']
# Slope
# everything with a percent slope greater than this number will be reclassifie
d as 1
slope_reclass_val_1 = 5
# everything with a percent slope less than this number will be reclassified a
s 6
slope_reclass_val_2 = 1

# Result layers
CN_upper_limit = 70  # everything with a CN greater than this number will be a
ssigned a 3
CN_lower_limit = 40  # everything with a CN less than this number will be assi
gned a 1
risk_upper_limit = 6  # everything with a combined risk greater than this numb
er will be assigned a 3
risk_lower_limit = 4  # everything with a combined risk less than this number
 will be assigned a 1
```

In [ ]:
```
# Used in Map Creation
dpi_value = 400
reservation_map_name = "Standing Rock"  # name of reservation for the map disp
lay
road_field = "RTTYP"  # field of road type
road_vairable = ["C", "M", "S", "U"]  # road type primary road code
river_field = "FCODE"  # field name of stream type
river_vairable = [46000, 46006]  # stream code
```

Folder Check Codes

return to 2. Vairables

These two blocks of code are used to check that all the folders are in order. Having the folders in place is not necessary, but will prevent the code from stopping if any of the folders that hold original data are missing.

Please make sure to have the correct data in the correct folder, no subfolders within, for the original data. As a reminder the original data folder vairables are as follows with their needed file types:

- landuse_original_folder

  - All raster files that cover the area

- soil_original_folder

  - All shapefiles that cover the area

- slope_original_folder

  - All DEMs that cover the area

- reservation_boundry_original_folder

  - A shapefile

- preped_csv

  - Must contain the soil table and landuse table

- map_layers

  - Shapefiles for cities, road, streams, and rivers (1 each)

In [ ]:
```
# Checking that parent folder exists, if not it is created.
if os.path.exists(parent_folder):
    os.chdir(os.path.join(parent_folder))
    print("Parent Path Exists")
else:
    os.makedirs(parent_folder)
    os.chdir(os.path.join(parent_folder))
    print("ERROR: Parent path did not exist. Directory has been created.")
```

In [ ]:
```
# Code to check that all necessary folders within the parent exist. If not the
folder is created.

# Folder lists by relational group type
nested_original_data_folders = [original_data_folders,
                                landuse_original_folder,
                                soil_original_folder,
                                slope_original_folder,
                                reservation_boundry_original_folder,
                                preped_csv,
                                map_layers, ]

results_nested_folders = [risk_folder,
                          landuse_result_folder,
                          soil_result_folder,
                          slope_result_folder,
                          saved_figs,
                          risk_input_folder,
                          risk_output_folder,
                          landuse_reproject_folder,
                          slope_reproject_folder]

# Checking for original data folders
need_data = []
for folder in nested_original_data_folders:
    if os.path.exists(folder):
        pass
    else:
        os.makedirs(folder)
        need_data.append(folder)
        print("Folder Created:  %s" % folder)

# If any original data folders created process ends for user to add data
if len(need_data) == 0:
    print("All original data folders exist")
else:
    sys.exit("One or more original data folder was created.\nPlease move origi
nal data to the approperate folders and then restart")

# Checking the rest of the folders for existance
folder_created = []
for folder in results_nested_folders:
    if os.path.exists(folder):
        pass
    else:
        os.makedirs(folder)
        folder_created.append(folder)
        print("Folder Created:  %s" % folder)

# Message to print if no folders were created
if len(folder_created) == 0:
    print("All processing folders exist")
else:
    pass
```

3. Defined Code Blocks

return to Preperation

This is code that will be called upon later in the analysis

An additional Temp folder and files will be created due to the time status tracking. Everytime the code is run the time laps from the start time will be added to a file that has the time stamp from when the code was initiated. The most recent file is a great place to see what the status of the code is at.

In [ ]:
```
# Creating a text file to track processing times
# Text file name is timefile_yymmdd_hhmmss to create a unique file name
filename='timefile_'+str(datetime.now().strftime("%y%m%d_%H%M%S"))
print(filename)

# Confirming that the folder exists, if not creating it before creating the te
xt file
if os.path.exists(os.path.join('temp')):
    with open(os.path.join('temp',filename+'.txt'), 'x') as file_object:
        pass
else:
    os.makedirs(os.path.join('temp'))
    with open(os.path.join('temp',filename+'.txt'), 'x') as file_object:
        pass

# Setting path to text file
timefile=os.path.join('temp',filename+'.txt')

# Initial two lines in the text file
with open(timefile, 'w') as file_object:
    file_object.write('date_time,description,time_elapse(sec)\n')

with open(timefile, 'a') as file_object:
    file_object.write(initial_timestamp+'Overall Start Time,0.0\n')

def time_status(description):
    """tracking when processes finish from the start time

        Parameters
        ----------
        description: text
            a description of what the time stamp means

        Returns
        -------
        Prints a message with the elapse time and saves the elapse time to a t
ext file.
    """
    complete_time=time.process_time()-start
    timestamp=datetime.now().strftime("%y%m%d_%H%M%S")
    with open(timefile, 'a') as file_object:
        file_object.write(timestamp+","+description+","+str(complete_time)+"\n
")
    if (complete_time)/60 <1:
        print(complete_time," Seconds have elapsed - "+description+"\nTimestam
p: "+timestamp)
    elif ((complete_time)/60)/60 <1:
        print((complete_time)/60," Minutes have elapsed - "+description+"\nTim
estamp: "+timestamp)
    else:
        print(((complete_time)/60)/60," Hours have elapsed - "+description+"\n
Timestamp: "+timestamp)
```

In [ ]:
```
# Reprojection Defined Code for Rasters
def reproject_et(inpath, outpath, new_crs):
    dst_crs = new_crs  # new projection

    with rio.open(inpath) as src:
        transform, width, height = calculate_default_transform(
            src.crs, dst_crs, src.width, src.height, *src.bounds)
        kwargs = src.meta.copy()
        kwargs.update({
            'crs': dst_crs,
            'transform': transform,
            'width': width,
            'height': height
        })

        with rio.open(outpath, 'w', **kwargs) as dst:
            for i in range(1, src.count + 1):
                reproject(
                    source=rio.band(src, i),
                    destination=rio.band(dst, i),
                    src_transform=src.transform,
                    src_crs=src.crs,
                    dst_transform=transform,
                    dst_crs=dst_crs,
                    resampling=Resampling.nearest)
    time_status("1 raster reprojected")
```

In [ ]:
```
def around(coords, precision=5):
    result = []
    try:
        return round(coords, precision)
    except TypeError:
        for coord in coords:
            result.append(around(coord, precision))
    return result
```

In [ ]:
```
def layer_precision(geometry, precision=5):
    geojson = mapping(geometry)
    geojson['coordinates'] = around(geojson['coordinates'], precision)
    return shape(geojson)
```

In [ ]:
```
# This code could use help to take care of seam issue when rasters overlap and
# there is variation or are not touching leaving cells with some random "no va
lue".

def raster_mosaic_and_mask(raster_folder, projection, yes):
    """mosaicing 2 or more .tif files from a folder and then applying a mask

        Parameters
        ----------
        raster_folder: path
            folder that contains all of the .tif files to be mosaiced

        projection: epsg value
            the projection that everything should be in.

        yes: string
            indicate yes or no to apply a mask

        Returns
        -------
        masked_mosaic: Pandas DataFrame
            updated Pandas DataFrame with the year set as the index

        first_tif_meta: meta data
            returns the meta data from the first tif file
    """
    tif_file_glob = glob(os.path.join(raster_folder, "*.tif"))
    folder_name = os.path.basename(raster_folder)
    base=folder_name[:-8]
    reproject_folder=base+"_Projection"

    # reproject
    for ned in tif_file_glob:
        reproject_et(inpath = os.path.join(ned),
                     outpath = os.path.join(raster_folder,reproject_folder,os.
path.basename(ned)),
                     new_crs = projection)
    time_status("all rasters reprojected")

    projected_path = glob(os.path.join(raster_folder,reproject_folder,"*.tif"
))

    tifs_to_mosaic = []

    # Open Rasters
    for ned in projected_path:
        src = rio.open(ned)
        tifs_to_mosaic.append(src)
    time_status("open reporjected rasters")

    # merge
    tif_mosaic, tif_out_trans = merge(tifs_to_mosaic)
    time_status("rasters have been mosaiced")

    # getting meta
    with rio.open(projected_path[0]) as src:
```

```
        first_tif_data = src.read()
        first_tif_meta = src.profile

    # getting shape size (x,y)
    tif_width_meta = tif_mosaic.shape[2]
    tif_height_meta = tif_mosaic.shape[1]

    # more meta information
    first_tif_meta['width'] = tif_width_meta
    first_tif_meta['height'] = tif_height_meta
    first_tif_meta['transform'] = tif_out_trans
    time_status("mosaic raster meta data collected")

    # closing nolonger needed files
    for raster in tifs_to_mosaic:
        raster.close()

    if yes=='yes':
        # applying mask to data
        masking_mosaic = np.where(tif_mosaic < 0, True, False)
        masked_mosaic = np.ma.masked_array(tif_mosaic, masking_mosaic)
    elif yes=='no':
        masked_mosaic=tif_mosaic
    else:
        print('Plese list yes or no as the 3rd parameter for the function to r
un.')
    time_status("raster_mosaic_and_mask function compleat")

    return masked_mosaic, first_tif_meta
```

In [ ]:
```
def grid_to_vector(file_outpath, savefile, projection):
    """Opening and converting a csv file to a Pandas Dataframe with
       the year in the file name as an index.

        Parameters
        ----------
        file_outpath: path
            list of the two folders where the tif flies are located.

        savefile: path
            list of the two folders where the tif flies are located.

        projection: epsg value
            the projection that everything should be in.

        Returns
        -------
        nothing, instead it saves the file in the designated path


            landuse_polygon = os.path.join(landuse_result_folder, "landuse_rec
lass_poly.shp")
    """
    mask = None
    with rio.Env():
        with rio.open(file_outpath) as src:
            image = src.read(1)  # first band
            # print(src.crs)
            results = (
                {'properties': {'raster_val': v}, 'geometry': s}
                for i, (s, v)
                in enumerate(
                    shapes(image, mask=mask, transform=src.transform)))
    time_status("raster file opened")

    geometry = list(results)
    time_status("raster geometry collected")

    poly_from_raster = gpd.GeoDataFrame.from_features(
        geometry, crs=projection)
    poly_from_raster.loc[:, "geometry"] = poly_from_raster["geometry"].apply(
        lambda x: layer_precision(x, precision=5))
    poly_from_raster.to_file(savefile)
    time_status("grid_to_vector function compleat")

    return print("Saved to %s" % savefile)
```

Analysis

return to Code Orginization

1. Reservation Boundary Identification go
2. Land-use Layer Prep go
3. Soil Layer Prep go
4. CN Risk Layer Creation go
5. Slope Layer Prep go
6. Flood Risk Layer Creation go

4. Reservation Boundary Identification

return to Analysis

This set of code isolates a particular reservation from the BIA shapefile and updates the projection to match the indicated ESPG value.

In [ ]:
```
# Define Reservation
time_status("Reservation Boundry Start")
reservation_boundry = gpd.read_file(reservation_path)
reservation_aoi = reservation_boundry[reservation_boundry[reservation_field] =
= reservation_name]
reservation_projected = reservation_aoi.to_crs(site_projection)
time_status("reservation projected")
```

In [ ]:
```
del reservation_boundry
del reservation_aoi
del reservation_path
del reservation_field
del reservation_name
time_status("clear out old vairables")
time_status("Reservation Boundry Complete")
```

5. Land-use Layer Prep

return to Analysis

Using the St. Lucia document as a guide the land-use raster data goes through the following steps:

1. Reprojecting and Mosaicing all Layers together
2. Reducing extent to match the Reservation boundry
3. Converting to a shapefile and tightening the file to the Reservation boundary
4. Joining the CN values from the approperate CSV

In [ ]:
```
time_status("Land-use Layer Prep Start")
# Mosaicing all of the land-use rasters together and applying a mask
landuse_mosaic_masked, landuse_meta = raster_mosaic_and_mask(landuse_result_fo
lder,site_projection,"yes")
```

In [ ]:
```
# Reducing the dimensions down by 1 (this is not affecting the data being use
d)
landuse_mosaic_squeezed = landuse_mosaic_masked.squeeze()

# Saving the newly created file mosaic and masked file
with rio.open(landuse_mosaic_outpath, 'w', **landuse_meta) as dst:
    dst.write(landuse_mosaic_squeezed, 1)
time_status("landuse mosaic raster saved")
```

In [ ]:
```
# Opening the mosaic file and cropping it
with rio.open(landuse_mosaic_outpath) as landuse_src:
    # Crop raster data to boundary
    landuse_data_crop, landuse_crop_meta = es.crop_image(
        landuse_src, reservation_projected)
# Define plotting extent using cropped array and transform from metadata
landuse_crop_plot_extent = plotting_extent(
    landuse_data_crop[0], landuse_crop_meta["transform"])

# Making any value that is less than 0 equal to 0. 0 is no value for Land-use
 code.
landuse_data_crop[landuse_data_crop < 0] = 0
time_status("mosaic raster open and meta collected")
```

In [ ]:
```
# Plotting the land-use data to make sure that it cropped correctly
landuse_crop_fig, ax = plt.subplots()

ep.plot_bands(landuse_data_crop,
              ax=ax,
              title="Landuse Cropped to the Extent\nof the Reservation Shapefi
le\nValues = Landuse Code",
              scale=False,
              cmap="tab20c_r",
              extent=landuse_crop_plot_extent)  # Use plotting extent from cro
pped array

reservation_projected.plot(color='None',
                   edgecolor='black',
                   linewidth=3,
                   ax=ax)
plt.show()
time_status("Landuse crop ploted")
plt.draw()
landuse_crop_fig.savefig(landuse_fig_check, dpi=dpi_value)
time_status("landuse crop figure saved")
```

In [ ]:
```
# Reducing the dimensions down by 1 (this is not affecting the data being use
d)
landuse_data_crop_squeezed = landuse_data_crop.squeeze()

# Saving the cropped file
with rio.open(landuse_crop_outpath, 'w', **landuse_crop_meta) as dst:
    dst.write(landuse_data_crop_squeezed, 1)
time_status("landuse crop saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del landuse_mosaic_outpath
del landuse_data_crop
del landuse_data_crop_squeezed
time_status("clear out old vairables")
```

In [ ]:
```
# Converting raster file to shapefile
grid_to_vector(landuse_crop_outpath,landuse_polygon,site_projection)
```

In [ ]:
```
# Opening vector file and disolving boundries by raster_val
landuse_polygon_open = gpd.read_file(landuse_polygon)
time_status("open landuse vector file")
landuse_dissolve_value = landuse_polygon_open[['raster_val', 'geometry']]
time_status("landuse prep dissolve")
landuse_dissolve = landuse_dissolve_value.dissolve(by='raster_val')
time_status("landuse dissolve")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del landuse_polygon
del landuse_polygon_open
del landuse_dissolve_value
time_status("clear out old vairables")
```

In [ ]:
```
# Applying buffer to solve very tiny dissolve errors
landuse_buffer = landuse_dissolve.buffer(0)
time_status("landuse buffer")
```

In [ ]:
```
# Clipping to match boundry
landuse_poly_clip = gpd.clip(landuse_buffer, reservation_projected)
time_status("landuse clip to reservation boundry")
```

In [ ]:
```
# Removing index
landuse_poly_clip = landuse_poly_clip.reset_index(0)
```

In [ ]:
```
# Saving shapefile
landuse_poly_clip.to_file(landuse_post_dissolve_clip)
time_status("landuse clip saved")
```

In [ ]:
```
# Opening shapefile and preparing for join. Making join field match between sh
apefile and table
landuse_table_open = pd.read_csv(landuse_table)
landuse_post_dissolve_clip_open = gpd.read_file(landuse_post_dissolve_clip)
landuse_post_dissolve_clip_open['raster_val'] = landuse_post_dissolve_clip_ope
n.raster_val.astype(
    int)
landuse_table_open['Value'] = landuse_table_open.Value.astype(int)
landuse_table_open.rename(columns={'Value': 'raster_val'}, inplace=True)
time_status("attirbute join prep")
```

In [ ]:
```
# Performing the join based on raster_val
landuse_join = landuse_post_dissolve_clip_open.merge(
    landuse_table_open, on='raster_val', how='outer')
time_status("attribute join complete")
```

In [ ]:
```
# Removing attributs that are not needed
landuse_join_reduced = landuse_join[landuse_join_head]
time_status("landuse headder adjustment")
```

In [ ]:
```
# Plotting land-use showing the different land-use types
landuse_result_fig, ax1 = plt.subplots()
landuse_join_reduced.plot(cmap='tab20c_r',
                     column='raster_val',
                     categorical=True,
                     edgecolor='None',
                     legend=False,
                     ax=ax1)
ax1.set(title="Landuse as a Vector File\nEach color is a different landuse typ
e")
plt.show()
time_status("final landuse layer plot")
plt.draw()
landuse_result_fig.savefig(landuse_fig_result, dpi=dpi_value)
time_status("landuse layer plot saved")
```

In [ ]:
```
# Save File
landuse_join_reduced.to_file(final_landuse_layer)
time_status("final landuse layer saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del landuse_buffer
del landuse_join
del landuse_join_head
del landuse_join_reduced
del landuse_poly_clip
del landuse_table
del landuse_table_open
time_status("clear out old vairables")
time_status("Land-use Layer Prep Complete")
```

6. Soil Layer Prep

return to Analysis

Using the St. Lucia document as a guide the Soil shapefiles go through the following steps:

1. Reprojecting and Merging all layers together
2. Clipping the layer to match the Reservation boundry
3. Joining the layer with Hydrologic Units from the approperate CSV
4. Dissolving boundaries by Hydrologic Units

In [ ]:
```
# Collecting and reprojecting all soil vector layers
time_status("Soil Layer Prep Start")

soil_list = []
for vect in all_soil:
    soil_vect = gpd.read_file(vect)
    soil_proj = soil_vect.to_crs(site_projection)
    time_status("soil file projected")
    soil_list.append(soil_proj)

time_status("soil files collected")
```

In [ ]:
```
# Concatinating and turning into a GeoPandaDataframe, then saving
merged_soil = gpd.GeoDataFrame(pd.concat(soil_list))
time_status("soil files opened and concatinated")

merged_soil.to_file(concat_soil)
time_status("concatinated soil saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del merged_soil
time_status("clear out old vairables")
```

In [ ]:
```
# Opening concatinated soil file and clipping to Reservation boundry
concat_soil_open = gpd.read_file(concat_soil)
time_status("concatinated soil opened")

soil_reduced = gpd.clip(concat_soil_open, reservation_projected)
time_status("concatinated soil clipped to reservation")
```

In [ ]:
```
# Opening prepared soil CSV table and making the Mukey field in
# both the table and shapefile match in type and format

# Fixing the shapefile
soil_reduced['MUKEY']=soil_reduced.MUKEY.astype(str)

# Opening and Fixing the table
soil_table_open = pd.read_csv(soil_table)
soil_table_open['mukey']=soil_table_open.mukey.astype(str)
soil_table_open.rename(columns={'mukey':'MUKEY'},inplace=True)

time_status("soil attribute join prep")
```

In [ ]:
```
# Plotting the soil shapefile to make sure that it cropped correctly
soil_crop_fig, ax1 = plt.subplots()
soil_reduced.plot(cmap='tab20c_r',
                  column='MUKEY',
                  categorical=True,
                  edgecolor='None',
                  legend=False,
                  ax=ax1)
ax1.set(title="Soil Cropped to the Extent\nof the Reservation Shapefile\nEach
 color is a different soil type")
plt.show
time_status("soil crop plotted")
plt.draw()
soil_crop_fig.savefig(soil_fig_check, dpi=dpi_value)
time_status("soil crop plot saved")
```

In [ ]:
```
# Joining the soil shapefile and table together by the MUKEY field
soil_join = soil_reduced.merge(soil_table_open, on='MUKEY', how='outer')
time_status("soil attribute join complete")
```

In [ ]:
```
# Reducing the joined headder and saving
soil_join_reduced = soil_join[soil_join_head]
time_status("soil headder updated")

soil_join_reduced.to_file(soil_join_file)
time_status("soil join saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del concat_soil
del soil_join_reduced
del soil_join_head
del soil_table
del soil_table_open
del concat_soil_open
time_status("clear out old vairables")
```

In [ ]:
```
# Opening soil file and disolving boundries by hydgrpdcd which contains the hydrologic soil group
soil_reduced = gpd.read_file(soil_join_file)
time_status("soil join opened")
soil_dissolve_field_cut = soil_dissolve_field[:10]
soil_dissolve_value = soil_reduced[[soil_dissolve_field_cut, 'geometry']]
time_status("soil prep dissolve")
soil_dissolve = soil_dissolve_value.dissolve(by=soil_dissolve_field_cut)
time_status("soil dissolve")
```

In [ ]:
```
# Reclipping the file, but this might not be necessary
soil_clip = gpd.clip(soil_dissolve, reservation_projected)
time_status("soil dissolve clipped")
soil_clip = soil_clip.reset_index(0)
```

In [ ]:
```
# Plotting the soil layer by Hydrologic group
soil_result_fig, ax1 = plt.subplots()
soil_clip.plot(cmap='Paired_r',
               column=soil_dissolve_field_cut,
               categorical=True,
               edgecolor='None',
               legend=True,
               ax=ax1)
ax1.set(title="Soil by Hydrologic Group")
plt.show
time_status("soil by hydro group plotted")
plt.draw()
soil_result_fig.savefig(soil_fig_result, dpi=dpi_value)
time_status("soil by hydro group plot saved")
```

In [ ]:
```
# Saving final created soil layer
soil_clip.to_file(final_soil_layer)
time_status("soil dissolve clip saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del soil_join_file
del soil_reduced
del soil_clip
del soil_dissolve
del soil_dissolve_value
del soil_dissolve_field
del soil_dissolve_field_cut
time_status("clear out old vairables")
time_status("Soil Layer Prep Complete")
```

7. CN Risk Layer Creation

return to Analysis

Using the St. Lucia document as a guide, this set of code combins the Land-use and Soil layers to create the CN_risk layer. Both layers are intersected and based on Hydrologic Code the approperate CN value from the Land-use iss assigned to the polygon. CN Risk Value is then assigned based on CN range.

In [ ]:
```
# Opening the soil and landuse layers
time_status("CN Risk Layer Start")
final_soil_layer_open = gpd.read_file(final_soil_layer)
time_status("soil layer open")

final_landuse_layer_open = gpd.read_file(final_landuse_layer)
time_status("landuse layer open")
```

In [ ]:
```
# Changing percision for overlay tool
final_soil_layer_open.loc[:, "geometry"] = final_soil_layer_open["geometry"].a
pply(
    lambda x: layer_precision(x, precision=5))
time_status("soil layer precision change")
```

In [ ]:
```
# Changing percision for overlay tool
final_landuse_layer_open.loc[:, "geometry"] = final_landuse_layer_open["geomet
ry"].apply(
    lambda x: layer_precision(x, precision=5))
time_status("landuse layer precision change")
```

In [ ]:
```
# Performing overlay
# If having issues or a time constraints comment out these two lines and uncom
ment the lines below
# Run code through section 6 and restart with the 3 lines below

CN_risk_Join = gpd.overlay(final_soil_layer_open, final_landuse_layer_open, ho
w='intersection')
time_status("Overlay Complete")
```

In [ ]:
```
# 3 Alternitive lines to use when overlay is taking to long/cant take time to
 test issues
# Using the soil and landuse layers created, perform overlay intersection in a
program like arcgis
# Make sure that the path is to the correct file from the spatial program

#soil_landuse_join = os.path.join("ArcGIS", "soil_land_join.shp")
#CN_risk_Join = gpd.read_file(soil_landuse_join)
#time_status("Overlay Complete")
```

In [ ]:
```
# Updating column headders for upcomming math
CN_risk = CN_risk_Join.assign(CN=0, CN_risk=0)
CN_risk = CN_risk.fillna(-9999)

column_old=['CN-A','CN-B','CN-C','CN-D']
column_new=['CN_A','CN_B','CN_C','CN_D']
settoint=[CN_risk.CN_A.astype(int),CN_risk.CN_B.astype(int),
          CN_risk.CN_C.astype(int),CN_risk.CN_D.astype(int)]

for acolumn_old, acolumn_new, aint in zip(column_old,column_new,settoint):
    CN_risk.rename(columns={acolumn_old:acolumn_new}, inplace=True)
    CN_risk[acolumn_new] = aint
time_status("CN Risk headders ready for math")
```

In [ ]:
```
# Identifying the correct CN value to apply to the polygon
hydrolic_code = ['A', 'A/B', 'A/C', 'A/D', 'B', 'B/C', 'B/D', 'C', 'C/D', 'D']
function = [CN_risk['CN_A'], ((CN_risk['CN_A']+CN_risk['CN_B'])/2), ((CN_risk[
'CN_A']+CN_risk['CN_C'])/2),
            ((CN_risk['CN_A']+CN_risk['CN_D']) /
             2), CN_risk['CN_B'], ((CN_risk['CN_B']+CN_risk['CN_C'])/2),
            ((CN_risk['CN_B']+CN_risk['CN_D']) /
             2), CN_risk['CN_C'], ((CN_risk['CN_C']+CN_risk['CN_D'])/2),
            CN_risk['CN_D']]

for hcode, func in zip(hydrolic_code, function):
    CN_risk['CN'] = np.where(
        (CN_risk['Hydrologic'] == hcode), func, CN_risk['CN'])

time_status("CN values updated")
```

In [ ]:
```
# Determining risk level based on upper and lower limits
CN_risk["CN_Risk"] = 2
CN_risk.loc[CN_risk.CN > CN_upper_limit, 'CN_Risk'] = 3
CN_risk.loc[CN_risk.CN < CN_lower_limit, 'CN_Risk'] = 1
CN_risk.loc[CN_risk.CN == -9999, 'CN_Risk'] = -9999

time_status("CN Risk Level Identified")
```

In [ ]:
```
# Dissolving CN Risk by the newly determined risk levels
CN_dissolve_prep = CN_risk[['CN_Risk', 'geometry']]
time_status("CN Risk dissolve prep")
CN_dissolve = CN_dissolve_prep.dissolve(by='CN_Risk')
time_status("CN Risk dissolved")
CN_dissolve = CN_dissolve.reset_index(0)
time_status("CN Risk index reset")
```

In [ ]:
```
# Plotting CN Risk values
CN_value_fig, ax1 = plt.subplots()
CN_dissolve.plot(cmap='autumn_r',
                 column='CN_Risk',
                 categorical=True,
                 edgecolor='None',
                 legend=True,
                 ax=ax1)
ax1.set(title="Risk Level of Soil and Landuse")
plt.show()
time_status("CN Risk plotted")
plt.draw()
CN_value_fig.savefig(CN_fig_result, dpi=dpi_value)
time_status("CN Risk plot saved")
```

In [ ]:
```
# Saving CN Risk layer
CN_dissolve.to_file(CN_soil_land_risk)
time_status("CN Risk saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del final_landuse_layer
del final_soil_layer
del final_soil_layer_open
del final_landuse_layer_open
del CN_risk_Join
del CN_risk
del CN_dissolve_prep
del CN_dissolve
time_status("clear out old vairables")
time_status("CN Risk Layer Complete")
```

8. Slope Layer Prep

return to Analysis

Using the St. Lucia document as a guide the DEMs go through the following steps:

1. Reprojecting and Mosaicing all Layers together
2. Reducing extent to match the Reservation boundry
3. Running a Slope Analysis to determine percent slope
4. Converting to a shapefile and tightening the file to the Reservation boundary
5. Assigning Risk Value based on slope range
6. Dissolving boundaries by Risk Value

In [ ]:
```
time_status("Slope Layer Prep Start")

# Mosaicing all of the land-use rasters together and applying a mask
dem_mosaic_masked, dem_meta = raster_mosaic_and_mask(slope_result_folder,site_
projection,"yes")
```

In [ ]:
```
# Cleaning up the data by removing single-dimensional entries
dem_mosaic_squeezed = dem_mosaic_masked.squeeze()
time_status("dem mosaic squeezed")
```

In [ ]:
```
# Write mosaiced raster to folder
with rio.open(dem_mosaic_outpath, 'w', **dem_meta) as dst:
    dst.write(dem_mosaic_squeezed, 1)
time_status("dem mosaic saved")
```

In [ ]:
```
# Opening the mosaic file and cropping it
with rio.open(dem_mosaic_outpath) as dem_src:
    # Crop raster data to boundary
    dem_data_crop, dem_crop_meta = es.crop_image(
        dem_src, reservation_projected)
# Define plotting extent using cropped array and transform from metadata
dem_crop_plot_extent = plotting_extent(
    dem_data_crop[0], dem_crop_meta["transform"])

time_status("dem cropped to reservation boundry")
```

In [ ]:
```
# Plotting DEM to make sure that it cropped correctly
slope_crop_fig, ax = plt.subplots()

ep.plot_bands(dem_data_crop,
              ax=ax,
              title="DEMs Cropped to the Extent\nof the Reservation Shapefile
\nValues = Elevation in meters",
              scale=False,
              cmap="gray",
              extent=dem_crop_plot_extent)  # Use plotting extent from cropped
array

reservation_projected.plot(color='None',
                   edgecolor='teal',
                   linewidth=3,
                   ax=ax)
plt.draw()
time_status("dem crop check ploted")

slope_crop_fig.savefig(slope_fig_check, dpi=dpi_value)
time_status("dem crop plot saved")
```

In [ ]:
```
# Squeezing DEM again and saving
dem_data_crop_squeezed = dem_data_crop.squeeze()
with rio.open(dem_crop_outpath, 'w', **dem_crop_meta) as dst:
    dst.write(dem_data_crop_squeezed, 1)
time_status("dem crop save")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del dem_data_crop
del dem_data_crop_squeezed
time_status("clear out old vairables")
```

In [ ]:
```
# Opening DEM with richdem and then calculating slope percentage
dem_crop_open = rd.LoadGDAL(dem_crop_outpath)
time_status("opening dem with richdem")

slope_calculated = rd.TerrainAttribute(
    dem_crop_open, attrib='slope_percentage')
time_status("slope calcualted")
```

In [ ]:
```
# Reclassifying Slope to risk levels.
# Note that risk level 1 is first reclassified as 1000 due to number conflicts
before changing to 1
slope_reclass = slope_calculated
slope_reclass[slope_reclass > slope_reclass_val_1] = 1000
slope_reclass[(slope_reclass >= slope_reclass_val_2)
              & (slope_reclass < 999)] = 2
slope_reclass[slope_reclass < slope_reclass_val_2] = 6
slope_reclass[slope_reclass == 1000] = 1

time_status("slope reclassified")
```

In [ ]:
```
# Recreating a slope layer that has not been reclassified
slope_orignal_calc = rd.TerrainAttribute(
    dem_crop_open, attrib='slope_percentage')
time_status("slope calcualted again")
```

In [ ]:
```
# Saving a copy of the reclassified and non-reclassified slope
with rio.open(slope_outpath, 'w', **dem_crop_meta) as dst:
    dst.write(slope_orignal_calc, 1)

with rio.open(slope_reclass_outpath, 'w', **dem_crop_meta) as dst:
    dst.write(slope_reclass, 1)

time_status("slope raster and reclassified slope raster saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del slope_calculated
del slope_reclass
del slope_orignal_calc
time_status("clear out old vairables")
```

In [ ]:
```
# Converting reclassified slope raster to a vector file
grid_to_vector(slope_reclass_outpath,slope_reclass_poly,site_projection)
```

In [ ]:
```
# Dissolving vector by raster value, which is the reclassified numbers
slope_reclass_poly_open = gpd.read_file(slope_reclass_poly)
time_status("open slope vector file")

slope_dissolve_value = slope_reclass_poly_open[['raster_val', 'geometry']]
time_status("slope prep dissolve")

slope_dissolve = slope_dissolve_value.dissolve(by='raster_val')
time_status("slope dissolved")
```

In [ ]:
```
# Saving dissolved slope file
slope_dissolve.to_file(slope_post_dissolve)
time_status("dissolved slope saved")
```

In [ ]:
```
# Buffering dissolved slope file
slope_buffer = slope_dissolve.buffer(0)
time_status("slope buffered")
```

In [ ]:
```
# Clipping slope vector file to match reservation boundry
slope_clip = gpd.clip(slope_buffer, reservation_projected)
slope_clip = slope_clip.reset_index(0)
slope_clip.rename(columns={'raster_val': 'Slope_Risk'}, inplace=True)
time_status("slope clipped")
```

In [ ]:
```
# Plotting slope showing the risk levels
slope_result_fig, ax1 = plt.subplots()
slope_clip.plot(cmap='autumn_r',
                    column='Slope_Risk',
                    categorical=True,
                    edgecolor='None',
                    legend=True,
                    ax=ax1)
ax1.set(title="Slope Reclassified to Risk Levels")

plt.show()
time_status("clipped slope plotted")

plt.draw()
slope_result_fig.savefig(slope_fig_result, dpi=dpi_value)
time_status("clipped slope plot saved")
```

In [ ]:
```
# Saving Slope File
slope_clip.to_file(final_slope_layer)
time_status("clipped slope saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del slope_reclass_poly_open
del slope_dissolve_value
del slope_dissolve
del slope_clip
time_status("clear out old vairables")
time_status("Slope Layer Prep Complete")
```

9. Flood Risk Layer Creation

return to Analysis

Using the St. Lucia document as a guide, this set of code combines CN Risk and Slope. Both layers are intersected and then the sum of their risk values is totaled. Flood Risk is then assigned based on Total Risk range.

In [ ]:
```
time_status("Flood Risk Layer Start")

# Opening the slope and CN risk files
final_slope_layer_open = gpd.read_file(final_slope_layer)
CN_soil_land_risk_open = gpd.read_file(CN_soil_land_risk)
time_status("slope and cn risk open")
```

In [ ]:
```
# Changing raster_val back to Slope_Risk
final_slope_layer_open.rename(columns={'raster_val': 'Slope_Risk'}, inplace=Tr
ue)
time_status("slope column name update")
```

In [ ]:
```
# Changing percision for overlay tool
final_slope_layer_open.loc[:, "geometry"] = final_slope_layer_open["geometry"]
.apply(
    lambda x: layer_precision(x, precision=5))
time_status("slope precision updated")
```

In [ ]:
```
# Performing overlay
total_risk_Join = gpd.overlay(CN_soil_land_risk_open, final_slope_layer_open,
                              how="intersection")
time_status("overlay complete")
```

In [ ]:
```
# Saving overlay
total_risk_Join.to_file(total_risk)
time_status("overlay saved")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del final_slope_layer
del CN_soil_land_risk
del final_slope_layer_open
del CN_soil_land_risk_open
del total_risk_Join
time_status("clear out old vairables")
```

In [ ]:
```
# Opening total risk file and creating new total risk attribute with a value o
f 0
total_risk_open = gpd.read_file(total_risk)
calc_risk = total_risk_open.assign(total_risk=0)
time_status("total risk field created")
```

In [ ]:
```
# Calculating total risk value
calc_risk["Total_Risk"] = calc_risk["CN_Risk"] + calc_risk["Slope_Risk"]
calc_risk.loc[calc_risk.CN_risk == -9999, 'Total_risk'] = -9999
time_status("total risk calculated")
```

In [ ]:
```
# Dissolving risk by total risk
calc_risk_dissolve_prep = calc_risk[['Total_Risk', 'geometry']]
time_status("total risk prep dissolve")

calc_risk_dissolve = calc_risk_dissolve_prep.dissolve(by='Total_Risk')
time_status("total risk dissolve")

calc_risk_dissolve = calc_risk_dissolve.reset_index(0)
time_status("total risk index reset")
```

In [ ]:
```
#Creating risk level attribute
final_risk = calc_risk_dissolve.assign(Risk_Level=0)
time_status("risk level field created")
```

In [ ]:
```
# Calculating risk level based on total risk value
final_risk["Risk_Level"] = 2
final_risk.loc[final_risk.Total_risk > risk_upper_limit, 'Risk_Level'] = 3
final_risk.loc[final_risk.Total_risk < risk_lower_limit, 'Risk_Level'] = 1
final_risk.loc[final_risk.Total_risk == -9999, 'Risk_Level'] = -9999
time_status("risk level calculated")
```

In [ ]:
```
# Dissolving final risk by risk level
final_risk_dissolve_prep = final_risk[['Risk_Level', 'geometry']]
time_status("total risk prep dissolve")

final_risk_dissolve = final_risk_dissolve_prep.dissolve(by='Risk_Level')
time_status("total risk dissolve")

final_risk_dissolve = final_risk_dissolve.reset_index(0)
time_status("final risk index reset")
```

In [ ]:
```
# Creating new attributes
final_risk_dissolve = final_risk_dissolve.assign(
    Flood_Risk="empty", Area=0, Acres=0, Sq_Mi=0)
time_status("additional fields created")

# Giving descriptive label to risk level under flood risk
final_risk_dissolve.loc[final_risk_dissolve.Risk_Level ==
                        3, 'Flood_Risk'] = "High"
final_risk_dissolve.loc[final_risk_dissolve.Risk_Level ==
                        2, 'Flood_Risk'] = "Medium"
final_risk_dissolve.loc[final_risk_dissolve.Risk_Level ==
                        1, 'Flood_Risk'] = "Low"
final_risk_dissolve.loc[final_risk_dissolve.Risk_Level == -
                        9999, 'Flood_Risk'] = "No Value"
time_status("descriptive risks given")
```

In [ ]:
```
# Calculating Area, Acres, and Sqare Miles
final_risk_dissolve["Area"] = final_risk_dissolve['geometry'].area
time_status("area calculated")

final_risk_dissolve["Acres"] = final_risk_dissolve['Area']/4046.85642
time_status("acres calculated")

final_risk_dissolve["Sq_Mi"] = final_risk_dissolve['Area']/2590000
time_status("square miles calculated")
```

In [ ]:
```
# Saving final flood risk layer
final_risk_dissolve.to_file(flood_risk)
time_status("FLOOD RISK SAVED!!!!")
```

In [ ]:
```
# Clearing out vairables from ram memory that are no longer needed
del total_risk_open
del calc_risk
del calc_risk_dissolve_prep
del calc_risk_dissolve
del final_risk
del final_risk_dissolve_prep
del final_risk_dissolve
time_status("clear out old vairables")
time_status("Flood Risk Layer Complete")
```

Document Creation

return to Code Orginization

10. Final Map

The set of code below results in a map being created showing the results of the analysis above.

In [ ]:
```
# Adjust font size and style of all plots in notebook with seaborn
sns.set(font_scale=1.5, style="whitegrid")
```

In [ ]:
```
# Vector Layers - opening, reporjecting, and clipping to reservation boundry

# cities
cities = gpd.read_file(city_layer)
cities_prj = cities.to_crs(site_projection)
bound_cities = gpd.clip(cities_prj, reservation_projected)
time_status("Cities are Ready")

#roads
road_file = gpd.read_file(road_layer)
road_prj = road_file.to_crs(site_projection)
bound_road = gpd.clip(road_prj, reservation_projected)
# Identifying which roads to include
road_primary = bound_road[bound_road[road_field] == road_vairable]
time_status("Roads are Ready")

# rivers (polygon layer)
river_file = gpd.read_file(river_polygon_layer)
river_prj = river_file.to_crs(site_projection)
bound_rivers = gpd.clip(river_prj, reservation_projected)
time_status("Rivers are Ready")

# streams (polyline layer)
stream_file = gpd.read_file(river_line_layer)
stream_prj = stream_file.to_crs(site_projection)
bound_stream = gpd.clip(stream_prj, reservation_projected)
# Identifying which streams to include
streams = bound_stream[bound_stream[river_field].isin(river_vairable)]
time_status("Streams are Ready")
```

In [ ]:
```
# Identifying how to plot the data from the flood risk layer
flood_results = gpd.read_file(flood_risk)

field_level = ['High','Medium','Low']
plot_level = [flood_results_high,flood_results_medium,flood_results_low]

for field, plot in zip(field_level,plot_level):
    plot = flood_results[flood_results['Flood_Risk'].isin(field)]
time_status("Flood Layer Ready")
```

In [ ]:
```
# Creating hillshade raster layer
with rio.open(dem_mosaic_outpath) as dem_src:
    # Crop raster data to boundary
    dem_data_crop, dem_crop_meta = es.crop_image(
        dem_src, reservation_projected)

# Define plotting extent using cropped array and transform from metadata
dem_crop_plot_extent = plotting_extent(
    dem_data_crop[0], dem_crop_meta["transform"])


squeezed_dem = dem_data_crop.squeeze()

dem_hillshade = es.hillshade(squeezed_dem)
time_status("Hillshade Ready")
```

In [ ]:
```
# Plot the data
import matplotlib.patches as patches
final_flood_risk_map, ax = plt.subplots(figsize=(15, 15))

# Hillshade
ep.plot_bands(dem_hillshade,
              cmap='gray',
              extent=dem_crop_plot_extent,
              ax=ax,
              cbar=False)

# Flood Risk
color = ['red','orange','yellow']
flood_level = [flood_results_high,flood_results_medium,flood_results_low]
label = ['High Risk','Medium Risk','Low Risk']
alpha = [0.6,0.7,0.7]

for acolor, aflood, alabel, aalpha in zip(color,flood_level,label,alpha):
    aflood.plot(color=acolor,
                column='Flood_Risk',
                edgecolor='None',
                label=alabel,
                alpha=aalpha,
                legend=True,
                ax=ax)

# Cities
bound_cities.plot(color='black',
                  marker='.',
                  markersize=200,
                  label='Cities',
                  legend=True,
                  ax=ax,
                  zorder=3)

# Roads
bound_road.plot(color='black',
                label='Roads',
                ax=ax,
                legend=True,
                zorder=2)

# Rivers
streams.plot(color='blue',
                   linewidth=2,
                   label='Rivers',
                   ax=ax,
                   legend=True,
                   zorder=1)

bound_rivers.plot(color='cyan',
                  edgecolor='blue',
                  ax=ax,
                  zorder=1)

legend_rect = [rectHigh,rectMedium,rectLow]
```

```
for acolor, alabel, arect in zip(color,label,legend_rect):
    arect = patches.Rectangle((0, 0), 1, 1, linewidth=1.5,
                              label=alabel, color=acolor)
    plt.gca().add_patch(arect)

# Legend
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles, labels)
# plt.legend('fontsize'=40)
# Title
tyear = str(date.today().year)
ax.set_title(reservation_map_name+"\nFlood Risk Map, "+tyear, fontsize=40)

plt.show()
time_status("Final Map Plotted")
plt.draw()
final_flood_risk_map.savefig(final_fig, dpi=dpi_value)
time_status("Final Map Plot Saved")
time_status("PROCESS COMPLETE!!!!!")
```