Research Methodology Plan

The following plan initiates the data processing phase of the research project. The research project's study area is Worldwide Reference System – 2 (WRS-2): Path 227 / Row 065. The research project's study period is from 11 February 2013 – 30 September 2023. The start date of the study period coincides with the launch of Landsat 8 and will also contain all Landsat 9 images following its launch on 27 September 2021. At this point, the Landsat images and their associated metadata have been successfully downloaded from https://earthexplorer.usgs.gov/. All images and metadata files are located in a single folder.

1. Scenes must first be separated into individual scene folders that contain all required images and metadata. The following code demonstrates the process followed to accomplish this task.

```
import os

# "inpath" is the path to where the images were downloaded
inpath = r''
# "outpath" is where the data will be stored for the next process
outpath = r''
```

Figure 1: *Import the* ***os*** *module to provide functionality later in the code. Define the input path as* ***inpath*** *and the output path as* ***outpath****. The "r" before the string apostrophes signifies the input string is in a raw format.*

```
newfolders = set()

# Extract scene names from file names in download directory
for path, _, files in os.walk(inpath):
    for f in files:
        fsplit = f.split('_')
        newfolders.add('_'.join(fsplit[:7]))
```

Figure 2: *Define a new empty set as* ***newfolders****. Use* ***os.walk*** *to explore the* ***inpath*** *file structure. Iterate through the files and capture unique filename designations in* ***newfolders****.*

```
# Create scene folders in "outpath" directory
for nfold in newfolders:
    try:
        os.makedirs(os.path.join(outpath, nfold))
    except:
        pass
```

Figure 3: *Iterate through **newfolders** and create the new folders that will contain the images and metadata for individual scenes.*

```
# Move scene files into newly created scene folders
for path, _, files in os.walk(inpath):
    for f in files:
        fsplit = f.split('_')
        if not fsplit[0] in path.split('\\'):
            os.rename(
                os.path.join(path, f),
                os.path.join(outpath, '_'.join(fsplit[:7]), f)
                )
            print('_'.join(fsplit[:7]))
```

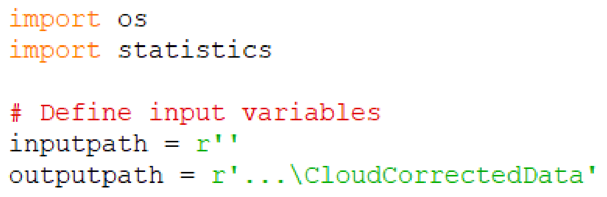
Figure 4: *Use **os.walk** to reexplore files and place them into the newly created scene folders based on the same logic used to define the scene folder names – Landsat Product ID.*

```
# Separate scene folders based on platform
scenesdic = {}
for folder in os.listdir(outpath):
    platform = folder.split('_')[0]
    if platform not in scenesdic:
        scenesdic[platform] = {}
    old = os.path.join(outpath, folder)
    new = os.path.join(outpath, platform, folder)
    scenesdic[platform][old] = new

for platform in scenesdic:
    if platform not in os.listdir(outpath):
        os.makedirs(os.path.join(outpath, platform))
    for path in scenesdic[platform]:
        os.rename(path, scenesdic[platform][path])
```

Figure 5: *Create the **scenesdic** dictionary to capture the **platform** of each scene, the current path of each scene folder – **old**, and the path to where each scene will be moved – **new**. Iterate through **scenesdic** to initially create the **platform** directory, if it does not already exist, and then move each scene folder into its appropriate platform directory.*

2. The next step in the process concerns the removal of cloud contaminated pixels from each band being used in the forest disturbance analysis. This is an important step in the process to ensure cloud contamination does not affect the subsequent processes. The *_QA_PIXEL.TIF contains quality assessment statistics for each scene. Pixels in the *_QA_PIXEL.TIF image with a value not equal to 21824 were accepted as cloud contaminated. Cloud contaminated pixels include both pixels where clouds directly obscure collection by remote sensing

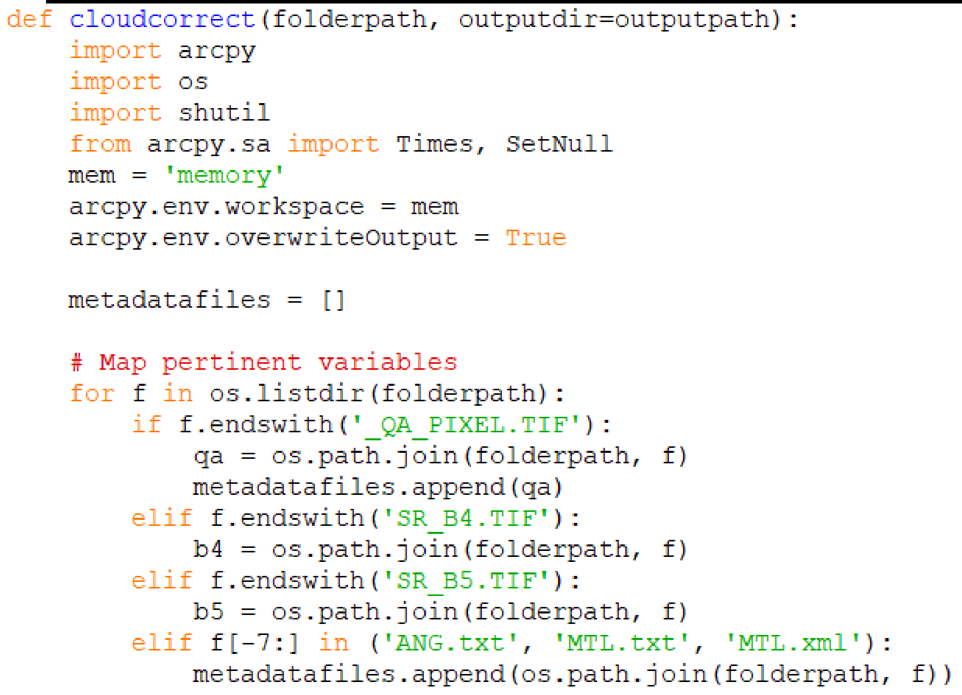
platforms, and those where cloud shadows potentially mask accurate measurements of the reflected energy. Comparisons between *_QA_PIXEL.TIF and the images themselves show that not all cloud contaminated pixels were correctly assessed in the *_QA_PIXEL.TIF. The potential for the presence of obscured pixels in the analysis area necessitates a process that considers pixel characteristics over time.



```
import os
import statistics

# Define input variables
inputpath = r''
outputpath = r'...\CloudCorrectedData'
```

Figure 6: *Import the **os** and **statistics** modules to provide functionality in the execution portion of the code. Define the input path as **inputpath** and the output path as **outputpath**. The "r" before the string apostrophes signifies the input string is in a raw format. The **outputpath** variable includes a suggestion that the output will be the cloud correct data.*

The next sequence of screenshots is tied to defining the **cloudcorrect** function which has several distinct portions.

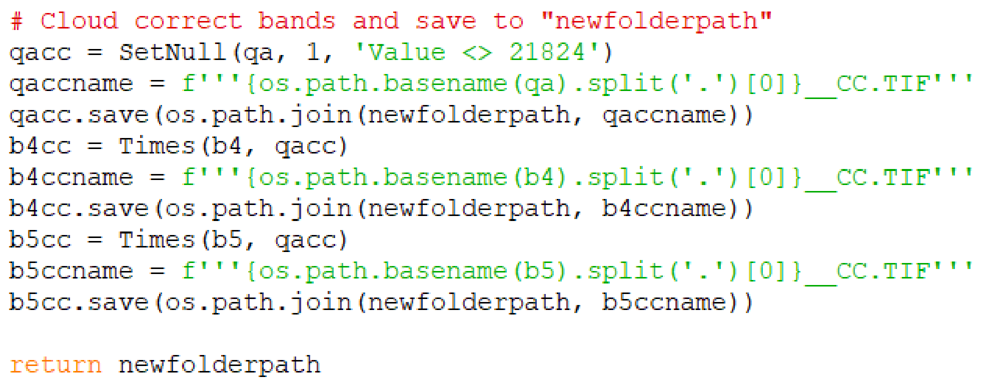


```
def cloudcorrect(folderpath, outputdir=outputpath):
    import arcpy
    import os
    import shutil
    from arcpy.sa import Times, SetNull
    mem = 'memory'
    arcpy.env.workspace = mem
    arcpy.env.overwriteOutput = True

    metadatafiles = []

    # Map pertinent variables
    for f in os.listdir(folderpath):
        if f.endswith('_QA_PIXEL.TIF'):
            qa = os.path.join(folderpath, f)
            metadatafiles.append(qa)
        elif f.endswith('SR_B4.TIF'):
            b4 = os.path.join(folderpath, f)
        elif f.endswith('SR_B5.TIF'):
            b5 = os.path.join(folderpath, f)
        elif f[-7:] in ('ANG.txt', 'MTL.txt', 'MTL.xml'):
            metadatafiles.append(os.path.join(folderpath, f))
```

Figure 7: *This is the beginning of* ***cloudcorrect*** *function definition. Import the* ***arcpy****,* ***os****, and* ***shutil*** *modules in total, and import the* ***Times*** *and* ***SetNull*** *functions from arcpy's Spatial Analyst. This function establishes the* ***mem*** *variable as the environment's workspace to minimize hard drive I/O and post-processing cleanup. The* ***metadatafiles*** *list is used as a container to store the file paths of pertinent metadata files. In the map pertinent variables section, the files within the given* ***folderpath*** *will be iterated and assigned the variable names* ***qa****,* ***b4****, and* ***b5*** *for use during the execution phase of the function. The* ***outputdir*** *input is an optional parameter that will use the* ***outputpath*** *variable defined at the beginning of this script if left undefined.*

```
# Create new folder directories and move "metadatafiles"
foldername = os.path.basename(folderpath)
platform = foldername[:4]
newfoldername = f'{os.path.basename(folderpath)}__CC'
newfolderpath = os.path.join(outputdir, platform, newfoldername)
try:
    os.makedirs(os.path.join(outputdir, platform))
except:
    pass
outfolds = os.listdir(os.path.join(outputdir, platform))
if newfoldername not in outfolds:
    os.makedirs(newfolderpath)
    for meta in metadatafiles:
        shutil.copyfile(
            meta, os.path.join(newfolderpath, os.path.basename(meta))
        )
```

Figure 8: *This is the second portion of the* ***cloudcorrect*** *function definition. As seen in the commented section at the top of the code pictured above, this portion creates new folder directories and copies the files that were previously distinguished in the* ***metadatafiles*** *list. The* ***folderpath*** *function input is used to derive the individual folder's name in the* ***foldername*** *variable. The platform, LC08 or LC09, is derived from the first four characters in the folder's name and is thusly captured in* ***platform*** *variable. The* ***newfoldername*** *and* ***newfolderpath*** *variables are defined to shorten improve the readability of the code/function. The* ***try*** *statement attempts to create the* ***platform*** *directory in the* ***outputdir*** *and ignores any errors that would arise if it already existed. This use of the* ***try*** *statement is a small attempt to make the code faster. The alternative would be to check if it already exists and then create it if it does not. After the* ***platform*** *directory is established a list of its contents are captured in the* ***outfolds*** *variable. If* ***newfoldername*** *is not already in* ***outfolds*** *then* ***newfoldername*** *is created in the* ***platform*** *directory and the files captured in the* ***metadatafiles*** *list are copied to the newly created directory. The files are copied versus just moving them to maintain the integrity of the source data.*



```
# Cloud correct bands and save to "newfolderpath"
qacc = SetNull(qa, 1, 'Value <> 21824')
qaccname = f'''{os.path.basename(qa).split('.')[0]}__CC.TIF'''
qacc.save(os.path.join(newfolderpath, qaccname))
b4cc = Times(b4, qacc)
b4ccname = f'''{os.path.basename(b4).split('.')[0]}__CC.TIF'''
b4cc.save(os.path.join(newfolderpath, b4ccname))
b5cc = Times(b5, qacc)
b5ccname = f'''{os.path.basename(b5).split('.')[0]}__CC.TIF'''
b5cc.save(os.path.join(newfolderpath, b5ccname))

return newfolderpath
```

Figure 9: *This is the third and final portion of the* ***cloudcorrect*** *function definition. Since the folders to store the cloud correction processed data were already created earlier in the* ***cloudcorrect*** *function, this is where the bands are corrected. The first step renders the cloud contaminated pixels as null values using the Spatial Analyst* ***SetNull*** *process. Logically, pixels that do not equal value 21824 are set as null and pixels that equal value 21824 are set as the value 1. The value of 1 was chosen for ease of use because when processing the bands* ***b4*** *and* ***b5****, a Spatial Analyst* ***Times*** *process is used to render the cloud contaminated pixels as null/no data values and maintain the uncontaminated pixels with their original values. The cloud corrected* ***qacc****,* ***b4cc****,* ***b5cc*** *are saved in the previously created* ***newfolderpath*** *directories and the* ***newfolderpath*** *is returned to complete the function.*

The final phase of correcting the cloud contaminated pixels is the execution portion of the script. This includes the logic of navigating to the folders that require processing, calling the **cloudcorrect** function to execute the afore described processes, and a bit of logic to check if there were any folders that did not properly process.

```
# Execute code:
if __name__ == '__main__':
    lenstats = {}
    for path, folders, _ in os.walk(inputpath):
        for f in folders:
            if f.startswith('LC0') and len(f) > 20:
                # Try statement used to account for images that may not
                # process due to significant cloud cover
                try:
                    cloudcorrect(os.path.join(path, f), outputpath)
                except:
                    pass

    # Count the number of files in each of the processed folders and record
    # that number and the folder path in the "lenstats" dictionary
    for path, folders, _ in os.walk(outputpath):
        for f in folders:
            if f.startswith('LC0') and len(f) > 20:
                fpath = os.path.join(path, f)
                lenstats[fpath] = len(os.listdir(fpath))

    # The assumption being made here is that folders will process properly, thus
    # folders that do not have the same number of files as the majority of
    # folders will be deleted
    mode = statistics.mode(list(lenstats.values()))
    for fpath in lenstats:
        if lenstats[fpath] != mode:
            for fi in os.listdir(fpath):
                os.unlink(os.path.join(fpath, fi))
            os.rmdir(fpath)
            print(f'Deleted {os.path.basename(fpath)}')
```

Figure 10: *The execution portion of the script begins with some logic that ensures the execution portion of the code will not run if the functions or variables are imported into another environment. An empty* ***lenstats*** *dictionary, short for length statistics, is created to later store the number of files present in each of the cloud corrected directories. The* ***os.walk*** *process is then called to explore the contents of the* ***inputpath*** *variable and to effectively locate the scene directories. The scene directories are then passed into the* ***cloudcorrect*** *function where they are processed as previously described. The use of a* ***try*** *statement when calling the* ***cloudcorrect*** *function is because some scene directories with an excessive amount of cloud cover may not properly process. Thus, scenes that do not properly process will not throw an error and interrupt further processing. However, this approach necessitates the inclusion of some logic to cleanup those scene directories that were not properly processed. To accomplish this,* ***os.walk*** *is again called but this time on* ***outputpath*** *to explore the number of files contained in each cloud corrected scene directory; these directory statistics are recorded in the* ***lenstats*** *dictionary. The statistical* ***mode*** *is calculated to find the most common number of files throughout the explored directories. The extracted mode is then used to identify folders that are irregular, which are subsequently deleted, and the folder names of the scene directories are displayed using the* ***print*** *function.*

3. The third step in the process concerns collecting statistics from the training data. Prior to beginning this step, a delineation between the training period and evaluation period is required. This study chose to separate the two periods into two distinct folders. Those distinct periods may be referred to as the first five years covering the training period or as the remaining years covering the evaluation period. The training data consists of the first five years of collected Landsat Operational Land Imager (OLI) data (11 February 2013 – 05 July 2018). A set of 500 training points were generated to sample the training data. The training points were randomly generated within the study area with a required separation of at least 300 meters between points. The training points were then manually classified into one of three categories: forest, non-forest, and border. Of the 500 generated points: 322 were classified as forest, 105 were classified as non-forest, and 73 as border. These classifications were based on assessments made by reviewing the very-high-resolution imagery basemap in ArcGIS Pro. The basemap had great coverage of the study area with very little cloud cover, but not all the images used to classify the training points were collected after the end date of the training period. The potential classification of training points as forest which may not have persisted throughout the entirety of the training period may introduce error into the training data statistics. An attempt is made to counter the potential introduction of error during the training statistics collection phase by having a large number of well distributed training data points to accurately represent the land surface phenology trends of the study area.

```
import os
import arcpy
import statistics

# Define variables
basepath = r''
ffy = r'...\CloudCorrectedData\FirstFiveYears'
inpath = os.path.join(basepath, ffy)
outpath = os.path.join(basepath, ffy, '_stats')
samplePoints = os.path.join(
    basepath,
    r'...\P227R065_ForestTrainingPoints_utm21n')
datadic= {}
```

Figure 11: *Import the* ***os***, ***arcpy***, *and* ***statistics*** *modules. The variables in this section primarily relate to defining where something is or where it should go.* ***samplePoints*** *is the path to where the training points are located. The* ***datadic*** *variable is later used for collecting all of the directory information.*

```
# Define functions
def convertdate(day, month, year):
    from datetime import datetime as dd
    fmt = '%Y.%m.%d'
    s = '.'.join([str(year), str(month), str(day)])
    dt = dd.strptime(s, fmt)
    return f'{dt.timetuple().tm_yday}_{year}'
```

Figure 12: *The* ***convertdate*** *function calculates the day of year (doy) number from a date. The* ***convertdate*** *function also captures the* ***year*** *for the doy to account for the potential of having multiple statistical datasets covering the same doy but from different years.*

```
# Input "filename" includes doy, year, vi type and ends with .txt
# Example of properly formatted "filename": 186_2018__SAVI.txt
def wtext(filename, dic):
    if len(dic):
        path = os.path.join(outpath, filename)
        dictext = 'dic = {'
        closedic = '}'
        valuetext = 'values = ['
        numkeys = [int(k) for k in list(dic.keys())]
        numkeys.sort()
        for sk in [str(k) for k in numkeys]:
            dictext += f'{sk}:{dic[sk]};'
            valuetext += f'{dic[sk]},'
        text = f'{dictext[:-1]}{closedic}\n\n{valuetext[:-1]}]'
        try:
            f = open(path, 'w')
            f.write(text)
        finally:
            f.close()
        return path
    else:
        return None
```

***Figure 13:** The **wtext** function writes text to the given input file name **filename** that is written to the afore defined **outpath**. The input **dic** will be later collected using a separate function. The **samplePoint** OIDs are used as the dictionary keys so the collected SAVI values can be tied back to a specific location. All of the values are captured in a list – **valuetext** – so the values can also be accessed separately without including their specific locations. The **numkeys** variable transforms the **dic** keys into integers so they can be sorted to enable an easier human reading experience if required. After sorting, the **numkeys** are transformed back into strings and used to access **dic** entries. These **dic** entries are then used to build **dictext** and **valuetext**. After all **dic** entries are iterated, the generated text is written to **filename** in **outpath**.*

```
def sampleSAVI(doy, b4, b5):
    import os
    import arcpy
    from arcpy.sa import Float, Sample
    mem = 'memory'
    arcpy.env.workspace = mem
    savi_dic = {}
    tablepath = os.path.join(mem, f'_{doy}__SAVI')
    outtext = f'{os.path.basename(tablepath)[1:]}.txt'

    # If the statistics text file doesn't already exist, collect the statistics
    if not os.path.exists(os.path.join(outpath,outtext)):
        # Run SAVI
        L = 0.5
        floatb4 = Float(b4)
        floatb5 = Float(b5)
        savi = (1 + L) * (floatb5 - floatb4) / (floatb5 + floatb4 + L)

        # Sample SAVI
        arcpy.MakeFeatureLayer_management(
            samplePoints, f'forest__{doy}', 'Type = 1' # Type = Forest
            )
        Sample(savi, f'forest__{doy}', tablepath, 'NEAREST', 'OID')
        fields = [field.name for field in arcpy.ListFields(tablepath)]
        sfields = [fields[0], fields[-1]]
        with arcpy.da.SearchCursor(tablepath, sfields) as search:
            for s in search:
                # Checks to see if the sampled value is Null, likely due to
                # cloud correction; record the SAVI value if not Null
                if s[1] != None:
                    savi_dic[str(s[0])] = str(s[1])
        # Write Sample values from SAVI calculation to *.txt
        wtext(outtext, savi_dic)
        arcpy.Delete_management(f'forest__{doy}')
        return os.path.join(path, outtext)
    else:
        return
```

***Figure 14:** The **sampleSAVI** function is defined with the following required inputs: **doy**, **b4**, and **b5**. The input **doy** is used to build unique names in several temporary datasets and to define the **filename** input that will be used in the **wtext** function. The **os** and **arcpy** modules are imported as well as the **Float**, and **Sample** functions from arcpy's Spatial Analyst. This function also establishes the **mem** variable as the environment's workspace to minimize hard drive I/O and*

*post-processing cleanup. The **savi_dic** empty dictionary is created to store the sampled data IDs and values. The **tablepath** path is created as a location to temporarily collect the sampled data. The process then checks if the **outtext** file name exists in the **outpath** directory; if it does not, then the remainder of this block of code is run. Firstly, the SAVI analysis is generated using the input band paths for **b4** and **b5**. Next, a Feature Layer is created from **samplePoints** for all of the points that are classified as forest in the training points dataset. This subset of training points is then used to collect the SAVI values from the point's corresponding location in the **savi** analysis layer. A list of **fields** is then generated from the temporary variable **tablepath**. The pertinent fields from **fields** are then captured in **sfields**, these fields are OID and Value. A search cursor is then executed on **tablepath** and the relevant data is logged in **savi_dic**. The defined **outtext** variable and the **savi_dic** dictionary are then passed to the **wtext** function to write the SAVI values into a recallable text file.*

```
def collectStats(dic, dic_key):
    doy = dic[dic_key][1]
    b4 = os.path.join(dic_key, dic[dic_key][0]['B4'])
    b5 = os.path.join(dic_key, dic[dic_key][0]['B5'])
    sampleSAVI(doy, b4, b5)
    return dic_key
```

Figure 15: *The **collectStats** function is used as bridge to connect all of the cloud corrected data in the training data repository – first five years – to the **sampleSAVI** function. Simply, it translates the information in **datadic** into the inputs required to execute **sampleSAVI**.*

```
# Execute code
if __name__ == '__main__':
    # Populate "datadic" with the required information to run "collectStats"
    for path, folders, files in os.walk(inpath):
        for f in files:
            if f.endswith('_CC.TIF'):
                b = f.split('_')[8]
                if path not in datadic:
                    # Translate day of year
                    year = int(f.split('_')[3][:4])
                    month = int(f.split('_')[3][4:6])
                    day = int(f.split('_')[3][6:])
                    doy = convertdate(day, month, year)
                    # Add to 'dic'
                    datadic[path] = [{b: f}, f'{doy}']
                else:
                    datadic[path][0][b] = f

    for path in datadic:
        print(collectStats(datadic, path))
```

Figure 16: *The execution portion of the script begins with some logic that ensures the execution portion of the code will not run if the functions or variables are imported into another environment. The **inpath** defined at the beginning of the script is explored using **os.walk**. The **os.walk** process iterates through all of the files in **inpath**. If the files end in ***_CC.TIF**, its path is added to **datadic** along with the **year**, **month**, **day**, and **doy** of the imagery's collection metadata if an entry is not already present in **datadic**. Following the construction of **datadic**, **datadic** is*

iterated and ***collectStats*** *is executed on each of* ***datadic****'s entries. This script is written to avoid generating statistics that have already been processed.*

4. The fourth step in the process concerns deriving an initial forest mask and analyzing the evaluation period – remaining years. The training statistics collected in the previous step are called upon to provide a calculation of the values of where the model expects forest pixel SAVI values to fall with in a certain tolerance. The number of standard deviations used for this model is 2.6 which effectively means that 99% of forest pixels should fall within the assessed tolerance range if the collected statistics conform to a normal bell curve. Pixels that are within the forest mask but fall outside of the tolerance range for a given day of year (doy) analysis will be logged in the newly generated forest mask. If the same pixel falls outside of the tolerance range in three consecutive analyses of the pixel, it will be deemed as forest disturbance and removed from the forest mask. To account for cloud contaminated pixels – rendered null during the cloud correction step – null values are ignored during the analysis and the value carried through in the latest forest mask will remain unchanged. Pixels that regain compliance by returning a value within the tolerance range are reset to zero, effectively eliminating any record of having ever been outside the tolerance range in the forest mask. Similarly, a regeneration mask and process analyze to check if any pixels outside of the forest mask have regained tolerance. The regeneration analysis is similar to the forest disturbance analysis but it checks to see if pixels that are not in the forest mask attain 10 consecutive analyses within the tolerance range. Pixels that attain 10 consecutive analyses within the tolerance range are removed from the regeneration mask and added back to the forest mask for subsequent analysis. Both forest disturbance and forest regeneration are logged in feature classes that record the date pixels are removed from the forest mask or removed from the regeneration mask.

```
import os
from time import strftime as t

# Define variables
basepath = r''
ffystats = r'...\CloudCorrectedData\FirstFiveYears\_stats'
fmanpath = r'...\CloudCorrectedData\1_ForestMaskAnalysis'
fmpath = r'...\CloudCorrectedData\2_ForestMasks'
rmanpath = r'...\CloudCorrectedData\3_RegenerationMaskAnalysis'
rmpath = r'...\CloudCorrectedData\4_RegenerationMasks'
dist_fc = r'...\ChangeLog.gdb\LS_227065_DisturbanceLog'
rege_fc = r'...\ChangeLog.gdb\LS_227065_RegenerationLog'
remyears = r'...\CloudCorrectedData\RemainingYears'
```

Figure 17: *Import the* ***os*** *module and* ***strftime*** *from the* ***time*** *module as the variable* ***t****. The* ***basepath*** *variable is defined as a common base path for the other path variables. The* ***ffystats*** *variable is a pointer to where the statistics derived from the training data are located. The* ***fmanpath*** *variable is the location where forest mask analysis will be permanently stored. The* ***fmpath*** *variable is the location where the forest masks will be permanently stored. The* ***rmanpath*** *variable is the location where regeneration mask analysis will be permanently stored. The* ***rmpath*** *variable is the location where the regeneration masks will be permanently stored. The* ***dist_fc*** *variable is a pointer to the feature class that stores polygons for where forest disturbances are recorded and the date they are removed from the forest mask. The* ***rege_fc*** *variable is a pointer to the feature class that stores polygons for where forest regeneration is recorded and the date they are removed from the regeneration mask and add back to the forest mask. The* ***remyears*** *variable is location where the cloud corrected data covering the evaluation period is stored.*

```
# Define functions
def convertdate(day, month, year):
    from datetime import datetime as dd
    fmt = '%Y.%m.%d'
    s = '.'.join([str(year), str(month), str(day)])
    dt = dd.strptime(s, fmt)
    return f'{dt.timetuple().tm_yday}_{year}'
```

Figure 18: *The* ***convertdate*** *function calculates the day of year (doy) number from a date. The* ***convertdate*** *function also captures the* ***year*** *for the doy to account for the potential of having multiple statistical datasets covering the same doy but from different years.*

```
def convertorddate(day, month, year):
    from datetime import datetime as dd
    return f'{dd(year, month, day).toordinal()}'
```

Figure 19: *The* ***convertorddate*** *function calculates the ordinal number from a date. The* ***convertorddate*** *function ensures analysis data is stored chronologically. This function calculates the ordinal number from a given date.*

```
def extract_doy_stats(txtfile):
    import os
    import statistics
    doynum = int(os.path.basename(txtfile).split('_')[0])
    opentxt = open(txtfile, 'r')
    readtxt = opentxt.read()
    opentxt.close()
    valuelis = [float(n) for n in readtxt.split('[')[1][:-1].split(',')]
    stdev = statistics.stdev(valuelis)
    return {str(doynum): [valuelis, stdev]}
```

Figure 20: *The* ***extract_doy_stats*** *function reads a single doy text file that contain the SAVI training values generated in step three. The SAVI training values extracted in step three and their standard deviation are loaded into a dictionary and returned.*

```
def compile_stats():
    import os
    # Build compiled dictionary
    compdic = {}
    stat_path = os.path.join(basepath, ffystats)
    for f in os.listdir(stat_path):
        if f.endswith('.txt') and 'SAVI' in f:
            compdic.update(extract_doy_stats(os.path.join(stat_path, f)))

    # Build sorted list of values
    complis = []
    dickeys = [int(num) for num in compdic.keys()]
    dickeys.sort(reverse = True)
    for key in dickeys:
        for value in compdic[str(key)][0]:
            complis.append((key - 365, value))
            complis.append((key, value))
            complis.append((key + 365, value))
    return complis
```

Figure 21: *The* ***compile_stats*** *function compiles all of the statistics captured over the training period in the* ***comp_dic*** *dictionary and then further transforms the values into the* ***complis*** *list. The* ***complis*** *list format is setup for loading the data into a two-dimensional array where the x-axis is the doy and the y-axis is a sample SAVI value for a single training point. The format of how the training data represents a day of year (doy) in any year, means that the statistics derived during the training period can all be laid on top each other as if they were all from a single year. In an attempt to mitigate inaccurate edge calculations – near doy 1 or doy 365 – the statistics derived from the training data were arranged as if they were three consecutive years. The intent is to provide ample data to a fitting function that will later call the* ***compile_stats*** *function.*

```
def compile_stdev():
    import os
    compdic = {}
    stat_path = os.path.join(basepath, ffystats)
    for f in os.listdir(stat_path):
        if f.endswith('.txt') and 'SAVI' in f:
            compdic.update(extract_doy_stats(os.path.join(stat_path, f)))

    # Build sorted list of values
    stdevlis = []
    dickeys = [int(num) for num in compdic.keys()]
    dickeys.sort(reverse = True)
    for key in dickeys:
        stdevlis.append((key - 365, compdic[str(key)][1]))
        stdevlis.append((key, compdic[str(key)][1]))
        stdevlis.append((key + 365, compdic[str(key)][1]))
    return stdevlis
```

***Figure 22:** The **compile_stdev** function compiles all of the generated standard deviations from the statistics captured over the training period in the **comp_dic** dictionary. Then those standard deviations are further transformed into the **stdevlis** list. Like the **compile_stats** function, the **stdevlis** list format is setup for loading the data into a two-dimensional array where the x-axis is the doy and the y-axis is the standard deviation derived from the SAVI values for a single doy. The **compile_stdev** function also attempts to arrange the standard deviation data in a manner that mitigates inaccurate edge calculations.*

```
def npfunction(input_data, input_polynomial=15):
    import numpy as np
    points = np.array(input_data)
    x = points[:,0]
    y = points[:,1]
    z = np.polyfit(x, y, input_polynomial)
    func = np.poly1d(z)
    return func
```

***Figure 23:** The **npfunction** function uses the base data compiled in the **compile_stats** and **compile_stdev** functions to generate a mathematical formula that fits the data. The mathematical formula used to fit the data enables the ability to derive an estimated mean value for a doy where no data currently exists. Furthermore, nearby doy variance is calculated into any returned value. The **input_polynomial** default value was set at 15 because it showed the least amount of edge variance when transitioning from doy 365 to doy 1. The **npfunction** function import **numpy** as **np**. A **numpy** array is generated from the **input_data** and stored in the **points** variable. The **x**, **y**, and **z** variables are used to fit the algorithm to the training data and create the **func** function which is returned.*

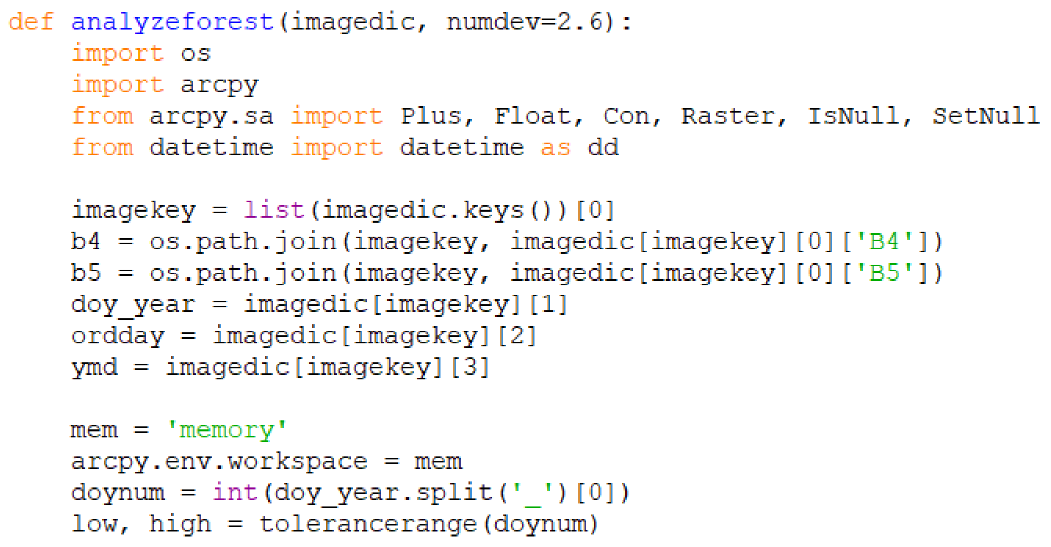
```python
# Extract tolerance range for VIs from training data
def tolerancerange(doynum, numdev=2.6):
    import os
    import numpy as np

    # Compile stats from text files
    statsmapped = compile_stats()
    stdevmapped = compile_stdev()

    # Create polynomial functions
    statsfunc = npfunction(statsmapped)
    stdevfunc = npfunction(stdevmapped)

    # Define range values
    center = statsfunc(doynum)
    stdev = stdevfunc(doynum)
    low = center - (stdev * numdev)
    high = center + (stdev * numdev)

    return low, high
```

Figure 24: *The* ***tolerancerange*** *function inputs the statistics derived from the* ***compile_stats*** *and* ***compile_stdev*** *functions into the* ***npfunction*** *function to derive a tolerance range for a given day of year. The optional* ***numdev*** *parameter is used to stipulate the number of standard deviations to use when computing the tolerance range. As previously described, 2.6 times the standard deviation should identify at least 99% of forest pixels. The* ***tolerancerange*** *function returns the* ***low*** *and* ***high*** *thresholds that define the input doy's tolerance range.*

```python
# Maps the band paths and generates the doy for an image folder
def mapimagefolder(imagefolder):
    import os
    imagedic = {}
    files = os.listdir(imagefolder)
    for f in files:
        if f.endswith('_CC.TIF'):
            band = f.split('_')[8]
            if imagefolder not in imagedic:
                # Translate day of year
                year = int(f.split('_')[3][:4])
                month = int(f.split('_')[3][4:6])
                day = int(f.split('_')[3][6:])
                doy_year = convertdate(day, month, year)
                ordday = convertorddate(day, month, year)
                ymd = f'{year}{str(month).zfill(2)}{str(day).zfill(2)}'
                # Add to 'dic'
                imagedic[imagefolder] = [
                    {band: f},
                    f'{doy_year}',
                    f'{ordday}',
                    f'{ymd}'
                    ]
            else:
                imagedic[imagefolder][0][band] = f
    return imagedic
```

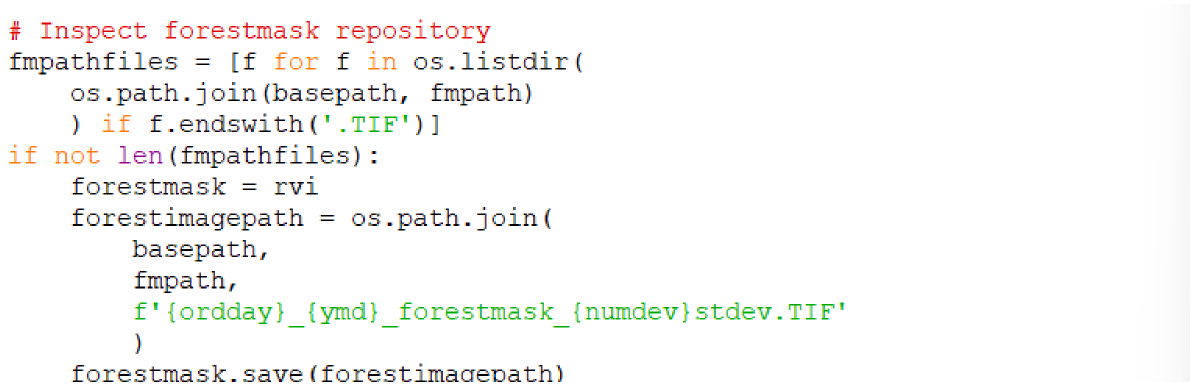
Figure 25: *The* ***mapimagefolder*** *function takes the path of an input folder –* ***imagefolder*** *– and maps the relevant band data paths as well as deriving values to populate the* ***doy_year*** *and* ***ordday*** *variables using the* ***convertdate*** *and* ***convertorddate*** *functions. These values are populated in the* ***imagedic*** *which is returned at the function's conclusion.*

The next sequence of screenshots is tied to defining the **analyzeforest** function which has several distinct portions.



```
def analyzeforest(imagedic, numdev=2.6):
    import os
    import arcpy
    from arcpy.sa import Plus, Float, Con, Raster, IsNull, SetNull
    from datetime import datetime as dd

    imagekey = list(imagedic.keys())[0]
    b4 = os.path.join(imagekey, imagedic[imagekey][0]['B4'])
    b5 = os.path.join(imagekey, imagedic[imagekey][0]['B5'])
    doy_year = imagedic[imagekey][1]
    ordday = imagedic[imagekey][2]
    ymd = imagedic[imagekey][3]

    mem = 'memory'
    arcpy.env.workspace = mem
    doynum = int(doy_year.split('_')[0])
    low, high = tolerancerange(doynum)
```

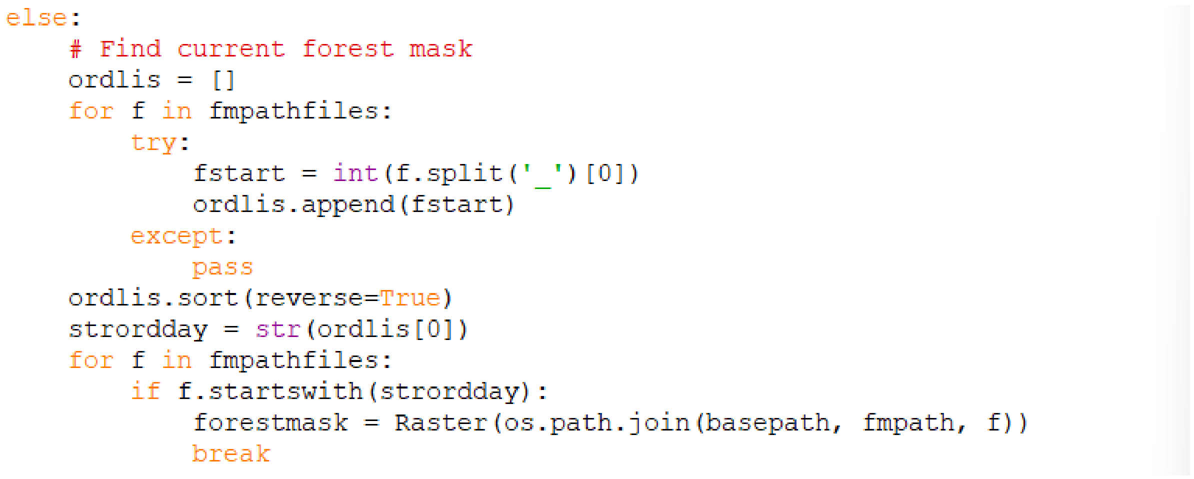
Figure 26: *The* ***analyzeforest*** *function defines a method to analyze the input image scenes in an attempt to understand the widespread pervasiveness of forest disturbances within the study area. Import the* ***os*** *and* ***arcpy*** *modules followed by importing the* ***Plus****,* ***Float****,* ***Reclassify****,* ***RemapValue****,* ***Con****,* ***Raster****,* ***IsNull****, and* ***SetNull*** *functions from arcpy's Spatial Analyst. From the* ***datetime*** *module,* ***datetime*** *is imported as* ***dd*** *for creating a datetime object to record when a pixel is logged as being forest disturbed in the* ***dist_fc*** *feature class. From the* ***imagedic*** *input parameter the* ***imagekey****,* ***b4****,* ***b5****,* ***doy_year****,* ***ordday****, and* ***ymd*** *variables are established. The* ***analyzeforest*** *function establishes the* ***mem*** *variable as the environment's workspace to minimize hard drive I/O and post-processing cleanup. The integer value of the day of year is converted and stored in the* ***doynum*** *variable. The tolerance range to be used during the classifying of the SAVI analysis is computed using the* ***tolerancerange*** *function which provides the low and high values of the tolerance range.*



```python
# Inspect forestmask repository
fmpathfiles = [f for f in os.listdir(
    os.path.join(basepath, fmpath)
    ) if f.endswith('.TIF')]
if not len(fmpathfiles):
    forestmask = rvi
    forestimagepath = os.path.join(
        basepath,
        fmpath,
        f'{ordday}_{ymd}_forestmask_{numdev}stdev.TIF'
        )
    forestmask.save(forestimagepath)
```

Figure 27: *The **fmpathfiles** variables is defined as a list of files present in the **fmpath** directory. The **fmpathfiles** variable is defined to ascertain whether any previous forest masks exist in the **fmpath** directory. If no files exist within the **fmpathfiles** list, then the initial forest mask is analyzed and stored in **fmpath**.*

The remaining blocks of code are analyzed if a forest mask already exists within **fmpath**. Subsequent analyses will call upon the most recent forest mask to define the analysis area and recall which pixels have recently experienced returns outside of the tolerance range.



```python
else:
    # Find current forest mask
    ordlis = []
    for f in fmpathfiles:
        try:
            fstart = int(f.split('_')[0])
            ordlis.append(fstart)
        except:
            pass
    ordlis.sort(reverse=True)
    strordday = str(ordlis[0])
    for f in fmpathfiles:
        if f.startswith(strordday):
            forestmask = Raster(os.path.join(basepath, fmpath, f))
            break
```

Figure 28: *The blocks of code following the **else** statement find the current forest mask and define it as the variable **forestmask**.*

```
# Create SAVI layer
# L introduces a correction factor for the SAVI equation
L = 0.5
floatb4 = Float(b4)
floatb5 = Float(b5)
vi = (1 + L) * (floatb5 - floatb4) / (floatb5 + floatb4 + L)

# "Reclassify-ish"
# Sets values within tolerance range to 0
ridlow = SetNull(vi, vi, f'Value <= {low}')
rvi = SetNull(ridlow, 0, f'Value >= {high}')
```

Figure 29: *The **vi** variable contains the SAVI analysis. The **low** and **high** variables, established when calling the **tolerancerange** function, are used to remove values below the **low** and above the **high** thresholds and store the resulting layer in the **rvi** variable.*

```
# Begin analysis
# Fills null pixels with the value of 1 - outside of tolerance range
analysis_isnull = IsNull(rvi)
analysisimagepath = os.path.join(
    basepath,
    fmanpath,
    f'{ordday}_{ymd}_analysis_{numdev}stdev.TIF'
    )
analysis_isnull.save(analysisimagepath)
```

Figure 30: *The **IsNull** function is called on the **rvi** variable to set a value of one for all null pixels. Then **analysis_isnull** is saved to the **analysisimagepath** in the **fmanpath** directory.*

```
# Capture cloud contaminated pixels in raw image
b4stats = IsNull(Raster(b4))
cloudcon = SetNull(b4stats, 0, 'Value <> 1')
```

Figure 31: *This portion of the code uses the null pixels in the **b4** band to create a layer representing cloud contaminated pixels and stores it in **cloudcon** where the cloud contaminated pixels are valued as zero and the non-contaminated pixels are recorded as null.*

```
# Analyze change
fm = Raster(forestmask)
# Add the forest mask and null corrected analysis together
fmann = Plus(fm, analysis_isnull)
# Identify which pixels haven't changed (Value = 1)
compfmann = fm == fmann
# Reset unchanged pixels to 0
resetunchan = Con(compfmann, 0, fmann, 'Value = 1')
# Capture cloud contaminated pixels from "fm"
ccfmpix = Plus(fm, cloudcon)
# Combine all analyses into a single image
com = Con(b4stats, ccfmpix, resetunchan, 'Value = 1')
```

Figure 32: *This block of code builds the framework of the new forest mask by logically accounting for different potential outcomes and assembling the individual component layers into a single output **com**. The **forestmask** is redesignated as **fm** to make it easier to recall. The **fm** variable is then added to **analysis** and stored in the **fmann** variable. In an attempt to identify unchanged pixels, pixels that the latest analysis analyzed as being within the tolerance range, **fm** and **fmann** pixels are compared on a pixel-by-pixel basis. The pixels that were identified as unchanged were reset with a value of zero and the remaining pixels were assigned their value from the **fmann** variable and stored in **resetunchan**. The cloud contaminated pixels identified in **cloudcon**, valued as zero, are added to **fm** to carryover the value from the last forest mask and stored in **ccfmpix**. Finally, the analyses are combined into the **com** variable using the **Con** function.*

```
# Log pixels that have three consecutive disturbances in "dist_fc"
if com.maximum > 2:
    imagedate = dd(int(ymd[:4]), int(ymd[4:6]), int(ymd[6:]))
    tempfc = os.path.join(mem, f'_{ordday}')
    rempix = SetNull(com, 0, 'Value <> 3')
    arcpy.conversion.RasterToPolygon(rempix, tempfc, 'NO_SIMPLIFY')
    arcpy.management.AddField(tempfc, 'Recorded', 'DATEONLY')
    with arcpy.da.UpdateCursor(tempfc, 'Recorded') as update:
        for u in update:
            u[0] = imagedate
            update.updateRow(u)
    arcpy.management.Append(tempfc,
                            os.path.join(basepath, dist_fc),
                            'NO_TEST')
```

Figure 33: *This portion deals with identifying and logging pixels that have three consecutive SAVI values outside of the tolerance range. First, the maximum of the **com** variable is tested to ascertain whether values of three or higher are present in the **com** layer. When there are values of three present in **com** the **imagedate** is extracted by calling the previously defined variable **ymd**. The **imagedate** is needed to log the date in **dist_fc** when pixels were removed from the forest mask. A temporary feature class, **tempfc**, is created to convert the pixels into polygons. The temporary layer **rempix** is created to contain only the pixels with a value of three while the other pixels present in the forest mask are set to null using the **SetNull** function. After the pixels are converted to polygons in the feature class **tempfc**, a new field is added so the date of removal can be logged. The date of removal is then logged in **tempfc** and the contents of **tempfc** are appended to **dist_fc**.*

```
    # Remove pixels that have three consecutive disturbances from "com"
    newforestmask = SetNull(com, com, 'Value = 3')
else:# if com.maximum > 2:
    newforestmask = com
```

Figure 34: *The variable **newforestmask** is then created by setting null values for pixels with a value equal to three, which removes those pixels from the forest mask and renders them outside of the analysis area for subsequent analysis. The **else** statement is used in case no pixels have a value higher than 2; in this case, **newforestmask** is a copy of **com**.*

```
    # Save and return "newforestmask"
    forestimagepath = os.path.join(
        basepath,
        fmpath,
        f'{ordday}_{ymd}_forestmask_{numdev}stdev.TIF'
        )
    newforestmask.save(forestimagepath)
    return newforestmask
```

***Figure 35:** The final portion of the **analyzeforest** function ends with defining the a path and name for the new forest mask and saving **newforestmask** to **forestimagepath**. The **newforestmask** is then returned to complete the function.*

The next portion of the code has several distinct portions tied to defining the **analyzeregeneration** function. The **analyzeregeneration** function is generally the same process as **analyzeforest** with a few changes. The differences in the two functions will be examined but portions that are the same will simply be identified as such.

```
def analyzeregeneration(imagedic, numdev=2.6):
    import os
    import arcpy
    from arcpy.sa import Plus, Float, Con, Raster, IsNull, SetNull
    from datetime import datetime as dd

    imagekey = list(imagedic.keys())[0]
    b4 = os.path.join(imagekey, imagedic[imagekey][0]['B4'])
    b5 = os.path.join(imagekey, imagedic[imagekey][0]['B5'])
    doy_year = imagedic[imagekey][1]
    ordday = imagedic[imagekey][2]
    ymd = imagedic[imagekey][3]

    mem = 'memory'
    arcpy.env.workspace = mem
    doynum = int(doy_year.split('_')[0])
    low, high = tolerancerange(doynum)
```

***Figure 36:** The **analyzeregeneration** function begins the same as **analyzeforest** is described in figure 26.*

```
    # Inspect forestmask repository
    fmpathfiles = [f for f in os.listdir(
        os.path.join(basepath, fmpath)
        ) if f.endswith('.TIF')]
```

***Figure 37:** The **fmpathfiles** variables is defined as a list of files present in the **fmpath** directory.*

```
# Find current forest mask
ordlis = []
for f in fmpathfiles:
    try:
        fstart = int(f.split('_')[0])
        ordlis.append(fstart)
    except:
        pass
ordlis.sort(reverse=True)
strordday = str(ordlis[0])
for f in fmpathfiles:
    if f.startswith(strordday):
        curfmpath = os.path.join(basepath, fmpath, f)
        forestmask = Raster(curfmpath)
        break
```

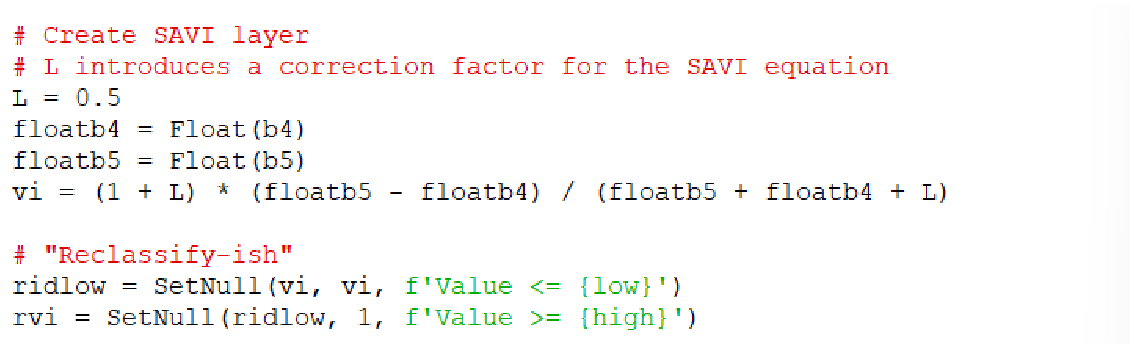
Figure 38: *Is the same code described in* ***figure 28*** *of the* ***analyzeforest*** *function.*

```
# Change the Null values in forestmask to 0 and set rest to Null
fm_isnull = IsNull(forestmask)
fm_nullas0 = SetNull(fm_isnull, 0,'Value = 0')
```

Figure 39: *This block of code defines the regeneration mask analysis area by converting the null pixels in the most recent* ***forestmask*** *to zero and nullifying those that defined the forest mask. Both* ***fm_isnull*** *and* ***fm_nullas0*** *are called upon later.*

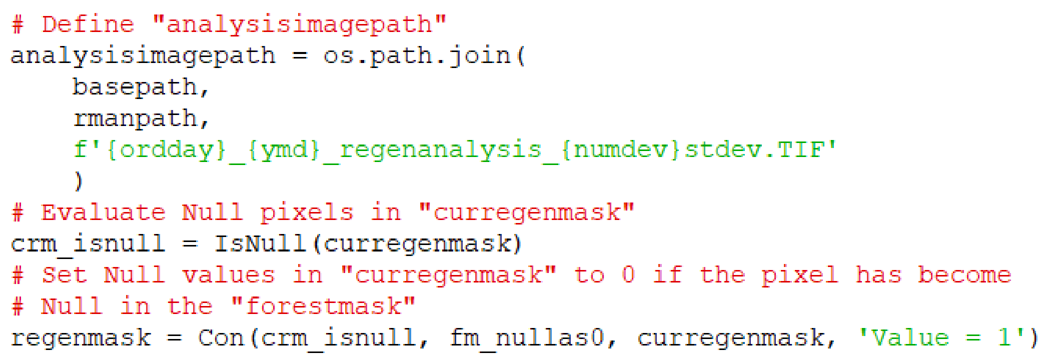
```
# Inspect regenmask repository
rmpathfiles = [f for f in os.listdir(
    os.path.join(basepath, rmpath)
    ) if f.endswith('.TIF')]
regenimagepath = os.path.join(
    basepath,
    rmpath,
    f'{ordday}_{ymd}_regenmask_{numdev}stdev.TIF'
    )
if not len(rmpathfiles):
    curregenmask = fm_nullas0
    curregenmask.save(regenimagepath)
else:
    # Find current regen mask
    ordlis = []
    for f in rmpathfiles:
        try:
            fstart = int(f.split('_')[0])
            ordlis.append(fstart)
        except:
            pass
    ordlis.sort(reverse=True)
    strordday = str(ordlis[0])
    for f in rmpathfiles:
        if f.startswith(strordday):
            curregenmask = Raster(os.path.join(basepath, rmpath, f))
            break
```

Figure 40: *This block of code is very similar to the process used to locate the most recent forest mask but instead finds the most recent regeneration mask, if one exists, and assigns it to variable **curregenmask**. If a regeneration mask does not yet exist, the regeneration mask generated from the previously assigned **fm_nullas0** variable is saved to **regenimagepath**.*



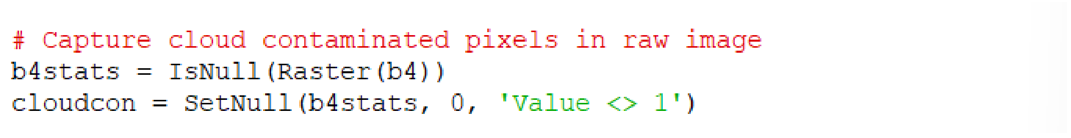
```
# Create SAVI layer
# L introduces a correction factor for the SAVI equation
L = 0.5
floatb4 = Float(b4)
floatb5 = Float(b5)
vi = (1 + L) * (floatb5 - floatb4) / (floatb5 + floatb4 + L)

# "Reclassify-ish"
ridlow = SetNull(vi, vi, f'Value <= {low}')
rvi = SetNull(ridlow, 1, f'Value >= {high}')
```

Figure 41: *Like the process described in **figure 29**, this block executes the SAVI analysis and uses the values provided by the **tolerancerange** function to classify adherent pixels. However, the adherent pixels in variable **rvi** are classified as one.*



```
# Define "analysisimagepath"
analysisimagepath = os.path.join(
    basepath,
    rmanpath,
    f'{ordday}_{ymd}_regenanalysis_{numdev}stdev.TIF'
    )
# Evaluate Null pixels in "curregenmask"
crm_isnull = IsNull(curregenmask)
# Set Null values in "curregenmask" to 0 if the pixel has become
# Null in the "forestmask"
regenmask = Con(crm_isnull, fm_nullas0, curregenmask, 'Value = 1')
```

Figure 42: *Defines the output path for the regeneration analysis as **analysisimagepath**. The null pixels in **currengenmask** are identified and used to create the **regenmask** variable.*



```
# Capture cloud contaminated pixels in raw image
b4stats = IsNull(Raster(b4))
cloudcon = SetNull(b4stats, 0, 'Value <> 1')
```

Figure 43: *Same as **analyzeforest** briefly described in **figure 31**.*

```
# Analyze change
rm = Raster(regenmask)
# Add the regen mask and null corrected analysis together
rmann = Plus(rm, analysis)
# Identify which pixels haven't changed (Value = 1)
comprmann = rm == rmann
# Reset unchanged pixels to 0
resetunchan = Con(comprmann, 0, rmann, 'Value = 1')
# Capture cloud contaminated pixels from "rm"
ccrmpix = Plus(rm, cloudcon)
# Combine all analyses into a single image
com = Con(b4stats, ccrmpix, resetunchan, 'Value = 1')
```

Figure 44: *Same as* ***analyzeforest*** *described in* ***figure 32*** *but the analysis is combined to generate an intermediate step of the regeneration mask.*

```
# Log pixels that have ten successive returns in tolerance
# in "rege_fc"
if com.maximum > 9:
    imagedate = dd(int(ymd[:4]), int(ymd[4:6]), int(ymd[6:]))
    tempfc = os.path.join(mem, f'r_{ordday}')
    rempix = SetNull(com, 0, 'Value <> 10')
    arcpy.conversion.RasterToPolygon(rempix, tempfc,
                                     'NO_SIMPLIFY')
    arcpy.management.AddField(tempfc, 'Recorded', 'DATEONLY')
    with arcpy.da.UpdateCursor(tempfc, 'Recorded') as update:
        for u in update:
            u[0] = imagedate
            update.updateRow(u)
    arcpy.management.Append(tempfc,
                            os.path.join(basepath, rege_fc),
                            'NO_TEST')
```

Figure 45: *If the variable* ***com*** *has pixels with a value that exceeds nine (the values should never be higher than teh), then the following steps are taken. This block of code is the same as in* ***figure 33*** *but it converts and adds the pixels with a value of ten to* ***rege_fc****.*

```
        # Remove pixels that have ten successive in tolerance
        # returns from "com"
        newregenmask = SetNull(com, com, 'Value = 10')

        # If pixels meet threshold for reintegration into the
        # forestmask, add logic here to update forestmask
        rm_10 = SetNull(com, 0, 'Value <> 10')
        newforestmask = Con(
            fm_isnull, rm_10, forestmask, 'Value = 1')

        # Have to be able to overwrite forestmask
        arcpy.env.overwriteOutput = True
        newforestmask.save(curfmpath)
        arcpy.env.overwriteOutput = False

    else:#if com.maximum > 9:
        newregenmask = com
```

Figure 46: *After converting the pixels with a value of ten to polygons, those pixels are then set to null and stored in* ***newregenmask****. Then the* ***com*** *layer is once more called but this time it is used to convert the pixels' value that were ten to zero so they can be readded into the forest mask for reassessment using* ***analyzeforest****. The amended* ***newforestmask*** *is then saved to the* ***curfmpath*** *directory. The* ***else*** *statement at the end is called when there are not any pixels with a value exceeding nine; in that case,* ***newregenmask*** *is constructed directly from the* ***com*** *layer.*

```
    # Save and return "newregenmask"
    if not newregenmask.isEmpty():
        newregenmask.save(regenimagepath)
        return newregenmask
    else:
        return
else:#if not rvi.isEmpty():
    return
```

Figure 47: *If* ***newregenmask*** *is not an empty layer, it is saved to the* ***regenimagepath*** *directory and returned to complete the function.*

After defining all the functions for the fourth step in the process, the analysis can commence. However, the initial forest mask must first be defined before iterating through the evaluation period image folders.

```
# Execute code
if __name__ == '__main__':
    print(f'''Start of processing:\t{t('%X')}''')
    individual = False
    if individual:
        imagefolder = r''
        imagedic = mapimagefolder(imagefolder)
        analyzeforest(imagedic, 2.6)
        print(f'''Finished processing:\t{t('%X')}''')
    else:
        datedic = {}
        for path, folders, files in os.walk(os.path.join(basepath, remyears)):
            if len(folders):
                for folder in folders:
                    if len(folder) > 20:
                        foldsplit = folder.split('_')
                        if foldsplit[0] in ('LC08', 'LC09'):
                            datedic[foldsplit[3]] = os.path.join(path, folder)
        datelis = [d for d in datedic]
        datelis.sort()

        for strdate in datelis:
            print(f'''Start processing of {strdate}:\t{t('%X')}''')
            imagedic = mapimagefolder(datedic[strdate])
            analyzeforest(imagedic, 2.6)
            analyzeregeneration(imagedic, 2.6)
            print(f'''\tCompleted processing of {strdate}:\t{t('%X')}''')
```

Figure 48: *The execution portion of the script begins with some logic that ensures the execution portion of the code will not run if the functions or variables are imported into another environment. To keep track of the progress of the code, several statements using the **print** function have been added. If the user plans to assess an individual image folder – one set of images for a single date – the **individual** variable is defined and set to **True**. The most likely usage for processing an individual image folder is to establish an initial forest mask. In this scenario, the path to folder that will establish the forest mask is defined as **imagefolder** and used to derive **imagedic** using the **mapimagefolder** function. The **analyzeforest** function is then called to create the initial forest mask.*

*When an initial forest mask is already established, the **individual** variable should be set to **False** so the evaluation period datasets can be iteratively analyzed. To ensure all image folders located in the **remyears** directory are processed in order of image date, the **os.walk** function iterates through the subdirectories of **remyears** and builds **datedic** where a dictionary key is the date the image was collected and its associated value is the image directory's path. After **datedic** contains all of the necessary data, the **datelis** list is constructed from the **datedic** dictionary keys and subsequently sorted so the dates stored within **datelis** are in chronological order.*

*The dates present within **datelis** are then chronologically iterated and used to extract their corresponding image directory's path. The image directory's path is used as the input to **mapimagefolder** which is stored in the **imagedic** variable. The **imagedic** variable is the input for both the **analyzeforest** and **analyzeregenertation** functions. A **print** statement is used to confirm the completion of processing for each image directory.*