Overview

Tristan Dyer

In this assignment, we'll cover the basics of using Python in GRASS GIS. We'll write Python scripts that will import and merge lidar point clouds, interpolate lidar points clouds in order to create a DSM, calculate map differences in an automated way, generate publishable HTML reports, and plot terrain profiles.

In each section of this assignment, one or more python script files are provided, as shown in the navigation sidebar. Scripts that rely on other scripts for code (such as `import_merge.py`) will need to be located in the same directory as the scripts on which they rely. The following `assignment7.py` script can be used to run through the entire assignment at once (note that if you'd like to rerun the script, certain maps will need to be removed first, as noted in subsequent sections).

All of the scripts used in this assignment can also be downloaded here:

Assignment 7 Scripts

The easiest way to run these scripts is to use the command line window that was opened when GRASS GIS was started. This will give python access to the grass.script module by default. After downloading and unzipping the scripts, type the following into the command window to run the assignment:

```
cd /path/to/scripts/directory
python assignment7.py
```

In the `assignment7.py` script, we can see that the first 5 lines are simply importing the functions we've declared in the files that you will see in the following section. In order for Python to find these scripts, the files all need to be located in the same directory. You'll also notice that after you run the script, a number of .pyc files will appear. These are the compiled versions of our code and can be safely deleted (they will be regenerated every time they are imported).

After the imports, we have two variables which can be set to whatever you like. The `workingDir` variable is the directory in which all data will be saved. The `lidar_dsm` variable is the name that will be given to the LIDAR derived DSM we'll be creating in this assignment.

Finally, as the comments show, each of the next sections runs the code described by the following sections.

assignment7.py

```
from import_merge import *
from interpolate_dsm import *
from compute_differences import *
from generate_report import *
from terrain_profiles import *

# Variables
workingDir = "/home/tristan/Desktop/assignment7/"
lidar_dsm = 'mid_pines_lidar_dsm'

# Part 1 - Import and merge
merged_las_vector = import_merge( workingDir )

# Part 2 - DSM interpolation with parallelization
interpolate_dsm( merged_las_vector, lidar_dsm )

# Part 3 - Working with large amount of maps
rasters = ['2015_06_20_pix4d_11GCP_dsm', '2015_06_20_DSM_Trimble_11GCP',
           '2015_06_20_DSM_agi_11GCP']
compute_differences( rasters )

# Part 4 - Generate report
generate_report( workingDir )

# Part 5 - Terrain profiles
rasters = ['2015_06_20_pix4d_11GCP_dsm', '2015_06_20_DSM_Trimble_11GCP',
           '2015_06_20_DSM_agi_11GCP', lidar_dsm]
coordinates = [[637160.446919, 219373.236976,
                637105.693795, 219392.416036,
                637072.439779, 219400.613538],
               [636723.722784, 219347.123879,
                637136.495982, 219383.20853],
               [637065.206795, 219731.733448,
                637220.106758, 219475.62044,
                637253.551068, 219309.279002],
               [636955.192616, 219495.863049,
                637001.838628, 219397.290345]]
csv = ['default.csv', 'south_field.csv', 'road.csv', 'tree.csv']
terrain_profiles( rasters, coordinates, csv )
```

Download and Merge LAS Data

The first task is to use python to download LAS data from the internet and merge it into a single vector in GRASS GIS. The function `import_merge()` will perform all of these steps and return the name of the vector in GRASS (default is `merged_points`). Note that the functions download_data(), `transform_coordinates()`, and `las_to_vector()` are each defined in a separate file, and can be seen in the following scripts. Also note that a few of these scripts have the option to be run directly from the command line using command line arguments as inputs.

import_merge.py

```
# Python imports
import glob
import os

# GRASS imports
import grass.script as gscript
from grass.pygrass.modules.grid import GridModule

# Script imports
from download_data import *
from transform_coordinates import *
from las_to_vector import *

# Downloads, imports, and merges las files into a GRASS vector
# - workingDir: directory in which to download download
# - merged_las_vector (optional): name to give merged vector
def import_merge( workingDir, merged_las_vector = 'merged_points' ):

    # Make sure the working directory exists
    if not os.path.exists( workingDir ):

        os.mkdir( workingDir )

    # Move to the working directory
    os.chdir( workingDir )

    # Check for required las files
    if len( glob.glob( '*.las' ) ) == 0:

        # There aren't any las files, so download them
        download_data( './' )

        # For each las file, transform to state plane meters
        for f in glob.glob( '*.las' ):
            transform_coordinates( f )

    # Set the region
    gscript.run_command( 'g.region',
        n=219637,
        s=219254,
        w=636730,
        e=637193,
        res=1,
        flags='p')

    # Import transformed las files into single vector
    lasfiles = glob.glob( '*_spm.las' )
    las_to_vector( lasfiles, merged_las_vector)

    # Return the name of the merged DEM
    return merged_las_vector
```

download_data.py

```
import sys
import os
import urllib

# Downloads required .las files to the given directory
def download_data( directory ):

    # Make sure a valid path was requested
    if os.path.exists( directory ):

        # The base url
        base_url = 'http://fatra.cnr.ncsu.edu/lidar/'

        # The list of las files we want to download
        files = [ '0791_005.las',
                  '0791_011.las',
                  '0792_017.las',
                  '0782_020.las',
                  '0781_008.las' ]

        # For each file that we want to download...
        for f in files:

            # Let the user know which one we're downloading
            print "Retrieving " + f

            # Request the file by combining the base url with the file name
            urllib.urlretrieve( base_url + f, directory + os.path.sep + f )

    else:

        print "Invalid path."

# If running this script from the command-line, pass in a directory
# as an argument to download the required .las files to that directory
if __name__ == "__main__":

    if len( sys.argv ) == 2:

        download_data( sys.argv[1] )

    else:

        print "Usage: python download_data.py [path to save directory]"
```

transform_coordinates.py

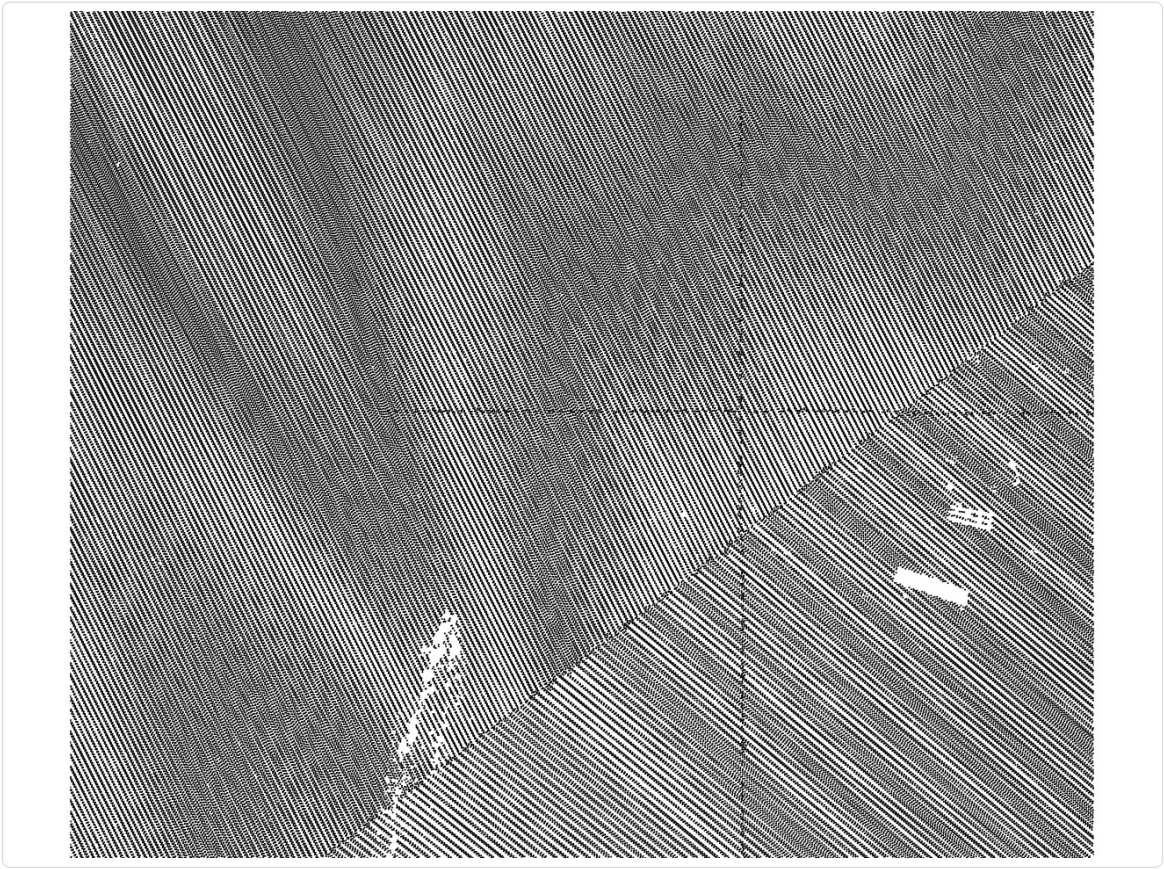
```
import os
from subprocess import call

# Transforms .las file coordinates from EPSG:2264 to EPSG:3358
# using las2las and saves new file with _spm suffix
def transform_coordinates( lasfile ):

    # Check that file exists and is a .las file
    if os.path.isfile( lasfile ) and lasfile.endswith( '.las' ):

        # Split filename and extension
        name, ext = os.path.splitext( lasfile )

        print "Transforming coordinates: " + lasfile

        # Call las2las
        call( ['las2las',
               '--a_srs=EPSG:2264', '--t_srs=EPSG:3358',
               '-i', lasfile,
               '-o', name + '_spm' + ext ] )

# If running this script from the command-line, pass in a list of
# files to transform to NCSP
if __name__ == "__main__":

    if len( sys.argv ) > 1:

        for f in sys.argv:

            transform_coordinates( f )

    else:

        print "Usage: python transform_coordinates.py [file1.las] [file2.las] ..
. [filen.las]"
```

las_to_vector.py

```
import grass.script as gscript

# Imports multiple las files to GRASS and combines
# them into a single vector named vectorName
#  - lasFiles: list of las files
#  - vectorName: name for combined vector in GRASS
def las_to_vector( lasFiles, vectorName ):

    # Get the current region
    region = gscript.region()

    # List of vectors
    vectors = []
```

```
    # Loop through files
    for f in lasFiles:

        # Get the bounding box of the point cloud
        bbox = gscript.read_command( 'r.in.lidar',
            input=f,
            output='foo',
            flags='go'
        ).strip()

        # Parse the bounding box
        bbox = gscript.parse_key_val( bbox,
            vsep=' ',
            val_type=float
        )

        # Ensure that the bounding box of the file overlaps the region
        if ( bbox['n'] < region['s'] or bbox['s'] > region['n'] or
             bbox['e'] < region['w'] or bbox['w'] > region['e']):

            gscript.info( 'Skipping tile %s' % f );
            continue

        # Import the las file as a vector
        name = 'tile_' + f.rsplit( '.', 1 )[0]
        vectors.append( name )

        gscript.run_command( 'v.in.lidar',
            input=f,
            output=name,
            flags='rto',
            class_filter=2
        )

    # Combine all vectors into single vector
    gscript.run_command( 'v.patch',
        input=vectors,
        output=vectorName,
        flags='b',
        overwrite=True
    )

    # Remove the individual vectors
    gscript.run_command( 'g.remove',
        type='vector',
        name=vectors,
        flags='f'
    )
```

Upon successfully running the `interpolate_dsm()` function, we'll now have a vector in our current mapset that looks like this:



Merged point cloud vector

DSM Interpolation in Parallel

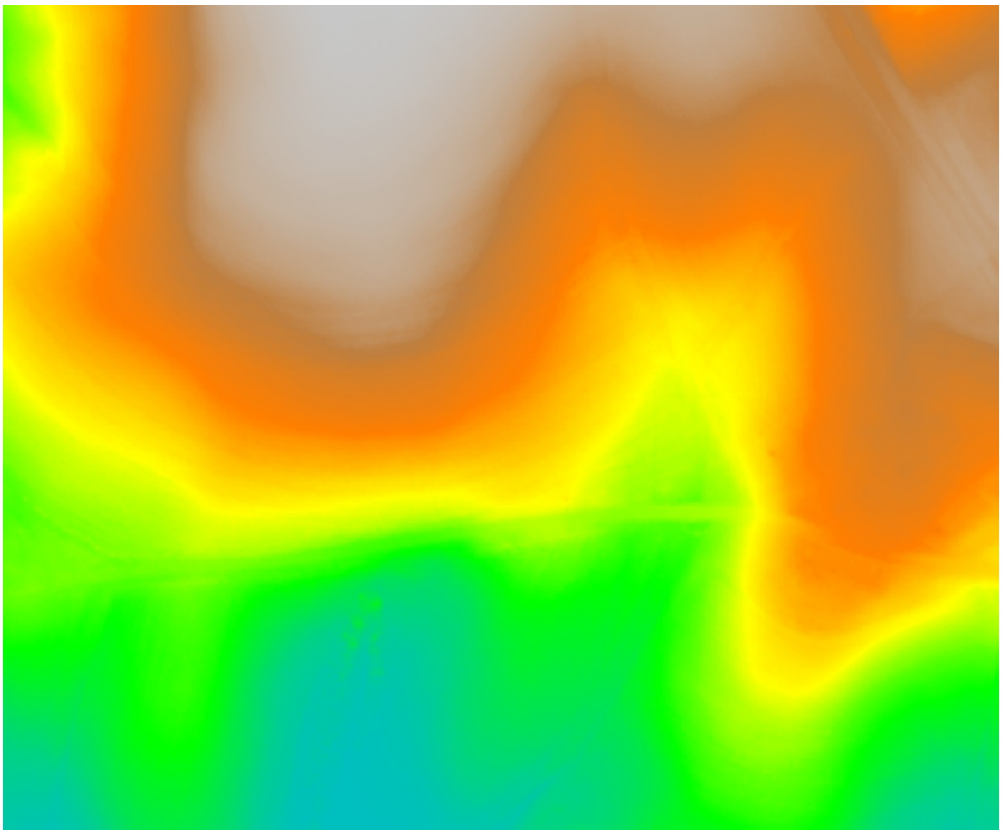
In this section, we'll interpolate a raster surface using the vector we just created by merging las tiles. Because `v.surf.rst` can take some time to run, it is desirable to run it in parallel, a process that can be accomplished using `GridModule`.

> **Note:** I was only able to get parallelization to work using OSGeo Live on the VCL. According to the assignment page, it does not work on Windows, and on my Ubuntu 14.04 machine, it completely uses up all 4 cores (8 threads) and never finishes.

`interpolate_dsm.py`

```
# GRASS imports
import grass.script as gscript
from grass.pygrass.modules.grid import GridModule

# Generate an interpolated DSM from a vector
# - vector_name: name of the vector in the current mapset
# - lidar_dsm (optional): name to give the resulting DSM
# - parallel (optional): Whether or not to run in parallel
def interpolate_dsm( vector_name, lidar_dsm = 'elevation_dsm', parallel = False
):

    # Get the current region
    region = gscript.region()

    # Calculate the width and height required to divide the region
    # into four equal sized tiles
    width = region['cols'] / 2 + 1
    height = region['rows'] / 2 + 1

    # Create the GridModule
    grid = GridModule( 'v.surf.rst',
        debug=parallel==False, width=width, height=height,
        overlap=10, processes=4, npmin=100,
        input=vector_name, elevation=lidar_dsm,
        tension=30, smooth=1, overwrite=True
    )

    # Run the GridModule
    grid.run()

    # Return the name of the new DSM
    return lidar_dsm
```

After running this python function on the merged vector we created in the previous section, we get the following DSM of the Mid-Pines area.



Merged LIDAR Derived DSM

Calculating Map Differences

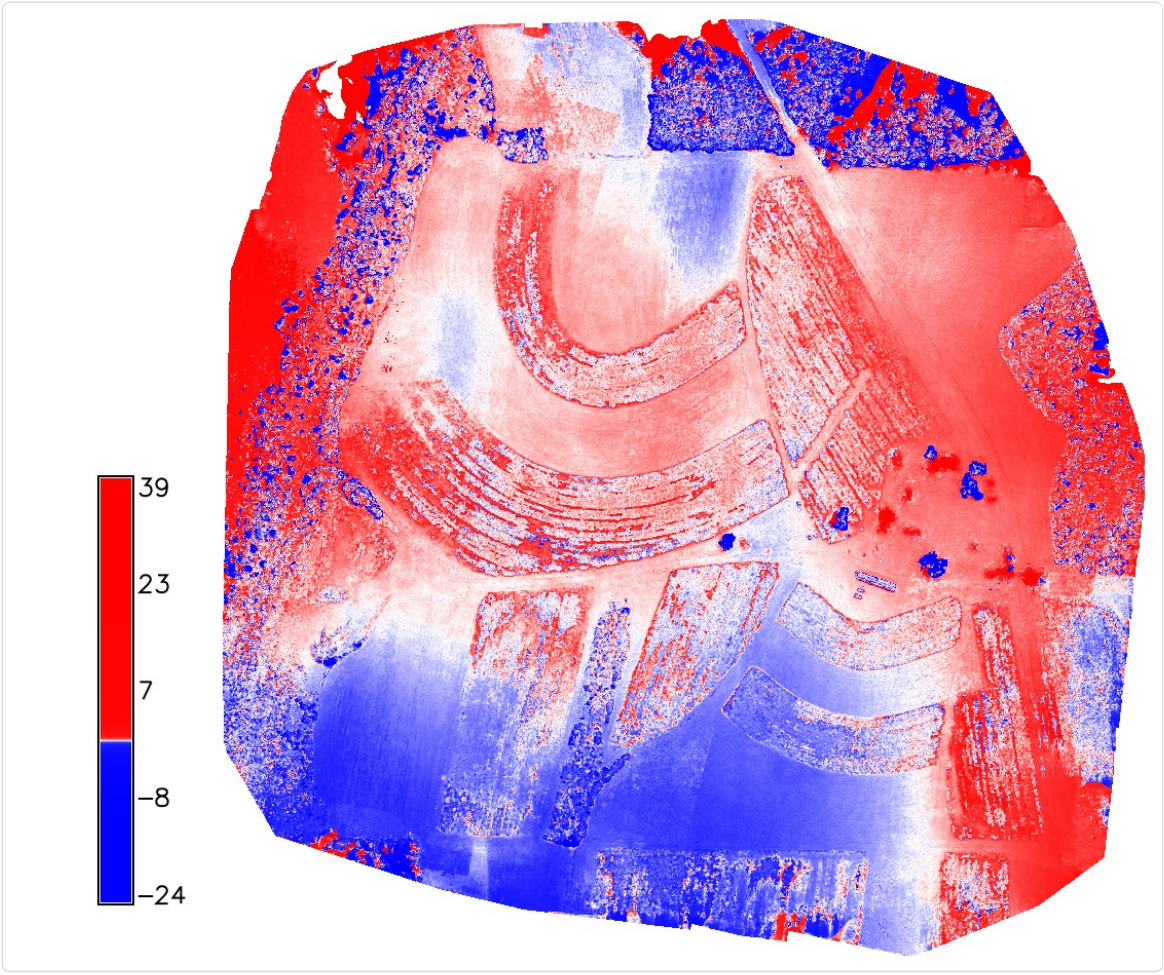
> Before re-running this code from `assignment7.py`, execute the following to remove the rasters that are created in this section:
>
> **`g.remove`** `type=raster pattern="diff_*" -f`

In this section, we wish to calculate the differences between all possible combinations of a list of rasters. The function `compute_differences()` uses a nested for loop to consider every possible combination (ignoring combinations of maps with themselves).

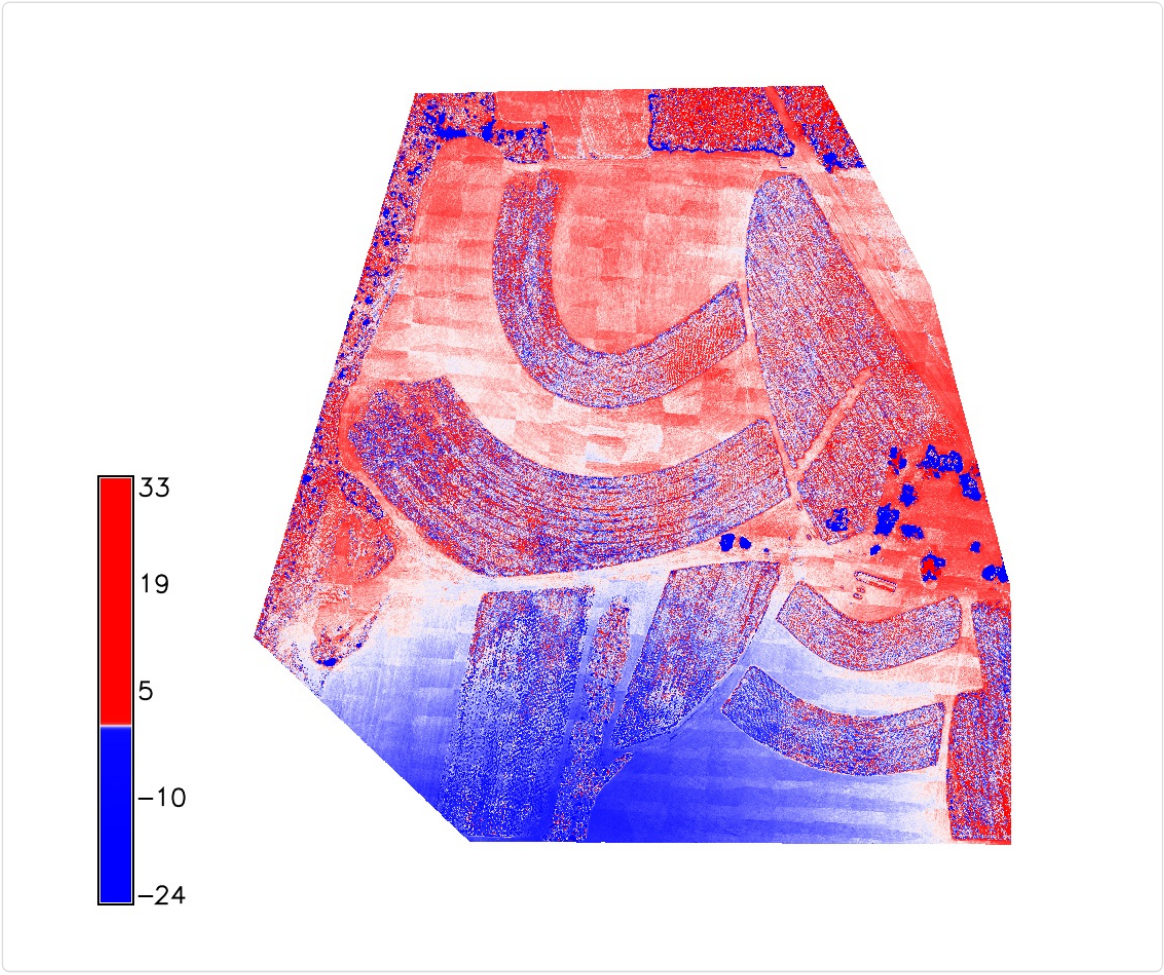
`compute_differences.py`

```
import grass.script as gscript

# Computes the differences between every
# combination of rasters in the given list
def compute_differences( raster_list ):

    # Set the region to the first raster in the list
    gscript.run_command( 'g.region', rast=raster_list[0] )

    # Outer loop
    for a in raster_list:

        # Inner loop
        for b in raster_list:

            # No need to calculate difference between a raster and itself
            if a == b:

                continue

            # Create the name for the difference raster
            difference = 'diff_{ra}_{rb}'.format( ra = a.replace( '@', '_' ),
                                                  rb = b.replace( '@', '_' ))

            # Compute differences using r.mapcalc
            gscript.mapcalc( '{diff} = {ra} - {rb}'.format( ra = a,
                                                            rb = b,
                                                            diff = difference))

            # Set the color table
            gscript.run_command( 'r.colors',
                map = difference,
                color = 'differences',
                flags = 'e'
            )
```

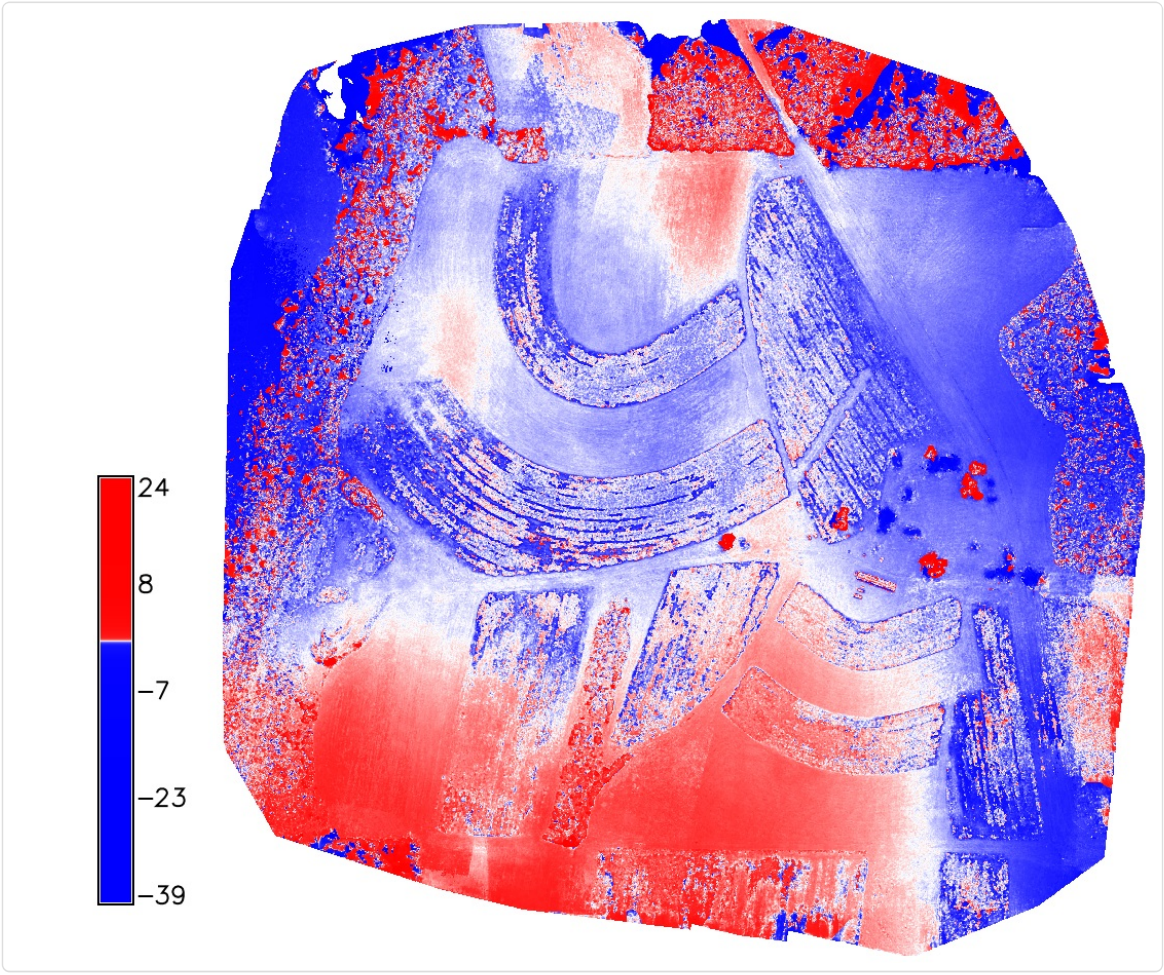
After running this function, we'll have 6 new rasters:



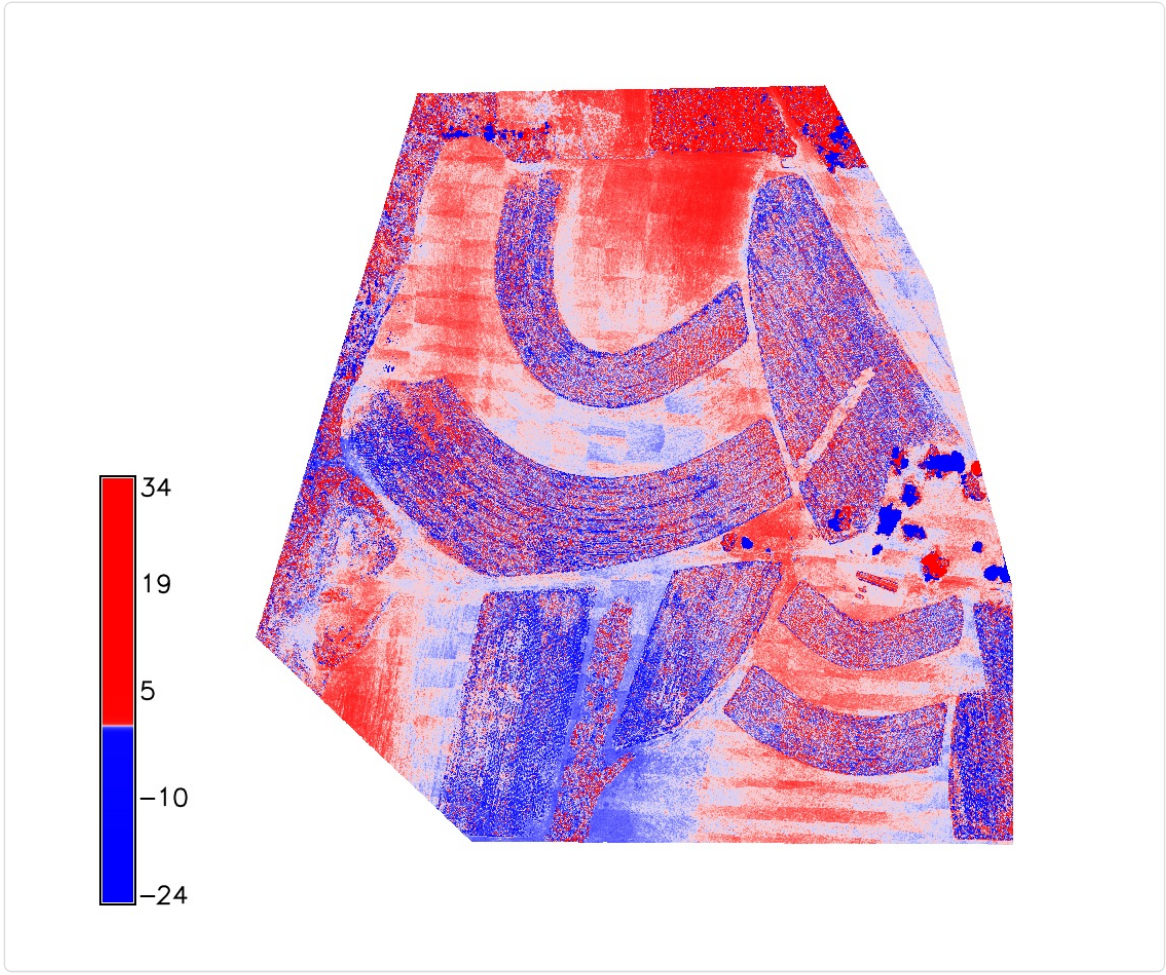
Agisoft - Pix4D



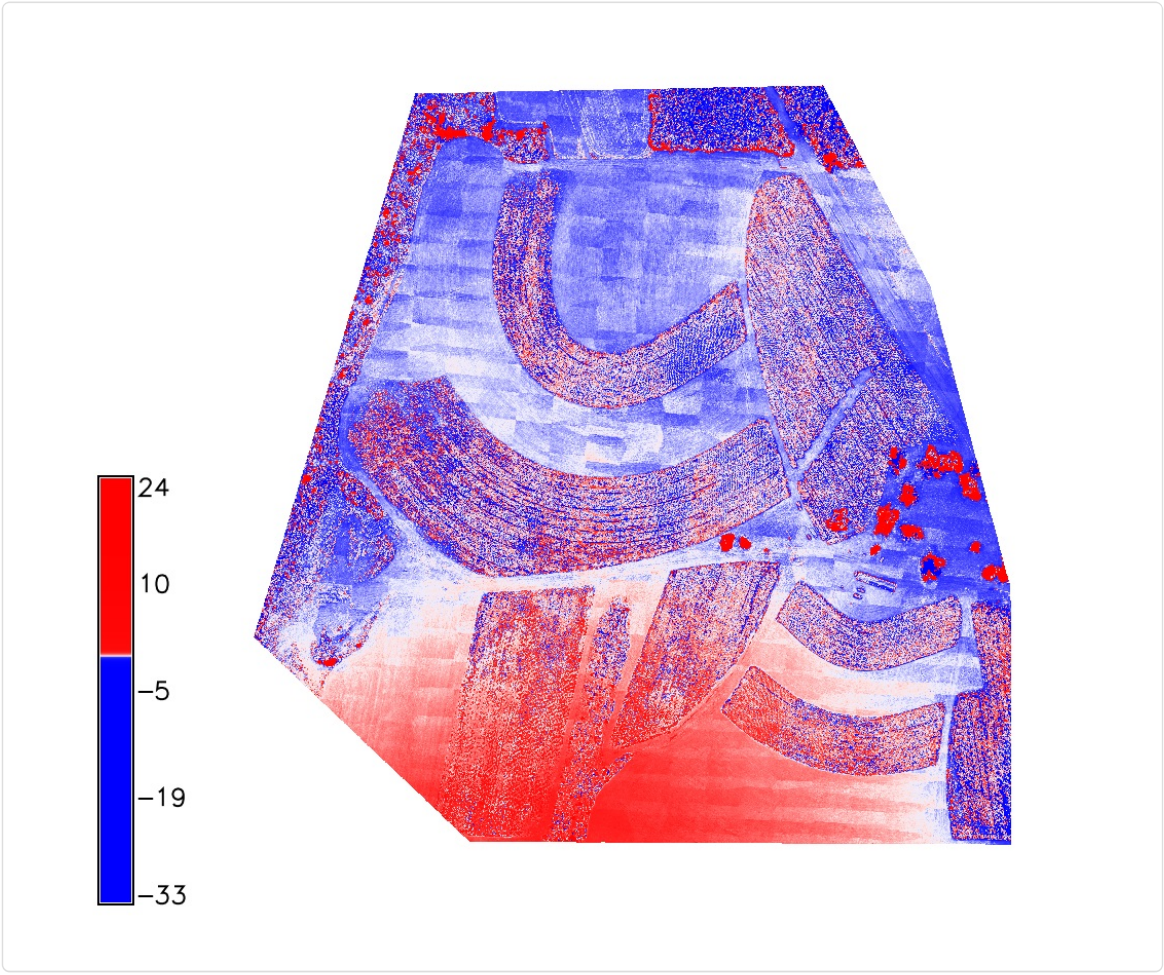
Agisoft - Trimble



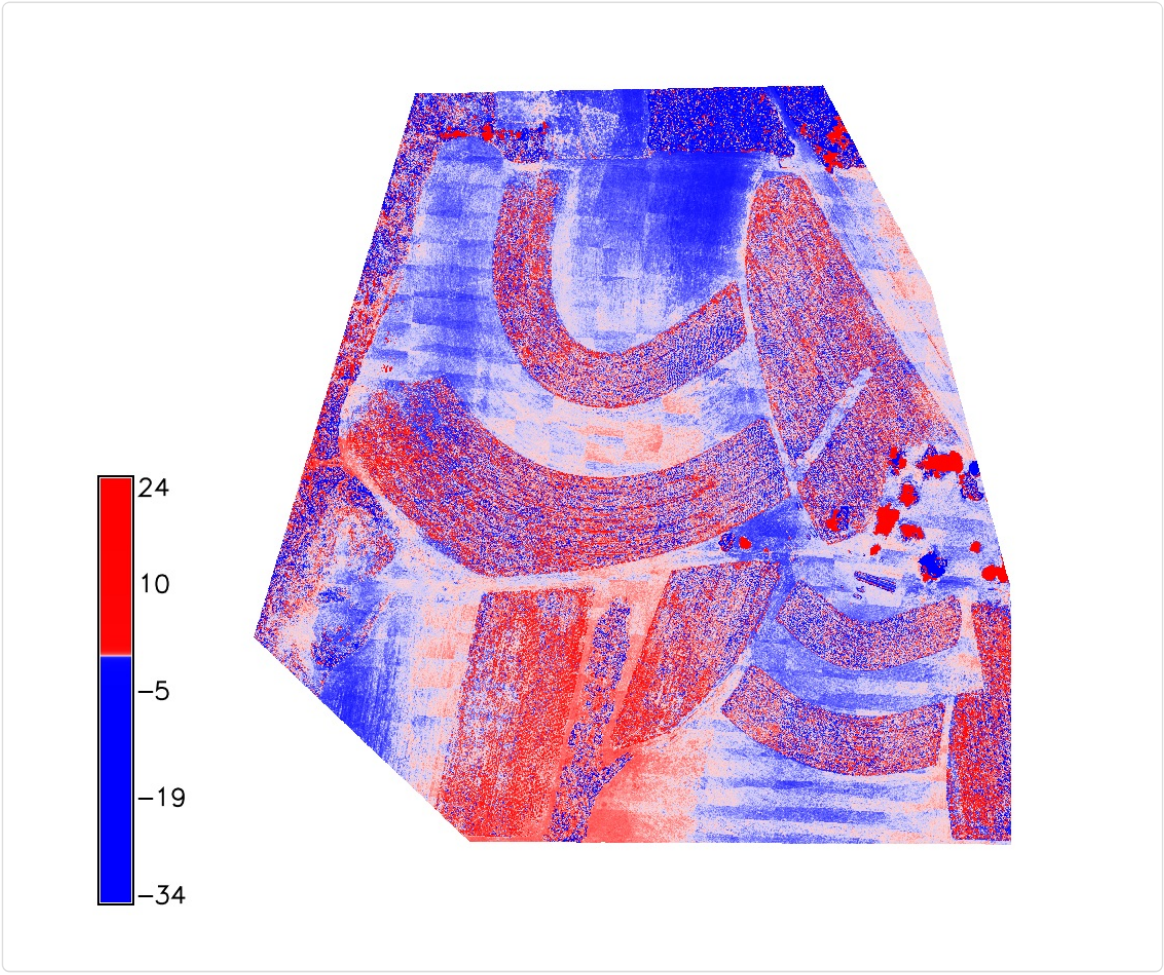
Pix4D - Agisoft



Pix4D - Trimble



Trimble - Agisoft



Trimble - Pix4D

Generating Reports

> Before re-running this code from `assignment7.py`, execute the following to remove the rasters that are created in this section:
>
> ```
> g.remove type=raster pattern="*_shade" -f
> ```

In this section, we'll use python to create images, calculate statistics, and generate a report in HTML format. View the resulting report here.

`generate_report.py`

```
import os
import grass.script as gscript

def generate_report( workingDir = '.' ):

    # Move to the working directory
    os.chdir( workingDir )

    # Get list of rasters we are interested in using a search pattern
    rasters = gscript.list_strings('raster', pattern='2015_06_*')

    # Initialize a dictionary to hold statistics for each raster
    stats = {}

    # Iterate through the list of rasters
    for raster in rasters:

        # Save univariate statistics for raster to dictionary
        stats[raster] = gscript.parse_command('r.univar', map=raster, flags='g')

        # Compute shaded relief, use name of the original raster including mapse
t
        shaded_relief = raster.replace('@', '_') + '_shade'
        gscript.run_command('r.relief', input=raster, output=shaded_relief,
                            overwrite=True)

        gscript.run_command('r.colors', map=raster, color='elevation')

        # Render the raster (geographical extent follows current region)
        gscript.run_command('d.mon', start='cairo', output=raster + '.png',
                            width=400, height=400, overwrite=True)
        gscript.run_command('d.shade', shade=shaded_relief, color=raster)
        gscript.run_command('d.mon', stop='cairo')

    # The HTML template that will be used for each raster map
    template = """<h2>Raster map {name}</h2>
    <h3>Statistics</h3>
    <table>
    <tr><td>Mean</td><td>{mean}</td>
    <tr><td>Variance</td><td>{var}</td>
    </table>
    <h3>Image</h3>
    <img src="{name}.png">
    """

    # write to a file using a template
    with open('report.html', 'w') as output:
            for raster in sorted(rasters):
                stat = stats[raster]
                output.write(template.format(
                    name=raster, mean=stat['mean'], var=stat['variance']))
```

Creating Terrain Profiles

In the final section, we'll use python to automatically sample any number of rasters for any number of terrain profiles, optionally saving statistical data to csv files and displaying plots. The `terrain_profiles()` function takes three arguments. The first is the list of rasters that will be sampled. The second is a list of lists of coordinates along which profiles are sampled (see the values used in `assignment7.py` for an example). The third is an optional list of filenames (each corresponding to a single profile) to which simple statistical data will be saved.

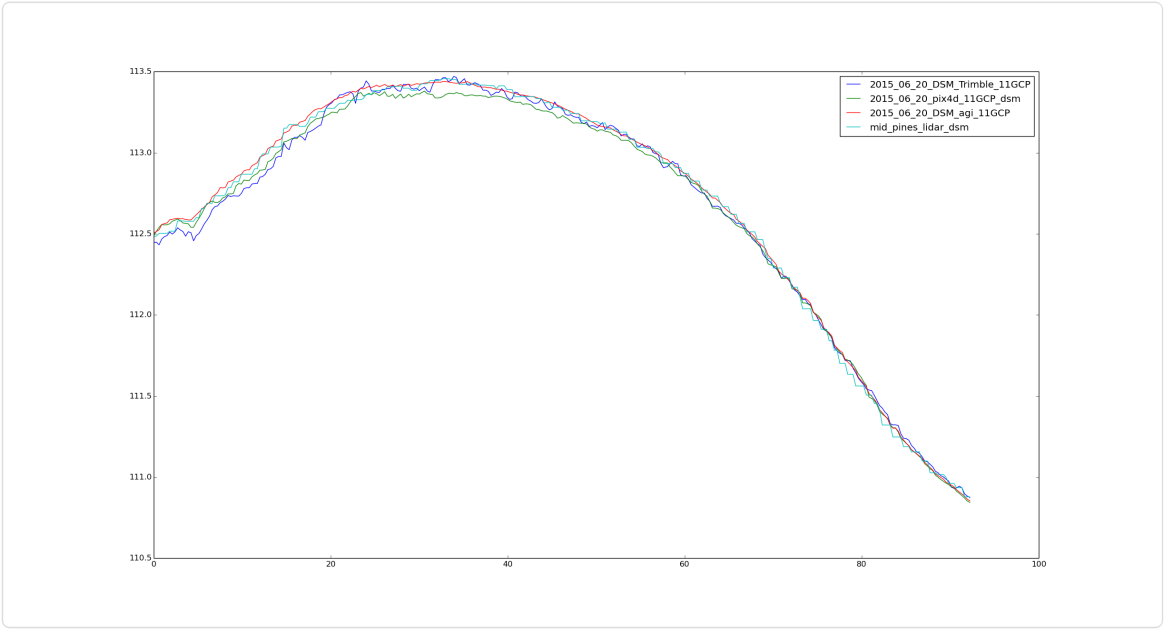
`terrain_profiles.py`

```
import os
import grass.script as gscript
import matplotlib.pyplot as plt
import numpy as np

# Plots terrain profiles in each raster and optionally generates
# statistics reports for each profile.
# - rasters: list of rasters to use for each profile
# - coordinates_lists: list of lists of coordinates along which profiles are dra
wn
# - stat_files: list of file names for stat files (one for each coordinate list)
def terrain_profiles( rasters, coordinates_lists, stat_files = [] ):

    # Loop through each list of coordinates
    for c in range( len( coordinates_lists ) ):

        # The current list of coordinates
        coordinates = coordinates_lists[c]

        # Create a dictionary that will store this profile of each raster
        profiles = {}

        # Loop through the rasters
        for raster in rasters:

            # Get the profile
            profile = gscript.read_command('r.profile', input=raster,
                                           coordinates=coordinates,
                                           quiet=True).strip()

            distance = []
            elevation = []

            # Parse profile data
            for line in profile.splitlines():
                dat = line.split()
                if dat[-1] != '*':
                    distance.append( float( dat[0] ) )
                    elevation.append( float( dat[-1] ) )
```

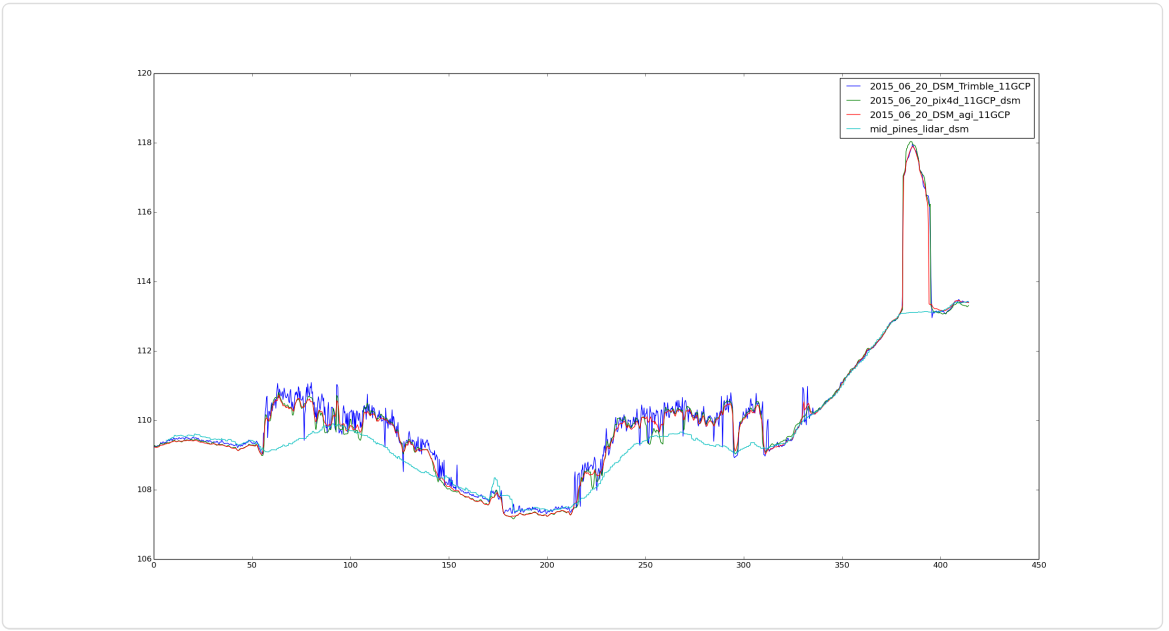
```
            # Save the profile in the dictionary
            profiles[raster] = (distance, elevation)

        # Is there a filename provided for the current coordinate list?
        if c < len( stat_files ):

            # Open the file for writing
            with open( stat_files[c], 'w' ) as f:

                # Write the header line
                f.write(','.join(['name', 'minim', 'maxim', 'mean',
                                  'stddev', 'median', 'roughness']))
                f.write('\n')

                # Create a dictionary to hold the stats
                stats = {}

                # Calculate statistics for each raster profile
                for raster in profiles:

                    profile = np.array( profiles[raster][1] )
                    maxim = np.max( profile )
                    minim = np.min( profile )
                    mean = np.mean( profile )
                    stddev = np.std( profile )
                    median = np.median( profile )
                    roughness = np.std( np.diff( profile ) )
                    stats[raster] = ( minim, maxim, mean, stddev, median, rough
ness )

                    # Write the statistics to the file
                    f.write(raster + ',')
                    f.write(','.join([str(i) for i in stats[raster]]))
                    f.write('\n')

        # Add each profile to the plot
        for raster in profiles:

            plt.plot( profiles[raster][0], profiles[raster][1], label=raster )
            plt.legend( loc=0 )

        # Show the plot
        plt.show()
```

Note that when we parse the profile data from each raster, we cannot assume that every profile will have the same number of values. If the profile line crosses an empty cell, that cell will return a null value in `r.profile`. Therefore, we need to keep track of both the distance and elevation data for each profile line of each raster, rather than using the same distance data for a profile line over all of the rasters (as is done on the assignment page).

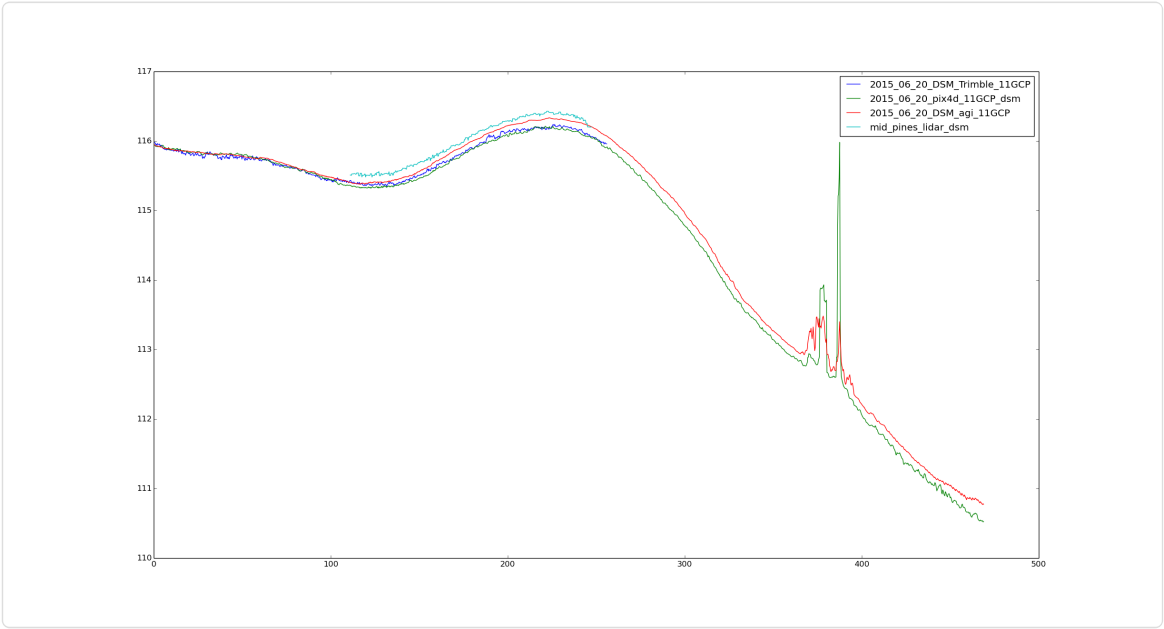
Running this function with the example values used in `assignment7.py` results in the following profiles.



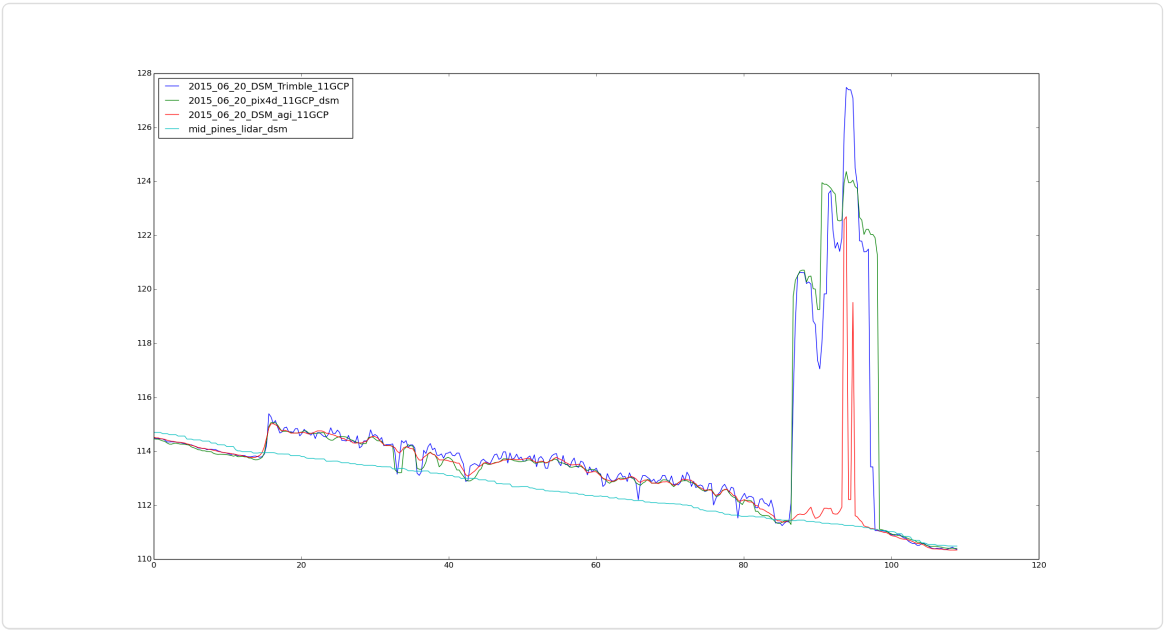
Profile line from assignment page



Profile in southern field



Profile along road



Profile through corn and over tree